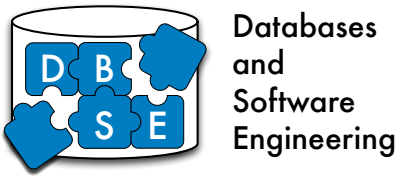
University of Magdeburg

School of Computer Science



Dissertation

Similarity-Driven Prioritization and Sampling for Product-Line Testing

Author:

Mustafa Zaid Saleh Al-Hajjaji

August 15, 2017

Reviewers:

Prof. Dr. Gunter Saake (University of Magdeburg, Germany)

Prof. Dr. Andy Schürr (University of Darmstadt, Germany)

Prof. Ebrahim Bagheri (Ryerson University, Canada)

Al-Hajjaji, Mustafa Zaid Saleh:
Similarity-Driven Prioritization and Sampling for Product-Line Testing
Dissertation, University of Magdeburg, 2017.

Similarity-Driven Prioritization and Sampling for Product-Line Testing

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Mustafa Zaid Saleh Al-Hajjaji

geb.am 01.01.1985 in IBB-YEMEN

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Andy Schürr
Prof. Ebrahim Bagheri

Magdeburg, den 15.08.2017

Abstract

A software product line comprises a family of software products that share a common set of features. Testing an entire product-line product by product is infeasible, because the number of possible products can be exponential in the number of features. Combinatorial interaction testing is a sampling strategy that selects a presumably minimal, yet sufficient number of products to be tested. Several sampling approaches have been proposed, however, they do not scale well to large product lines, as they require a considerable amount of time to compute the samples. In addition, the number of generated products can still be large, especially if the product line has a large number of features. Since the time budget for testing is limited or even a priori unknown, the order in which products are tested is crucial for effective product-line testing to increase the probability of detecting faults faster. Hence, we propose 1) product prioritization to increase the probability of detecting faults faster and 2) incremental sampling to generate samples in a step-wise manner. Regarding product prioritization, we propose similarity-driven product prioritization that considers problem-space information (i.e., feature selection) and solution-space information (i.e., delta modeling) to select the most diverse product to be tested next. With respect to sampling, we propose an incremental algorithm for product sampling called IncLing, which enables developers to generate samples on demand in a step-wise manner. The results of similarity-driven product prioritization show a potential improvement in the effectiveness of product-line testing (i.e., increasing the early rate of fault detection). Moreover, we show that applying the algorithm IncLing to sample products enhances the efficiency of product-line testing compared to existing sampling algorithms. Thus, we conclude that in which order these products are generated as well as tested may enhance the product-line testing effectiveness.

Zusammenfassung

Eine Softwareproduktlinie ist eine Familie verwandter Softwareprodukte, die eine gemeinsame Menge von Features teilen. Das Testen einer gesamten Produktlinie durch die Generierung und Testen aller möglichen Produkte ist nicht praktikabel, da die Anzahl der möglichen Produkte exponentiell zur Anzahl der Features wachsen kann. Combinatorial Interaction Testing ist eine Sampling-Strategie, bei der eine möglichst minimale, aber ausreichende Anzahl an Produkten getestet wird. In der Vergangenheit wurden bereits verschiedene Ansätze vorgeschlagen, die die Strategie des Combinatorial Interaction Testing anwenden. Leider skalieren diese Ansätze nicht für große Produktlinien, da die Auswahl geeigneter Samples eine erhebliche Menge an Zeit benötigt. Außerdem kann die Anzahl der generierten Produkte trotzdem groß sein, insbesonders wenn die Produktlinie eine große Anzahl an Features enthält. Da die Zeit zum Testen begrenzt oder a priori unbekannt ist, kann ggf. nur eine begrenzte Anzahl an Produkten getestet werden. Für ein effektives Testen mit einer hohen Wahrscheinlichkeit Fehler zu erkennen ist eine geeignete Reihenfolge in der die Produkte einer Produktlinie getestet werden entscheidend. Deshalb schlagen wir 1) Product Prioritization vor um die Wahrscheinlichkeit zu erhöhen Fehler schneller aufzudecken. Im Speziellen schlagen wir Similarity-Driven Product Prioritization vor, das Problem- und Solution-Space-Informationen berücksichtigt, um ein möglichst unterschiedliches Produkt für den nächsten Test auszuwählen. Hierbei betrachten wir die Auswahl der Features als Problem-Space-Informationen und Delta-Modellierung als Solution-Space-Informationen. Des Weiteren schlagen wir 2) Incremental Sampling vor um Samples schrittweise zu generieren. Im Speziellen schlagen wir IncLing vor, einen inkrementellen Algorithmus, welcher es Entwicklern ermöglicht Samples schrittweise je nach Bedarf zu generieren. Die Ergebnisse der Similarity-Driven Product Prioritization zeigen eine mögliche Verbesserung der Effektivität von Produktlinien-Tests durch eine frühere Erkennung von Fehlern. Außerdem, zeigen wir, dass die Verwendung von IncLing zum Sampling von Produkten, die Effizienz von Produktlinien-Tests im Vergleich zu existierenden Sampling-Algorithmen verbessert. Daher, schlussfolgern wir, dass die Wahl der Reihenfolge in der die Produkte generiert und getestet werden die Effektivität von Produktlinien-Tests erhöhen kann.

Acknowledgments

Pursuing a Ph.D. is a long journey, and it would not be possible to reach the end of that journey without the help of many people. Here, I would like to thank everyone who helped me. Without their support, this thesis would not have come into existence.

First, I would like to express my deep gratitude to Gunter Saake for giving me the opportunity to pursue my Ph.D. under his supervision. He provided an excellent research environment and gave me the chance to choose a research topic of my own choice. He always came up with practical solutions whenever I faced scientific or organizational challenges.

Second, I want to thank Thomas Thüm for his support from the first day of my Ph.D. study. His advice, comments, and feedback improved my scientific writing skills. During the last four years, we had many brainstorming sessions that had a major impact on my research.

Third, to all my colleagues with whom I had the chance to collaborate and discuss, I want to express my deepest gratitude for their support and fruitful discussions. In particular, I thank Sebastian Krieter, Fabian Benduhn, Jacob Krüger, Jens Meinicke, Sandro Schulze, Reimar Schröter, Wolfram Fenske, Gabriel Campero Durand, and David Broneske. I also thank Andreas Meister, who I shared the office with four years for being supportive, good discussion partner, and colleague to have a laugh with.

Fourth, I would like to thank many researchers I was lucky to meet at my path and interact with, especially Malte Lochau, Sascha Lity, and Thomas Leich. Their concert and insightful comments had a great impact on my research. Special thanks to Andy Schürr and Ebrahim Bagheri for being the reviewers of my thesis.

Last but not the least; I want to thank my wife Eman, son Azzam, parents, brothers, and sisters for their encouragement, love, support, and understanding in good and bad times. I am very grateful to my parents who supported me, my dreams, and plans from the very beginning of my life.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1. Introduction

Today, many software systems are developed in a large variety of similar variants to meet different requirements of customers. Typically, these variants are built by adapting existing ones instead of being developed from scratch [Antkiewicz et al., 2014; Hemel and Koschke, 2012; Laguna and Crespo, 2013]. Software product-line engineering is a technique empowering software companies to develop software systems that meet the requirements of individual customers. Using product-line engineering, software systems can be developed efficiently by considering their similarities and differences in terms of features to facilitate systematic reuse [Clements and Northrop, 2001; Apel et al., 2013a; Pohl et al., 2005; Czarnecki and Eisenecker, 2000]. A feature is defined as an increment to the functionality that is recognized by customers [Batory et al., 2004; Kang et al., 1990].

A software product line is defined as a set of software systems that share many features, but also differ in others. It aims to reduce development and maintenance costs, increase quality, and decrease time to market [van der Linden et al., 2007]. Hence, several companies such as Hewlett-Packard, Toshiba, and General Motors have adapted their software development process to software product-line engineering [Apel et al., 2013a; Weiss, 2008]. Despite the aforementioned potential advantages of product-line engineering, it poses new challenges for quality assurance [Thüm et al., 2014a; Lee et al., 2012].

Testing is a necessary step in software development processes. It consumes at least 50% of the whole development costs [Harrold, 2000; Willcock, 2011]. While testing a single software system is already difficult, testing product lines is even more challenging as the number of products is up-to exponential in the number of features. Ideally, every product should be tested, especially for safety-critical systems, to ensure that it meets its specifications. However, given a product-line implementation, one major challenge is that it is difficult to guarantee that all its features and every possible combination of them in a product will work as expected. The reason for this difficulty is the following:

- The number of possible products potentially grows exponentially in the number of features.
- Testers usually have a limited amount of time and resources to run tests on a specific product.
- Every further derivation of a product under test causes additional costs (since this may include assembling of software and hardware, in the case of industrial systems).

Reducing the number of products to test as well as the number of executed test cases on products under test while still maintaining a desired level of coverage is required [Harrold, 2000]. Thus, several approaches such as combinatorial interaction testing [Cohen et al., 2007] have been proposed. Combinatorial interaction testing is an approach used in product-by-product testing to systematically reduce the number of products under test [Oster et al., 2010; Perrouin et al., 2012, 2010]. In particular, T-wise combinatorial interaction testing is based on the observation that most faults are caused by interactions among a fixed number of at most T features [Johansen et al., 2012a; Kästner et al., 2009]. Consequently, combinatorial interaction testing is used to sample a presumably minimal subset of configurations that cover each T-wise interaction by at least one product under test [Cohen et al., 2007]. This optimization problem is NP-hard [Engebretsen, 2005], and several heuristics have been proposed to perform T-wise sampling such as CASA [Garvin et al., 2011], Chvatal [Johansen et al., 2011; Chvatal, 1979], ICPL [Johansen et al., 2012a], IPOG [Lei et al., 2007], and MoSo-PoLiTe [Oster et al., 2010, 2011c]. The output of these algorithms is a presumably minimal product sample to be tested to obtain T-wise combinatorial interaction testing coverage. Moreover, the aforementioned algorithms approximate the solutions to this constraint optimization problem. However, those sampling algorithms do not scale well to larger product lines in terms of CPU time and memory consumption [Medeiros et al., 2016; Henard et al., 2014b]. In addition, even for small sets of features, these algorithms require a considerable amount of time until the first product of the sample can be tested as the complete sample is computed without delivering intermediate products (i.e., one by one) until a sufficient feature interaction coverage is reached.

Although sampling algorithms can reduce the number of products significantly, the number can still be large, especially if the product line has a large number of features. As the time budget in practice is limited in testing, the order in which products are tested matters to presumably find faults as soon as possible within the available amount of time. For example, testing two almost identical products is not likely to detect different faults. For this purpose, several approaches have been proposed to prioritize products based on different criteria [Devroey et al., 2014; Baller et al., 2014; Johansen et al., 2012b; Henard et al., 2013a; Sánchez et al., 2015; Devroey et al., 2016]. However, as reported by Elbaum et al. [2004], no prioritization criterion is the best for any system as their effectiveness varies based on the considered systems. To overcome the challenges of having a large number of products as well as the scalability of the existing

sampling algorithms, we propose two possible solutions: product prioritization and efficient incremental sampling.

1.1 Contribution

We propose similarity-driven product prioritization that considers problem-space as well as solution-space information to prioritize products. With similarity-driven product prioritization, we incrementally select the least similar product to be tested next in order to increase feature interaction coverage faster during product-by-product testing. Based on the information type, we recognize two instances of this approach, configuration-based and delta-oriented prioritization. With configuration-based prioritization, we select the next product to test using problem-space information in terms of feature selection. However, with delta-oriented prioritization, we consider solution-space information in terms of delta modeling to select the next product. In case the diversity between products is large, testing efforts may increase as a result of the redundancy in the test-case execution, especially in regression testing [Lity et al., 2017]. Thus, we propose cluster-based product prioritization as a trade-off approach, where we group products into clusters. The products in each cluster share common features. This allows also clustering products based on particular parameters (e.g., the most demanded products). The products in a particular cluster might be ordered using different prioritization techniques. In this thesis, we prioritize products in clusters using our configuration-based prioritization approach.

With respect to sampling, we propose **Inc**remental samp**Ling** (IncLing) that generates products one at a time with the aim of enhancing the sampling efficiency, in terms of the required time to generate a sample, as well as testing effectiveness, in terms of the interaction coverage rate. With IncLing, we take already generated and tested products into account while selecting further products into the sample. At the beginning of the sampling process, we efficiently detect the invalid feature combinations that will not appear in any products due to the dependencies between features. Then, our greedy algorithm generates the next product that covers as many of uncovered feature combinations as possible. In particular, we increase the diversity among products by covering dissimilar pairwise feature combinations each time a further product is generated to be tested. Increasing the covering rate of feature combinations might lead to a faster fault detection. This way, we dynamically generate further products into the sample until the testing time is over or a certain degree of coverage is achieved.

1.2 Structure of the Thesis

This thesis is structured as follows. In Chapter 2, we give a brief introduction to software product line-engineering as well as combinatorial interaction testing. In addition, we introduce readers to test-case prioritization.

In Chapter 3, we propose our configuration-based and cluster-based prioritization approaches that consider problem-space information in terms of feature selections to pri-

oritize products. Using product lines of various sizes, we evaluate the increase in effectiveness (i.e., finding faults as soon as possible) of both approaches compared to random orders and the default order of existing sampling algorithms.

In Chapter 4, we introduce delta-oriented product prioritization that considers the solution-space information with respect to deltas to prioritize products without generating them. In addition, we propose a combined approach that considers feature selection and deltas in product prioritization. Furthermore, we investigate the effectiveness of our proposed approach compared to a default order of a sampling algorithm and random orders using an automotive product line.

In Chapter 5, we present our incremental sampling algorithm (IncLing) that generates products one at a time to enhance the sampling efficiency. Moreover, we present and discuss the reported results of comparing our algorithm against existing algorithms using feature models of different sizes.

In Chapter 6, we conclude the thesis and discuss potential future work. In the Appendix, we present the functionalities of FeatureIDE that we implemented to support testing product lines (cf. Section A.1). In addition, we include other detailed results (cf. Section A.2) as well as supplementary material (cf. Section A.3) that support our evaluation.

2. Background

This chapter shares material with SoSyM'16 article "Effective Product-Line Testing Using Similarity-Based Product Prioritization" [Al-Hajjaji et al., 2016d].

In this chapter, we present background on software product-line engineering, including the development process of product lines and the modeling of product-line commonalities and differences. Furthermore, we give an overview of combinatorial interaction testing as well as test-case prioritization.

2.1 Software Product-Line Engineering

Using software product-line engineering, companies are able to compose software systems that meet the requirements of individual customers by considering their commonalities and differences in terms of features to facilitate systematic reuse [Clements and Northrop, 2001; Apel et al., 2013a; Pohl et al., 2005; Czarnecki and Eisenecker, 2000].

Features are distinctive aspects or characteristics of software systems that are recognized by customers [Kang et al., 1990]. A general definition of a feature is introduced by Apel et al. [2013a], where they define it as *"a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle".* The commonalities and differences of products need to be managed as they usually define the domain of the market segment of a product line. Clements and Northrop [2001] describe a software product line as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".*

As a result of the systematic reuse, software product-line engineering can reduce the time-to-market as well as the costs for development and maintenance, while the quality of the generated products is increased. The notion of product-line engineering is not new as it is already applied successfully in the automotive industry for several years to handle the individual needs of customers. Software product lines follow the same notion of product line engineering in the automotive domain, but focus on reusing the software artifacts [Apel et al., 2013a]. Previously, a similar concept defined as *program family* has been proposed with the aim of reducing the cost of development and facilitating the systematic reuse of artifacts [McIlroy, 1968; Dijkstra, 1976; Parnas, 1976]. For instance, McIlroy [1968] suggests avoiding the redundancy in the implementation by handling the implemented components as interchangeable parts. Similarly, Parnas [1976] defines a program family as a set of *"programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members"*. Recently, software product lines are gaining widespread acceptance in the industry. Several companies such as Bosch, Philips, Siemens, General Motors, Hewlett-Packard, Boeing, and Toshiba apply product-line approaches to facilitate reuse [Weiss, 2008; Northrop and Clements, 2007; Apel et al., 2013a].

We discuss the main development process of a software product line in Section 2.1.1. In Section 2.1.2 we introduce feature models [Kang et al., 1990] that are used to model the commonalities and differences in a product line. In Section 2.2, we introduce the task of testing product lines including combinatorial interaction testing (cf. Section 2.2.1) and covering array algorithms (cf. Section 2.2.2). In Section 2.2.3, we give an overview on test-case prioritization in single systems as well as in software product lines.

2.1.1 The Development Process of Software Product Lines

Regardless of the specific development process used in single-system software engineering, the usual life cycle of a single system includes the following: First, the requirements for the target system are collected and analyzed. Second, developers design and implement the system based on these requirements. The implementation process could be carried out in a single separate phase, consecutive phases or in agile cycles. Third, the system is passed through a testing process. After releasing the system, the maintenance phase is launched. Compared to the development process of single software engineering, the development process of software product lines is different, as variety of desirable systems, which are similar but not identical, are considered. In particular, the development process of product lines, as illustrated in Figure 2.1, can be divided into two main tasks, *domain engineering* and *application engineering* [Weiss and Lai, 1999; van der Linden, 2002].

The domain is the knowledge area that defines the requirements of customers as well as the way software systems will be built to fulfill these requirements [Czarnecki and Eisenecker, 2000]. Pohl et al. [2005] defines domain engineering (cf. top half of Figure 2.1) as *"a process of software product line engineering in which the commonality and the variability of the product line are defined and realized"*. One of the main key points

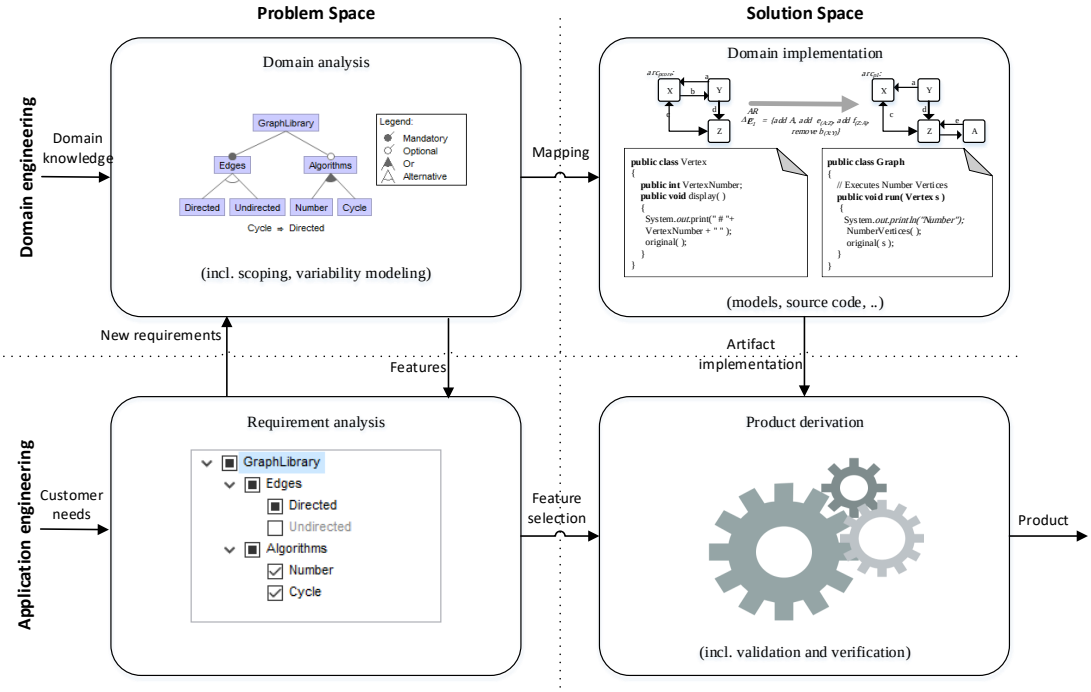


Figure 2.1: Overview of an engineering process for software product lines [Apel et al., 2013a]

of product-line development is to define well the domain and its scope [Czarnecki and Eisenecker, 2000].

The outcomes of domain engineering are artifacts that are used in several, if not all, products of a product line [Apel et al., 2013a]. The commonality and the variability mentioned in the definition of Pohl et al. [2005] are typically represented by a widely used variability modeling technique, called *feature model*. We refer to the common and variable functionality as well as the artifacts of a set of similar products as features. In contrast to domain engineering, which aims to enable development for reuse, application engineering targets the goal of developing a specific product that satisfies the need of a particular customer. Pohl et al. [2005] defines application engineering (cf. bottom half of Figure 2.1) as a *"process of software product-line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability"*.

Furthermore, in product-line development, the separation between problem space and solution space is recognized [Czarnecki and Eisenecker, 2000]. The problem space (cf. left half of Figure 2.1) considers the viewpoints of customers, including their requirements and needs. It also displays the entire domain, where features describe the problem space. In contrast, the solution space takes the perspectives of developers. It

refers to concrete products developed during the architecture, design and implementation phases.

Based on the distinctions between domain and application engineering as well as problem and solution space (cf. Figure 2.1), we recognize four main tasks in product-line development:

- Domain analysis deals with the requirements of the entire product line. In this task, the decision of including features, which should be implemented as reusable artifacts, in a product line is taken. The outcome of this task is usually documented in the form of a feature model. In this thesis, we use feature models as an input to the sampling algorithms to generate products of a product line.
- Requirements analysis explores the needs of a particular customer by mapping their requirements to the feature selections. These feature selections, which represent problem-space information, are used to prioritize products (cf. Chapter 3).
- Domain implementation is the process of developing artifacts that are mapped to features recognized in the domain analysis. There are several types of artifacts in product lines related to architecture, design, documentation, and test. In this thesis, we consider the architecture model that represents the solution-space information as an input to our product prioritization approach (cf. Chapter 4).
- Product derivation is the process of generating products by combining artifacts based on the results of requirements analysis. In our work, we derive products for the purpose of testing in order to ensure that the features and the combinations between them work as expected (cf. Chapter 5).

In the next section, we introduce feature modeling as an activity that is used to model the variability and commonality of a product line.

2.1.2 Feature Modeling and Configurations

Feature models consist of a tree-like hierarchical structure that captures the information of all possible configurations of a product line in terms of features and relationships among them. These models are usually represented graphically by feature diagrams [Kang et al., 1990]. Figure 2.2 shows the feature model of our running example, the *GraphLibrary* product line. The feature *GraphLibrary* is the root of the feature diagram and it represents the common part of all products of the product line. Selection of a feature in a configuration implies the selection of its parent feature in the same configuration.

Each feature has to be either optional or mandatory. The mandatory features must be included in each created configuration if their parent feature is selected. The optional features may or may not be selected in case their parent feature is already part of a configuration. An example of a mandatory feature in Figure 2.2 is feature *Edges*,

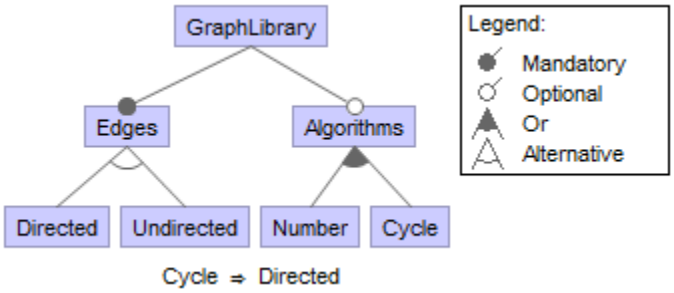


Figure 2.2: Feature diagram of product line *GraphLibrary*

while feature *Algorithms* represents an example of an optional feature. In addition, the features can be grouped into *alternative* and *or* groups. Only one feature of an *alternative* group must be selected in a configuration. For instance, a configuration can contain only one of the features *Directed* and *Undirected*. From features in an *or* group, at least one of them must be selected in a configuration. For instance, features *Number*, *Cycle*, or both can be selected in a configuration.

In addition to the dependencies defined by the hierarchical structure, further dependencies between features can be defined by so-called *cross-tree constraints*. Examples of these constraints are *Require* and *Exclude*. A *Require* constraint, the selection of a feature implies the selection of another feature. For instance, in our running example (cf. Figure 2.2), the selection of feature *Cycle* in a configuration implies the selection of feature *Directed*. Regarding *Exclude* constraint, the features are involved in such constraints cannot be both selected in a configuration.

Propositional Formulas

All feature models can be represented in a form of propositional formulas, where each feature is represented as a Boolean variable [Batory, 2005]. The logical operators are used to connect those Boolean variables in order to model the dependencies between features. To analyze feature models, algorithms often use propositional formulas as input, because most of the tasks that could be applied to feature models can be reduced to well-known problems in Boolean algebra.

In Figure 2.3, we show the propositional formulas of the feature model of our running example. The relationship between features can be represented in propositional formulas as follows. The mandatory features (e.g., Equation 2.2) as well as the parent-child relationship (e.g., Equation 2.3) can be expressed as an implication (i.e., $\Leftrightarrow$ and $\Rightarrow$, respectively). *Or* groups are represented by the logical operator *OR* between features. That is, they are expressed using disjunctions (e.g., Equation 2.4). Similar to *or* groups, *alternative* groups are represented with the logical operator *or* with an additional condition, all children exclude each other. The additional rule is represented as a set of pairwise disjunctions (e.g., Equation 2.5). Additional constraints can also be defined as propositional formulas using logical operators, such as conjunction ($\wedge$), disjunction

$GraphLibrary$	root feature	(2.1)
$\wedge(GraphLibrary \Leftrightarrow Edges)$	mandatory feature	(2.2)
$\wedge(Algorithms \Rightarrow Edges)$	optional feature	(2.3)
$\wedge((Number \vee Cycle) \Leftrightarrow Algorithms)$	OR-group	(2.4)
$\wedge(((Edges \Leftrightarrow (Directed \vee Undirected))$	alternative-group	(2.5)
$\wedge\neg(Directed \wedge Undirected))$		
$\wedge(Cycle \Rightarrow Directed)$	cross-tree constraint	(2.6)

Figure 2.3: Propositional formula of the *GraphLibrary* feature model

($\vee$), negation ($\neg$), and implication ($\Rightarrow$). However, the algorithms that work on feature models often consider the *conjunctive normal form* (CNF) of a propositional formula as input.

Conjunctive Normal Form

All propositional formulas can be represented as Conjunctive Normal Forms (CNFs). Using logical operators disjunction, conjunction, and negation placed in a certain order, CNF consists of a conjunction of clauses that contain a disjunction of single positive or negative variables. These positive and negative variables, which indicate the selection state of each feature, are defined as literals L (i.e., L = $\{l_1, l_2, ..., l_n\}$). A feature should be selected, if it is represented as l_i and it should not if it is represented as $\neg l_i$. The negation is allowed only for a single variable, not for the whole clause or the entire formula. For instance, the constraint $Cycle \Rightarrow Directed$ can be written in a CNF as $\neg\ Cycle \vee Directed$. In Figure 2.4, we show the propositional formula in CNF for the *GraphLibrary* feature model. We rely in our thesis on CNFs of feature models as input to our proposed sampling algorithm.

A formal definition of feature models is described as follows:

Definition 2.1. *(Feature Model Definition).*
A feature model $\mathcal{FM} = (F, R)$ is defined as a tuple of a set features F and a set of constraints R, where F contains all features of the product line (i.e., F=$\{f_1, f_2, ..., f_n\}$, n is the number of features), and R is a set of constraints represented as clauses, where each contains pair of variables (i.e., features).

As mentioned, a configuration is a combination of a set of selected features from a feature model. Formally a configuration c is defined as following:

Definition 2.2. *(Configuration Definition).*
A configuration c for a feature model $\mathcal{FM} = (F, R)$ is a set of literals in L, such that $c \subset L$ and $\forall i \in \{1,2,...,|F|\} : \{l_i, \neg l_i\} \nsubseteq c$, where l_i indicates that the feature should be selected and $\neg l_i$ indicates that the feature should not be selected.

A feature can be *defined* or *undefined* based on its literal value. The feature is *defined* if its state in a configuration is specified (i.e., selected l_i or not selected $\neg l_i$); otherwise

$$
\begin{aligned}
&GraphLibrary \\
&\wedge(\neg Edges \vee GraphLibrary) \wedge (Edges \vee \neg GraphLibrary) \\
&\wedge(\neg Algorithms \vee Edges) \\
&\wedge(\neg Number \vee Algorithms) \wedge (\neg Cycle \vee Algorithms) \\
&\quad \wedge(Number \vee Cycle \vee \neg Algorithms) \\
&\wedge(\neg Edges \vee Directed \vee Undirected) \wedge (Edges \vee \neg Directed) \wedge (Edges \vee \neg Undirected) \\
&\quad \wedge(\neg Directed \vee \neg Undirected) \\
&\wedge(\neg Cycle \vee Directed)
\end{aligned}
$$

Figure 2.4: Propositional formula of the *GraphLibrary* feature model in CNF

it is *undefined*. That is, if every feature in a configuration c is defined, we call this configuration as *complete*, and it is partial otherwise.

As a result of feature dependencies and constraints, a feature can be a conditionally *dead* or *core* if it is under certain circumstances (i.e., features that must be selected or deselected given the feature model and already fixed features of the current configuration) [Benavides et al., 2010]. For instance, if feature *Directed* is selected, feature *Undirected* will be a dead feature. The feature *Directed* may also be core, if feature *Cycle* is selected.

Feature models can be used to restrict the variability of a product line as not all combinations of features are valid. A combination of a set of features is a valid configuration if it does not violate the feature dependencies and the constraints defined in the feature model. For instance, with respect to Figure 2.2 on Page 9, c = {GraphLibrary, Edges, Directed, Algorithms, Number, Cycle} is a configuration representing a software product that includes all features except feature *Undirected*. This configuration is valid, because it satisfies all the constraints and the feature dependencies defined in the feature model. On the contrary, c = {GraphLibrary, Edges, Algorithms, Number, Cycle} is not valid, because it violates the constraint $Cycle \Rightarrow Directed$. Each configuration can be used to generate a product implementing the selected features.

Besides the definition of the configuration space, feature models are mainly used by sampling algorithms, such as CASA [Garvin et al., 2011], Chvatal [Johansen et al., 2011], ICPL [Johansen et al., 2012a], IPOG [Lei et al., 2007], and MoSo-PoLiTe [Oster et al., 2010], to derive representative subsets of configurations. In our thesis, we also use feature models as input to our approaches to achieve efficient sampling. The sampling algorithms apply combinatorial interaction testing to reduce the number of generated products to be tested while achieving a certain degree of coverage [Perrouin et al., 2010; Oster et al., 2010; Perrouin et al., 2012].

2.1.3 Implementation Techniques for Software Product Lines

There are several implementation techniques to realize product lines [Apel et al., 2013a]. These techniques specify the generation and transformation mechanisms that determine the source code for single products. In the following, we give a short overview of these implementation techniques.

Preprocessors

The preprocessor-based mechanism is a popular approach to implement product lines, relying on annotations. The C preprocessor (cpp) tool is commonly used to support this mechanism [Kästner et al., 2008]. In preprocessor-based mechanism, directives (a.k.a. macros) control syntactical program transformations. Examples of these directives are file inclusion (i.e., `#include` directive), macro definition (i.e., `#define`), and conditional inclusion (i.e., `#ifdef`) [Liebig et al., 2010]. These directives control the inclusion or exclusion of feature code.

The preprocessors-based mechanism has been widely used in practice to implement product lines. However, it has several weak points, such as neglecting the separation of concerns [Parnas, 1972; Dijkstra, 1976]. For instance, the code of a feature is scattered across the program code and also tangled with the code of other features [Apel et al., 2013a]. In addition, programs that have been implemented with preprocessors-based are difficult to analyze and also prone to simple errors [Liebig et al., 2010].

Feature-Oriented Programming

Feature-oriented programming is a composition-based approach that has been proposed to overcome limitations of the preprocessors-based mechanism, such as neglecting the separation of concerns, by providing feature modularization. It depends mainly on the notion of features to build software product lines [Apel et al., 2013a]. Prehofer [1997] proposed feature-oriented programming as an extension to object-oriented programming, where classes are decomposed to feature modules that implement a specific feature. Once the features are selected, the feature modules are composed to a program automatically. With feature-oriented programming, a feature has the ability to refine previous features in a flexible way using a keyword called *Original* [Apel et al., 2009].

Delta-Oriented Programming

Schaefer et al. [2010] proposed delta-oriented programming, where a preselected product, called core, and a set of delta modules are defined. These delta modules capture change operations that need to be applied to the core product in order to derive a new product. These operations include adding and removing domain artifacts to the core product. Given a configuration, the composer identifies the delta modules and applies them to the core module to generate a new product.

Aspect-Oriented Programming

Using a provided meta-language, aspect-oriented programming transforms existing object-oriented programs to modularize the crosscutting concern (i.e., extending a program at different places, which cuts across the module boundaries introduced by classes) [Kiczales et al., 1997]. This crosscutting leads to code scattering and tangling. Thus, aspect-oriented programming aims at reducing such code scattering and tangling induced by concerns that are not well-separated. The crosscutting concern problem is solved by

implementing these concerns as *aspects*, which encapsulates the implementation of crosscutting concerns. Using the *aspects*, the code linked to a crosscutting concern can be localized as a one code unit. Moreover, *aspects* can be used to make new features for existing programs. By applying a set of implemented aspects to an existing program, different products can be generated.

2.2 Software Product-Line Testing

Software testing is one of the most distinguished aspects of software quality assurance that can inform customers about the quality of systems. Testing is *"the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"* [IEEE, 1990]. With testing, a certain software system is executed with the intention of finding faults. A fault is defined as an incorrect step, process, or data definition in a computer program, that results often in an error and leads to a failure [IEEE, 1990].

Model-based testing [El-Far and Whittaker, 2002; Utting and Legeard, 2007] is a software testing technique to generate test cases from models that encode the intended behavior of the system under test. In particular, all testing activities are based on an executable formal test model of the expected behavior. Compared to traditional manual testing, the activities in model-based testing, such as generating test cases, are achieved in a systematic way considering predefined metrics (i.e., coverage). In addition, model-based testing enables testers to automate test execution as well as test evaluation by comparing the actual behaviors with the expected ones in the test model of software under test. Model-based testing can be a cost-effective technique that supports a variety of test generation strategies as well as model coverage criteria [Utting, 2008].

Product-line testing is more critical than testing a single system because a fault in a single feature can exist in thousands or even millions of products. Usually, testing a software product line includes creating configurations, building the corresponding products, testing them by executing their test cases, and using the results for debugging [Perrouin et al., 2010].

Several model-based testing strategies have been pursued to test product lines [Thüm et al., 2014a]. First, product-by-product testing follows a traditional strategy where concrete products are generated and tested individually using established testing techniques from single product testing. Second, regression-based product-line testing is a strategy where the new test cases for a product under test are generated and executed. These test cases are not reusable from previously tested products. Some of the already generated test cases are reused in case they cover components that have been changed or affected by a change. However, every single product has to be considered to guarantee a complete software product-line test coverage. Third, family-based testing is a strategy where all products are checked in a single run whether they satisfy their specifications [Bürdek et al., 2015]. In the family-based strategy, testing is achieved without considering any particular product. It uses a virtual representation of the

product line that simulates all products (i.e., by superimposing test specifications of all products). However, effective testing should not only investigate the software, but also its interplay with the hardware and the environment, which is not supported by recent family-based testing techniques. In addition, family-based testing can be time consuming and even incomplete (e.g., IO) as it requires a complex and specialized execution environment [Meinicke et al., 2016b; Nguyen et al., 2014; Apel et al., 2013c]. To overcome those limitations, we consider product-by-product testing instead of family-based testing.

2.2.1 Combinatorial Interaction Testing

Testing a product line exhaustively product-by-product is infeasible or even impossible due to the aforementioned exponential explosion of products and many redundant testing steps caused by the commonality among products. To alleviate this problem, combinatorial testing is used to reduce the effort of product-line testing by selecting a minimal, yet sufficient subset of products [Carmo Machado et al., 2014; Oster et al., 2010; Perrouin et al., 2010]. Concerning combinatorial interaction testing in particular, faults that are triggered by erroneous interactions between a certain number of features can be detected.

A feature interaction occurs when the integration of two or more features modifies or influences the behavior of other features [Jackson and Zave, 1998]. Two types of feature interaction can be recognized, positive and intended interaction as well as critical and inadvertent interaction [Ferber et al., 2002]. With positive and intended interaction, features are interacting to exchange information, reuse functionality of other features, or collaborate to achieve a certain task [Apel et al., 2013a]. However, the critical and the inadvertent interaction may lead to undesired results and even system failure. It is a challenging task in product-line testing to detect faults that are caused by those unintended and inadvertent feature interactions.

With T-wise combinatorial interaction testing, a kind of feature-combination coverage needs to be achieved. For instance, in pairwise combinatorial interaction testing (i.e., T=2), each valid combination of two features is required to appear in at least one configuration of the sample. In our running example, the valid combinations of feature *Directed* and feature *Undirected* are $Directed \wedge \neg Undirected$ and $\neg Directed \wedge Undirected$, and the invalid ones are $Directed \wedge Undirected$ and $\neg Directed \wedge \neg Undirected$. Each valid combination is required to appear at least in one of the created configurations of the sample.

Recently, T-wise testing has been generalized from pairwise testing to cover all T-wise combinations of features [Johansen et al., 2012a]. A trade-off exists between the time computation and test coverage. The higher the value of T, the higher is the test coverage, the more computation time is required to find a minimum number of products that covers all combinations of T features and as a result the number of products to be tested increases. In Table 2.1, we list five configurations that are created from

ID	configurations
c_1	{GraphLibrary, Edges, Directed}
c_2	{GraphLibrary, Edges, Directed, Algorithms, Number, Cycle}
c_3	{GraphLibrary, Edges, Undirected}
c_4	{GraphLibrary, Edges, Undirected, Algorithms, Number}
c_5	{GraphLibrary, Edges, Directed, Algorithms, Cycle}

Table 2.1: Configurations of *GraphLibrary* product line created using the pairwise sampling algorithm ICPL

feature model *GraphLibrary* using the pairwise sampling algorithm ICPL [Johansen et al., 2012a].

Creating these configurations are instances of the *covering array* problem [Johansen et al., 2011]. In particular, the configurations can be represented as a covering array. The main challenge of generating covering arrays is to find the minimal number of configurations that covers the T-wise combinations of features, which is an NP-hard problem [Engebretsen, 2005]. In addition, finding a valid configuration in a feature model is an NP-complete Satisfiability Problem (SAT) [Cook, 1971]. Numerous algorithms have been proposed to approximate these minimal covering arrays [Chvatal, 1979; Johansen et al., 2011, 2012a; Garvin et al., 2011]. In the following, we give an overview of a set of existing sampling algorithms, which have been used to sample configurations using the feature models as an input.

2.2.2 Covering Array Algorithms

In this thesis, we consider for evaluation purposes the sampling algorithms CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979; Johansen et al., 2011], ICPL [Johansen et al., 2012a], IPOG [Lei et al., 2007], and MoSo-PoLiTe [Oster et al., 2010], as they are well-known sampling algorithms in the product-line testing community [Henard et al., 2014b; Medeiros et al., 2016].

CASA [Garvin et al., 2011] uses simulated annealing to generate T-wise covering arrays of product lines. It is a non-deterministic algorithm where different configurations may be created in different orders when applied multiple times to the same feature model. CASA separates the problem of generating the T-wise covering arrays into two iterated steps. The first step minimizes the number of created configurations. The second step ensures that a certain degree of coverage is achieved.

Chvatal [Johansen et al., 2011] is a heuristic algorithm proposed by Chvatal [Chvatal, 1979] to approximate optimal solutions for the minimal covering array. The basic version of the algorithm does not incorporate feature dependencies. Johansen et al.

[2011] adapted and improved the algorithm to create samples from feature models. The steps of generating covering arrays with the Chvatal algorithm are as follows. First, all T-wise feature combinations are generated. Second, an empty configuration is created. Third, all feature combinations are iterated to add them to the configuration. Each time a new combination is added to the configuration, the validity of this configuration, with respect to the feature model, is checked using a SAT solver [Mendonça et al., 2009b]. If the configuration is invalid, the combination is removed from the configuration. The newly created configuration is added to the final set of configurations if it at least contains one uncovered combination. The creation of configurations continues until all valid T-wise feature combinations are covered at least once.

ICPL [Johansen et al., 2012a] is based on the Chvatal algorithm with several improvements, such as identifying invalid feature combinations at an early stage. It generates the T-wise covering array more efficiently, because the parallelization of the algorithm shortens the computation time significantly. ICPL aims to cover the T-wise combinations of features as fast as possible by covering the maximum number of uncovered feature combinations, each time a configuration is created.

IPOG [Lei et al., 2007] is an algorithm proposed to create covering arrays. These arrays are generated from scratch for the first T features, and then the arrays grow horizontally and vertically. With the horizontal growth, a feature and its value are added, while in the vertical growth, new combinations of the newly added feature and the old ones are added to achieve the coverage, if needed. In the following, we illustrate IPOG's functionality by means of our running example. First, the covering array starts with one feature, e.g., *GraphLibrary*, and its values (*True* and *False*). Second, the covering array grows horizontally by adding a new feature, e.g., *Algorithms*, and its values (*True* and *False*). In particular, we have the following combinations of the two features, $True \wedge True$ and $False \wedge False$. Third, to cover all possible combinations of the two features, the covering array grows vertically by adding the following combinations, $True \wedge False$ and $False \wedge True$. The horizontal and vertical growth continues until all features and their combinations are covered.

MoSo-PoLiTe [Oster et al., 2010; Oster, 2012] is a test framework that combines pairwise testing, to generate a minimal set of products covering all pairwise combinations, and model-based testing, to derive test cases for the generated products. In our work, we focus on the pairwise testing, where products are generated. These products will be used in our evaluation. The pairwise algorithm in MoSo-PoLiTe is proposed based on the existing algorithms IPO [Lei and Tai, 1998] and AETG [Cohen et al., 1994]. MoSo-PoLiTe works as follows: the valid combinations of pair features, w.r.t. the feature model, are generated. Then, the algorithm starts to combine these feature combinations to create valid configurations by starting with the first combination and

iteratively adding the remaining combinations. For each step, forward checking [Haralick and Elliott, 1980] is applied to check whether the selected combination can be added to the remaining combinations in the current configuration. If the current configuration is not valid, the selected combination is removed and another combination is selected to be added instead. The algorithm continues until all pairwise feature combinations are covered by at least one configuration.

Although there are promising approaches for generating covering arrays, most of them do not scale well to large feature models [Medeiros et al., 2016; Liebig et al., 2013] and their execution takes a considerable amount of time. As a result, the Linux kernel developers use the built-in facility of the Linux kernel build system `randconfig` to generate random configurations, because none of the existing sampling algorithms scale to the feature model of the Linux kernel with over 15 thousand features [Melo et al., 2016]. Furthermore, testers cannot start testing until the entire sampling process has terminated, because no intermediate results are reported. In this work, we propose the IncLing algorithm to incrementally sample products one by one based on a greedy selection heuristics to achieve pairwise coverage (cf. Chapter 5). As the testing time in practice is limited, the order in which products are tested matters to find faults as soon as possible. Each sampling algorithm typically outputs an ordered list of configurations. The orders of the aforementioned sampling algorithms may already be effective as they all aim to cover as many interactions as fast as possible. In this thesis, we investigate whether the orders of these algorithms are effective.

2.2.3 Test-Case Prioritization

In a single system, it may require a large amount of time and effort to run all test cases in an existing test suite, especially for large-scale real-world software. Rothermel et al. [2001] report that running test cases of their 20 KLOC industrial software takes approximately seven weeks. To reduce the number of executed test cases, different techniques have been developed, such as test set minimization, test case selection, and test case prioritization [Yoo and Harman, 2012]. While minimization and selection test cases aim to reduce the test sets, test case prioritization aims to reorder test cases based on their priority while keeping the same size. Rothermel et al. [2001] define test case prioritization problem as follows:

Definition 2.3. *(Test-Case Prioritization Problem).*
***Given**: A set of test cases T, the set of its permutations PT_T and function f: $PT_T \rightarrow \mathbb{R}$ computing a test case priority.*
***Problem**: Find $T' \in PT_T$ such that $(\forall T'')$ $(T'' \in PT_T)$, $(T'' \neq T')$, $[f(T') \geq f(T'')]$.*

According to defined criteria, such as the code coverage, test cases with the highest priority should be executed earlier than those with lower priority. Several goals of prioritization can be achieved, such as reducing the costs of testing or increasing the rate of fault detection of test cases (i.e., finding faults as soon as possible). To meet the goals of prioritization, various criteria have been proposed, such as the coverage of the

test cases that cover a large part of the code and the estimated ability of test cases to reveal faults [Rothermel et al., 2001; Henard et al., 2016; Hemmati et al., 2013]. Another technique is time-aware prioritization, which uses a genetic algorithm [Walcott et al., 2006] to prioritize test cases to be executed within a given time budget. Regarding the interaction coverage, Bryce and Memon [2007] propose to prioritize test cases by examining all T-wise interactions of a software system. We refer to the latter in our thesis as *interaction-based* approach.

In our thesis, we follow the same concept of prioritization (cf. definition 2.3), but to prioritize products of a product line under test. In product-lines, prioritization has been exploited to prioritize products under test [Devroey et al., 2014; Johansen et al., 2012b; Henard et al., 2013a; Sánchez et al., 2015; Ensan et al., 2011; Henard et al., 2013c] as well as the generated test cases [Lachmann et al., 2016, 2015; Baller et al., 2014]. While we consider prioritizing products in this thesis, considering test case prioritization for these products may enhance product-line testing effectiveness. In product lines, interaction-based is a search approach that systematically enumerates all possible configurations and selects the configuration which covers the highest number of feature interactions to be tested next. In particular, the interaction-based approach works as follows. We derive all feature combinations from the given configurations. Then, we incrementally select one configuration at a time covering the largest number of uncovered feature combinations. In case we have two or more configurations covering the same number of uncovered combinations, we randomly select the next configuration to be tested next.

The interaction-based approach does not scale to complex product lines due to the expensive computation of the T-wise interactions as well as the exponential growth of the configuration number which must be enumerated each time a configuration is selected [Henard et al., 2014b; Yoo and Harman, 2012]. In this chapter, in addition to random orders and the default orders of sampling algorithms, we used the interaction-based approach to measure the effectiveness of our proposed approaches, where we prioritize configurations based on the dissimilarity among them, as we show in the next chapter.

3. Configuration-Based Similarity-Driven Product Prioritization

This chapter shares material with SoSyM'16 article "Effective Product-Line Testing Using Similarity-Based Product Prioritization" [Al-Hajjaji et al., 2016d], SPLC'14 paper "Similarity-Based Prioritization in Software Product-Line Testing" [Al-Hajjaji et al., 2014], and the AST'17 paper "Efficient Product-line Testing Using Cluster-based Product Prioritization" [Al-Hajjaji et al., 2017b]. Furthermore, we have given tool demo support for product-lines testing at GPCE'16 [Al-Hajjaji et al., 2016b].

As indicated in the previous chapter, the number of possible products can reach up-to 2^n, where n is the number of features. Hence, testing each individual product of a software product line can be unfeasible due to the usual limited testing time. Thus, several approaches have been proposed to restrict the number of products under test [Shi et al., 2012; Carmo Machado et al., 2014; Devroey et al., 2014; Perrouin et al., 2010]. For instance, combinatorial interaction testing, especially pairwise testing, is a promising approach widely used in single system testing as well as in product-line testing to reduce testing effort [Nie and Leung, 2011; Cohen et al., 2007; Lopez-Herrejon et al., 2013; Johansen et al., 2012a; Garvin et al., 2011; Oster et al., 2010]. While combinatorial interaction testing considerably reduces the number of products to consider for testing, this number can still be large to fit in the allocated testing budget. For example, applying pairwise testing for a version of the Linux kernel with $6,888$ features results 480 products [Johansen et al., 2012a], which is a large number to test.

As the time budget for testing in practice is limited, the order in which products are tested matters to presumably find faults as soon as possible within the available amount of time. Hence, different criteria are used to prioritize products, such as statistical analysis of usage models [Devroey et al., 2014], cost and profit of selecting a set of test cases [Baller et al., 2014], domain knowledge [Johansen et al., 2012b; Henard et al.,

2013a; Sánchez et al., 2015], and similarity among products [Henard et al., 2014b; Lity et al., 2017; Devroey et al., 2016]. The notion of similarity heuristic is used in model-based testing of single systems to prioritize test cases [Hemmati and Briand, 2010; Chen et al., 2010; Cartaxo et al., 2011]. Hemmati et al. [2011] report that dissimilar test cases are more likely to detect more different faults than the similar ones.

In this chapter, we propose configuration-based prioritization to prioritize products based on the similarity between them in order to investigate whether it increases the effectiveness of product-line testing. For this purpose, we consider problem-space information in terms of feature selection for product-by-product testing. With regard to the previously tested products, we incrementally select the least similar product in terms of features to be tested next. Identifying the feature selections that distinguish products and prioritize them based on the degree of similarity can yield a meaningful order for testing [Henard et al., 2014b]. Configuration-based prioritization is a heuristic approach that focuses on prioritizing products. Furthermore, configuration-based prioritization can be combined with any sampling technique and can even be applied to the set of productively-used configurations (i.e., configurations defined by users).

In case the diversity between products in a product line is large, testing efforts may increase due to changing environmental testing settings, especially in the automotive domain [Lity et al., 2016]. Thus, we propose cluster-based prioritization to cluster products based on their similarity. On these clusters, developers can apply different testing strategies: First, they may want to cover a specific cluster in more detail, for instance, because it contains commonly demanded products. Hence, test efforts can be reduced by enabling faster fault detection. Second, clustering products can be useful to optimize the setup time for testing. For instance, instead of consuming time to change the testing infrastructure (e.g., changing a specific hardware) for each product, products can be clustered based on a specific parameter and the same testing setup can be used for the whole products in a cluster. Finally, the developer may select a subset of products from each cluster. Hence, the sample covers dissimilar products, and thus, most likely different feature interactions.

The remainder of this chapter is organized as follows. In Section 3.1, we introduce configuration-based prioritization and explain it by means of our running example (cf. Figure 2.2 on Page 9, Table 2.1 on Page 15, and Table 3.1 on Page 24). In Section 3.2, we present the implementation of the proposed approach and the integration of sampling algorithms in FeatureIDE. In Section 3.3, we present our evaluation introducing the experimental setting and used subject product lines. Then, we compare the proposed approach to random orders, interaction-based prioritization, and the default order of sampling algorithms and discuss the reported results. We introduce the cluster-based product prioritization approach in Section 3.4. We discuss the threats to validity of our evaluations that may affect our results in Section 3.5. In Section 3.6, we present related work to our product prioritization approaches.

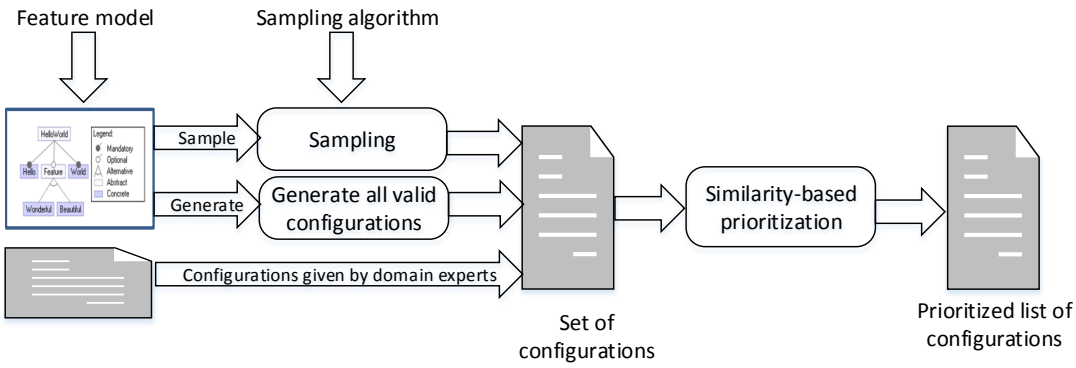


Figure 3.1: Overview of configuration-based prioritization approach

3.1 Configuration-Based Prioritization

With configuration-based prioritization, we prioritize products based on the similarity between them in terms of feature selections. As illustrated in Figure 3.1, the input of configuration-based prioritization is a set of configurations. These configurations can be generated as follows. For small product lines, all valid configurations can be used as input to prioritization. For larger product lines, these configurations can be reduced using sampling algorithms, since it may not be possible to generate all valid configurations. Another option is that these configurations can be given by domain experts as they have insight into the product line, such as which configurations are mostly desired by customers. The outcome of our approach is a prioritized list of configurations.

In configuration-based prioritization, we select one product at a time to be tested. We select the product that has the lowest value of similarity compared to *all* previously tested products in terms of feature selections. With configuration-based prioritization, we consider the *selected and deselected* features when we calculate the distance (cf. Equation 3.1). The rationale of taking the deselected features into account is based on the observation that some faults can be triggered because other features are *not* selected [Abal et al., 2014; Medeiros et al., 2016]. In the following, we present the main algorithm of our approach in Algorithm 3.1 and explain its steps with our running example.

3.1.1 Initialization

The inputs of our algorithm are a set of valid products $P_{SPL} \in SPL$ and the corresponding feature model $\mathcal{FM}$. The output of the algorithm is a list of prioritized products in terms of feature selections (i.e., their configurations). At the beginning, we initialize the required variables F, which represents all features in a feature model $\mathcal{FM}$ and $AllDistance$, which represents the degree of similarity between products (Lines 2–10).

Algorithm 3.1 Configuration-Based Prioritization.

Require: $\mathcal{FM}$ ▷ feature model
P_{SPL} ▷ a set of products
Return: P_{tested} ▷ list of tested products

```
1:  function PRIORITIZATION(P_SPL, FM)
2:      F ← GETFEATURES(FM)
3:      P_tested ← EMPTYLIST
4:      for each p_i ∈ P_SPL do                    ▷ Calculate distances between products
5:          for each p_j ∈ P_SPL do
6:              if p_i ≠ p_j then
7:                  AllDistance[i, j] ← GETDISTANCES(p_i,p_j,F)
8:              end if
9:          end for
10:     end for
11:     allyesconfig ←GETALLYESCONFIG( FM, P_SPL)          ▷ Algorithm 3.2
12:     P_tested ← P_tested ∪ {allyesconfig}
13:     P_SPL ← P_SPL \ {allyesconfig}
14:     TESTPRODUCT(allyesconfig)
15:     while |P_SPL|>0 do
16:         p_i = SELECTCONFIGURATION(P_SPL, P_tested, AllDistances))   ▷ Algorithm 3.3
17:         P_tested ← P_tested ∪ {p_i}
18:         P_SPL ← P_SPL \ {p_i}
19:         TESTPRODUCT(p_i)
20:     end while
21:     return P_tested
22: end function
```

We measure the similarity between products by calculating the distances between them (Lines 4–10). We use an adaptation of the Hamming distance [Hamming, 1950] to measure the similarity among products. Devroey et al. [2016] investigate different types of distance measurements in product-line testing and they report that Hamming and Jaccard distances are most effective distance functions. While we use Hamming distance in this chapter, we conduct a comparison between the Hamming and Jaccard distances, with respect to the testing effectiveness, in the next chapter. Using Hamming distance, we define the distance between two given products p_i and p_j relative to the set of all features F as

$$distance(p_i, p_j, F) = 1 - \frac{|p_i \cap p_j| + |(F \backslash p_i) \cap (F \backslash p_j)|}{|F|} \tag{3.1}$$

where $|p_i \cap p_j|$ is the number of features selected in products p_i and p_j, $|(F \backslash p_i) \cap (F \backslash p_j)|$ is the number of features deselected in products p_i and p_j, and F is the set of all features. The definition of $distance(p_i, p_j, F)$ is symmetric with regard to p_i and p_j, which can be used to reduce the number of computational steps of our algorithms. The

distance values among products are normalized, and therefore, they range from 0 to 1. Values close to 0 indicate similar feature selections, while values close to 1 indicate that the products differ in almost every feature.

Example 3.1. *In this example, we show how the distances between products are calculated. We calculate the distance between p_1 and p_2 (cf. Table 2.1 on Page 15), where each configuration represents the selected features of the corresponding product). With p_1, we have three selected features and four deselected features, while with p_2 we have six selected features and one deselected feature. The value of $|p_i \cap p_j|$ is 3, while the value of $|(F \backslash p_i) \cap (F \backslash p_j)|$ is 1. Hence,*

$$\begin{aligned} distance(p_1, p_2, F) &= 1 - \frac{3+1}{7} \\ &= 0.429 \end{aligned}$$

We follow the same step to calculate the distances between all products. The resulting distances between products are reported in Table 3.1 on Page 24.

After calculating and saving the distances between products, which is the last step in the initialization phase, we start to select products to be tested.

3.1.2 First Product Selection

Since considering the distance does not help to select the first product to test, we select the product that has the maximum number of selected features to be tested first (cf. Algorithm 3.2). We iterate all products in order to find the product with the maximum number of selected features. The rationale for selecting the product that has the maximum number of selected features as the first to be tested is the following:

- Most faults that may exist in an individual feature can be detected with this product [Bessey et al., 2010].
- Most faults that are caused by the interaction of selected features can be detected with this product [Abal et al., 2014; Medeiros et al., 2016].

For these reasons, it is a common strategy in the Linux community to test this product first (a.k.a. *allyesconfig*) [Dietrich et al., 2012]. If more than one product has the same value of the maximum number of selected features, we select the first iterated product that has this value (cf. Algorithm 3.2 Line 6). The selected product is added to the list P_{tested} (Algorithm 3.1 Line 17), which contains the prioritized products, and removed from set P_{SPL} (Algorithm 3.1 Line 18), which contains the remaining products.

Example 3.2. *From Table 2.1 on Page 15, we have product p_2 that has the maximum number of selected features (six features). We select product p_2 to be the first product (allyesconfig) to be tested (cf. Algorithm 3.2). In particular, product p_2 is added to list P_{tested} and removed from set P_{SPL}.*

	p_1	p_2	p_3	p_4	p_5
p_1	0	0.429	**0.286**	0.571	0.286
p_2	0.429	0	**0.714**	0.429	**0.143**
p_3	**0.286**	**0.7140**	**0.286**	0.571	
p_4	0.571	0.429	**0.286**	0	0.571
p_5	0.286	**0.143**	0.571	0.571	0

: The max distance to the first tested product (p_2)
: The max distance over min to the tested products

Table 3.1: Distances between the five configurations listed in Table 2.1

Algorithm 3.2 Select First Product To Test.

Require: $\mathcal{FM}$ ▷ feature model
P_{SPL} ▷ a set of products
Return: $allyesconfig$ ▷ product with the maximum number of selected features

```
1: function GETALLYESCONFIG(P_SPL, FM)
2:     allyesconfig ← p_1
3:     F_allyesconfig ← GETSELECTEDFEATURES(allyesconfig)
4:     for each p_i ∈ P_SPL do
5:         F_Selected ← GETSELECTEDFEATURES(p_i)
6:         if |F_Selected| > |F_allyesconfig| then
7:             allyesconfig ← p_i
8:         end if
9:     end for
10:    return allyesconfig
11: end function
```

3.1.3 Incremental Product Selection

After selecting the first product (*allyesconfig*), we select a new product to be tested that is least similar to the first one (Algorithm 3.1 Line 16). In case the new product is the second one, we only need to compare the distances of products to the first selected one (*allyesconfig*) (the inner loop of Algorithm 3.3 Lines 5–11). Considering the distances, which are already calculated in the initialization phase (cf. Table 3.1), we are able to select the second product to be tested.

Example 3.3. *In list S, we already have product p_2. The distances between product p_2 and the other products is already calculated (cf. the second row or column of Table 3.1). The products with the maximum distances (0.714) to product p_2 is product p_3 (highlighted with circle and additional background color). The product p_3 is added to list P_{tested} and removed from set P_{SPL}. As a result, two products exist in list $P_{tested} = (p_2, p_3)$ and all other untested products remain in set P_{SPL} (i.e., $P_{SPL} = \{p_1, p_4, p_5\}$).*

After selecting the second product, we select the third product that is least similar to *all* previously tested products. By incorporating distances between *all* previously tested products, the selection of the next product ensures fast coverage of feature interactions [Hemmati et al., 2013]. In the literature, a strategy has been used to calculate distances between more than two products in product-line testing, namely *maximum over distance summation* [Henard et al., 2014b].

Based on the sum of the distances between each product in the untested products P_{SPL} and all tested products in P_{tested}, we select the next product to be tested that has the maximal sum distances. Therefore, we define the corresponding function $next_{sum} : \mathcal{P}(P_{SPL}) \times \mathcal{P}(P_{SPL}) \rightarrow \mathcal{P}(P_{SPL})$ for selecting the next product as follows:

$$next_{sum}(P_{SPL}, P_{tested}) = p \in P_{SPL} \setminus P_{tested}\, with \forall p' \in P_{SPL} \setminus P_{tested} : \sum_{p_j \in P_{tested}} distance(F_{p'}, F_{p_j}, F_{SPL}) \geq \sum_{p_i \in P_{tested}} distance(F_{p'}, F_{p_i}, F_{SPL}) \tag{3.2}$$

However, the summation may be prone to favor outliers. For example, products which have solely one large distance to one of the already tested products are selected for testing instead of products which have average distances to the tested ones with a smaller summation. Hence, we consider a strategy to calculate distances between more than two products in product-line testing, namely *maximum over distance minimum* [Kuby, 1987]. In the strategy maximum over distance minimum, we perform two steps:

- First, we determine the minimum distances, which we already have from the initialization phase, for all untested products to the already tested products (Algorithm 3.3 Lines 5–11).
- Second, we determine the maximum of these minimum distances and select the corresponding product as next product to be tested (Algorithm 3.3 Lines 12–15).

By incorporating the maximum over distance minimum, we are able to determine the best increment in feature coverage and further are more stable against outliers. To determine minimal distances, we define the relation $minDist \subseteq P_{SPL} \times P_{SPL}$ capturing the pair of untested and tested products for which the minimal distance exists such that for all $p \in P_{SPL} \setminus P_{tested}$ a corresponding pair exists as follows:

$$(p, p') \in minDist \Leftrightarrow \exists p' \in P_{tested} \ : \forall p'' \in P_{tested} \setminus \{p'\} : distance(F_p, F_{p'}, F_{SPL}) \leq distance(F_p, F_{p''}, F_{SPL}) \tag{3.3}$$

We use this relation for the product selection function $next_{min} : \mathcal{P}(P_{SPL}) \times \mathcal{P}(P_{SPL}) \rightarrow \mathcal{P}(P_{SPL})$ which is defined as follows:

Algorithm 3.3 Select the next Configuration.

Require: P_{SPL} ▷ a set of products
P_{tested} ▷ a list of tested products
$AllDistances[\,]$ ▷ The calculated distances between products
Return: $NextConfig$ ▷ product with the largest distance to the tested ones

```
1: function SelectConfigurations(P_SPL, P_tested, AllDistances)
2:     NextConfigDistance ← 0
3:     for each p_i ∈ P_SPL do
4:         TempDistance ← 1
5:         for each p_j ∈ P_tested do
6:             Distance_{p_i p_j} ← GetDist(p_i, p_j)
7:             if Distance_{p_i p_j} < TempDistance then
8:                 TempDistance ← Distance_{p_i p_j}
9:                 P_{New_Temp} ← p_i
10:            end if
11:        end for
12:        if TempDistance > NextConfigDistance then
13:            NextConfigDistance ← TempDistance
14:            NextConfig ← P_{New_Temp}
15:        end if
16:    end for
17:    return NextConfig
18: end function
```

$$next_{min}(P_{SPL}, P_{tested}) = \{p \in P_{SPL} \setminus P_{tested} |\ \forall p' \in P_{SPL} \setminus P_{tested} : \min_{p_i \in P_{tested}} distance(F_{p'}, F_{p_i}, F_{SPL}) > \min_{p_j \in P_{tested}} distance(F_{p'}, F_{p_j}, F_{SPL})\} \tag{3.4}$$

The result may comprise more than one potential candidate for the next product to be tested. In such a case, we, again, select the first iterated candidate.

Example 3.4. *Consider again our running example, we already have two tested products in the list P_{tested} (p_2 and p_3). To select the third product, we determine first the minimum distances for all untested products (p_1, p_4, and p_5) to all tested products (p_2 and p_3) (Algorithm 3.3 Lines 5–11). These minimum distances are* 0.286, 0.286, *and* 0.143 *(highlighted with bold font in Table 3.1 on Page 24). Second, we select the product with maximum of these minimum distances (Algorithm 3.3 Lines 12–15). In our case, we have two products, p_1 and p_4 with* 0.286. *In this case, we select the first iterated one of these products, which is product p_1 (highlighted with circle and bold font in Table 3.1). We repeat the previous step until all products are selected to be tested. The resulting testing order is p_2, p_3, p_1, p_4, and p_5.*

Performing configuration-based prioritization is computationally cheap, because we essentially rely on the Hamming distance, which is a linear operation in the number of

features, to calculate distances among products. We implemented our approach and combined it with existing sampling algorithms in FeatureIDE [Al-Hajjaji et al., 2016c] as will be described in the next section.

3.2 Configuration-Based Prioritization in FeatureIDE

FeatureIDE [Meinicke et al., 2017; Thüm et al., 2014b] is a set of Eclipse plug-ins that support all phases of product-line development from domain analysis to software generation. FeatureIDE covers the entire product-line development process and integrates with tools for the implementation of product lines. In this section, we focus on the functionalities of FeatureIDE that is related to product-by-product testing.

Product-by-product testing is a technique that generates and tests individual products using an existing testing technique from single systems engineering. For product-by-product testing and analyses, it is useful to automatically derive configurations from the feature model. To automatically derive configurations as well as to generate and test products in FeatureIDE, we support several strategies to provide configurations for testing, namely using user-defined configurations, deriving all valid configurations, and using T-wise sampling [Al-Hajjaji et al., 2016c; Meinicke et al., 2017]. *User-defined* configurations can be created manually using the integrated configuration editor [Pereira et al., 2016]. This is a straightforward strategy, which is commonly used in practice, especially for the small product lines. *All valid* configurations can be generated using an algorithm that exploits the tree structure of the feature model. Generating all valid configurations scales only for small product lines. *T-wise sampling* aims to generate a minimal set of configurations that covers all interactions among T features. In the following, we present the integrated sampling algorithms that have been proposed to sample product lines. These algorithms are also used in our evaluation.

T-Wise Sampling Algorithms

In FeatureIDE, we integrate three T-wise sampling algorithms, namely CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979; Johansen et al., 2011], and ICPL [Johansen et al., 2012a].

- CASA [Garvin et al., 2011] uses simulated annealing to derive configurations. Therefore, it is a non-deterministic algorithm where a different number of configurations may be created for the same product line in different runs.
- Chvatal [Chvatal, 1979; Johansen et al., 2011] is a heuristic algorithm proposed to create configurations for software product lines.
- ICPL [Johansen et al., 2012a] is based on the Chvatal algorithm with several improvements, such as identifying invalid feature combinations at an early stage. It generates the T-wise covering array efficiently as it makes use of multi-threading.

Each sampling algorithm produces some implicit order as part of its output that is given by the positioning of the products within the output data structure. However, this order is not explicitly mentioned by the authors who proposed these sampling algorithms and is somehow influenced by coverage criteria. Moreover, the order produced by some sampling algorithms is even non-deterministic (i.e., different runs with the same input often lead to different orders).

In FeatureIDE, we enable users to choose the way to order configurations, either by the default order of these sampling algorithms or by configurations-based prioritization, which is implemented and integrated in FeatureIDE.

In addition, we implemented the *interaction-based* approach, a greedy approach used in our evaluation as a baseline [Bryce and Memon, 2007]. The interaction-based approach aims to optimize feature interaction coverage. For this, the configuration that covers most feature interactions that are not already covered by previous configurations is selected. This process is continued until all configurations are tested, or all interactions are covered (i.e., the number of configurations to test may be smaller). To evaluate our approach, we used the aforementioned implementation to conduct our experiments.

3.3 Evaluation of Configuration-Based Prioritization

For a given set of products, configuration-based prioritization aims at detecting faults earlier (with respect to random and sampling orders) by faster increasing interaction coverage. We measured the potential improvements of effectiveness in terms of the fault detection rate compared to random orders, to the default order of sampling algorithms, and to the interaction-based approach [Bryce and Memon, 2007]. Especially the interaction-based approach is commonly used to measure the effectiveness of combinatorial interaction testing approaches [Henard et al., 2014b]. The rationale of the comparison to the interaction-based approach is that our approach aims to increase the interaction coverage for a product line under test over time. In our experiments, we consider the interaction of only up-to $T = 2$ in the interaction-based approach, as it does not scale well to large t [Henard et al., 2014b; Yoo and Harman, 2012].

In particular, the run-time complexity of the interaction-based approach is $\mathcal{O}(m^2 n^t 2^t)$, where m is the number of products, n is the number of features, and t is the degree of the interaction coverage. However, the run-time complexity of the configuration-based prioritization approach is $\mathcal{O}(m^2 n)$. Regarding the total required memory space, the interaction-based approach requires $mn + n^t 2^t$, which represents the memory of storing the feature selections of products (mn) and the combination of features ($n^t 2^t$). However, the required memory space of configuration-based prioritization is $mn + m^2$, which represents the memory of storing the feature selections of products (mn) and the distances between products (m^2). That is, the interaction-based approach requires more memory space as well as time than configuration-based prioritization. Note that a space–time trade-off can be made to optimize the needed space and time for both approaches.

In our evaluation, we aim at answering the following research questions:

RQ1 Can configuration-based prioritization detect faults faster compared to random orders and the interaction-based approach?

RQ2 Do sampling algorithms produce samples with an acceptable effective order?

RQ3 Is the computational overhead required for configuration-based prioritization negligible compared to the efforts required for sampling?

Regarding RQ1, we ran experiments using three available subject product lines with real faults in the feature source code. These subjects have already been used in previous studies on product-line verification [Apel et al., 2013c; Meinicke et al., 2016b]. In Table 3.3, Page 32, we summarize statistics on these product lines. As these product lines are rather small in terms of the number of features, we ran further experiments to evaluate our approach using real-world feature models and artificial ones of various sizes. With respect to the experiment of product lines with real faults and to the experiment of feature models, we derived the following, more specific research questions from *RQ*1:

RQ1.1 Can configuration-based prioritization detect faults faster than random orders and the interaction-based approach *for product lines with real faults*?

RQ1.2 Can configuration-based prioritization cover *feature interactions* faster than random orders and the interaction-based approach?

In the remainder of this section, we introduce metrics that are used to evaluate our results in Section 3.3.1. In Section 3.3.2, we describe the experiment to address *RQ*1.1, where we used three subject product lines with real faults in the source code. In Section 3.3.3, we present the settings and the results of the experiment with feature models to address *RQ*1.2, *RQ*2, and *RQ*3. Finally, we discuss the threats to validity in Section 3.5.

3.3.1 Evaluation Metrics

In this section, we introduce APFD metric to evaluate our approaches and the Mann-Whitney U test to analyze the reported results.

Average Percentage of Faults Detected (APFD)

We use a well-known metric called APFD developed by Elbaum et al. [2000] to evaluate the pace of fault detection. This metric is widely used in the literature to evaluate the prioritization testing approaches [Kuhn et al., 2013; Rothermel et al., 2001; Elbaum et al., 2002; Li et al., 2007; Yoo and Harman, 2012; Qu et al., 2007; Henard et al., 2016; Sánchez et al., 2014; Walcott et al., 2006]. The APFD is calculated by measuring the average number of faults detected in the system under test. APFD values range

Faults	Products				
	p_1	p_2	p_3	p_4	p_5
f_1				x	x
f_2	x	x		x	
f_3		x		x	
f_4			x		x
f_5			x		
f_6			x	x	x

Table 3.2: Fault matrix

from 0 to 1; higher values of APFD indicate faster fault detection rates. APFD can be calculated as

$$\text{APFD} = 1 - \frac{tf_1 + tf_2 + ... + tf_m}{n * m} + \frac{1}{2n} \tag{3.5}$$

where n is the number of test cases, which represent products in our case, m is the number of faults, and tf_i is the position of the first test t that exposes the fault. We show how to calculate APFD using our running example. We have five products $P = \{p_1, \ldots, p_5\}$, listed in Table 2.1 on Page 15, and we assume that six faults $f_1, \ldots, f_6$, are distributed as shown in Table 3.2. Moreover, assume we have two orderings of these products, ordering $O_1 : p_1,\ p_2,\ p_3,\ p_4,\ p_5$ and ordering $O_2 : p_3,\ p_1,\ p_4,\ p_2,\ p_5$. Incorporating the data from Table 3.2, the APFD calculation yields

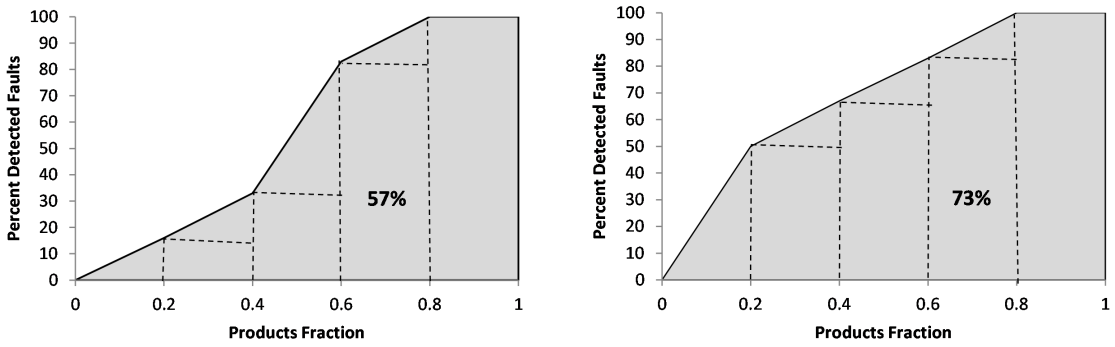
$$\text{APFD} = 1 - \frac{4 + 1 + 2 + 3 + 3 + 3}{6 * 5} + \frac{1}{2 * 5} = 0.57$$

for O_1 and

$$\text{APFD} = 1 - \frac{3 + 2 + 3 + 1 + 1 + 1}{6 * 5} + \frac{1}{2 * 5} = 0.73$$

for O_2. Thus, the ordering O_2 yields a better fault detection rate (0.73) than O_1 (0.57).

In the following, we illustrate the effect of prioritization using APFD metric. Using the fault matrix in Table 3.2 and considering the two aforementioned orders O_1 and O_2, we show in Figure 3.2, the percentage of detected faults versus the fraction of the products tested. With respect to order O_1, we detect one of the six faults, which represent 16% of the faults, after testing product p_1, which represents 0.2 of the products (cf. Figure 3.2(a)). After testing the second product p_2, we detect two faults, which represent 33% of the faults. The area inside the rectangle (dashed boxes) in Figure 3.2(a) represents the weighted percentage of faults detected over the corresponding fraction of the



(a) Order 1 (b) Order 2

Figure 3.2: Example illustrating the APFD measure.

products under test. That is, the area under the curve represents the weighted average of the percentage of faults detected over the progress of testing products (i.e., over the life of products under test). In particular, this area represents the average percentage faults detected in the prioritized products under test (APFD), which is in Figure 3.2(a) 57%. Changing the order of products under test to O_2: p_3, p_1, p_4, p_2, p_5, yields to a faster detection, as illustrated in Figure 3.2(b), with an APFD of 73%.

Mann-Whitney U test

In this thesis, we compared our proposed approaches against other existing ones that include some randomness. Therefore, to analyze the effectiveness of these approaches, it is important to study the distribution of their results (e.g., APFD values). Considering only the *average values* can be misleading [Arcuri and Briand, 2011]. Thus, statistical tests [Rice, 2006] can be used to assess whether the results are reliable. These tests show whether there is enough empirical evidence to claim a difference between these approaches.

Based on guidelines for reporting statistical tests [Arcuri and Briand, 2011], we use in our thesis the Mann-Whitney U test to analyze the APFD values. The Mann-Whitney U test is a non-parametric statistical test of the null hypothesis that two samples come from the same population against an alternative hypothesis (i.e., a particular population tends to have larger values than the other). From this test, we obtain the p-value representing the probability that two samples are equal. The significance level is 0.05. That means, if the p-value is less than or equal to 0.05, we reject the null hypothesis that the two samples are equal.

3.3.2 Experiment with Code Base of Existing Product Lines

In this section, we describe our experiment using three product lines in order to address *RQ1.1*: *Can configuration-based prioritization detect faults faster than random orders and the interaction-based approach for product lines with real faults?*.

Product lines	LOC	Features	Specifications	Products	Faults
Elevator	1046	6	9	20	8
Mine-pump	580	7	5	64	4
E-mail	1233	9	9	40	9

Table 3.3: Overview of subject product lines

Subject Product Lines

We selected three product lines that have been previously used to evaluate verification strategies of product lines [Apel et al., 2013c; Meinicke et al., 2016b]. These product lines are the only ones we are aware of that have real faults caused by erroneous feature interactions and that are publicly available:

- The *Elevator* system is an Elevator model designed by Plath and Ryan [2001]. It provides various features, such as stopping if the elevator is empty or high priority service for a special floor.
- The *Mine-pump* system simulates a water pump in mining operations [Kramer et al., 1983]. An example of these operations is keeping the bottom of the mine shaft dry and deactivating the pump in case the mine contains methane gas.
- The *E-mail* system of Hall [Hall, 2005] provides several features, such as encryption, decryption, and automatic forwarding.

The aforementioned product lines are implemented using feature-oriented programming [Prehofer, 1997]. In particular, these product lines are implemented in Java using the tool *FeatureHouse* [Apel et al., 2009]. Each feature of these product lines has individual specifications. The specifications, in form of assertions, are introduced using AspectJ. Apel et al. [2013c] adapted the initial specifications that are written by the original authors of these product lines. The three product lines contain faults that are documented by the authors of these product lines. Apel et al. [2013c] focus on the faults that are caused by the feature interaction. These faults violate at least one specification. Mainly, the specifications concern specific safety properties. For instance, in the Elevator system, if the weight is more than the maximum weight, the elevator should not move. In this system, the detected faults are caused by the interaction of up-to three features. In the E-mail system, an encrypted e-mail must not be transferred in plain text. The reported faults in E-mail system are caused by the interaction of up-to six features. For the Mine-pump system, an example of a safety concern is that in case a methane gas is detected, Mine-pump must be deactivated. The detected faults in this system are caused by the interaction of up-to four features. The information of these product lines is summarized in Table 3.3. In addition to the product lines and

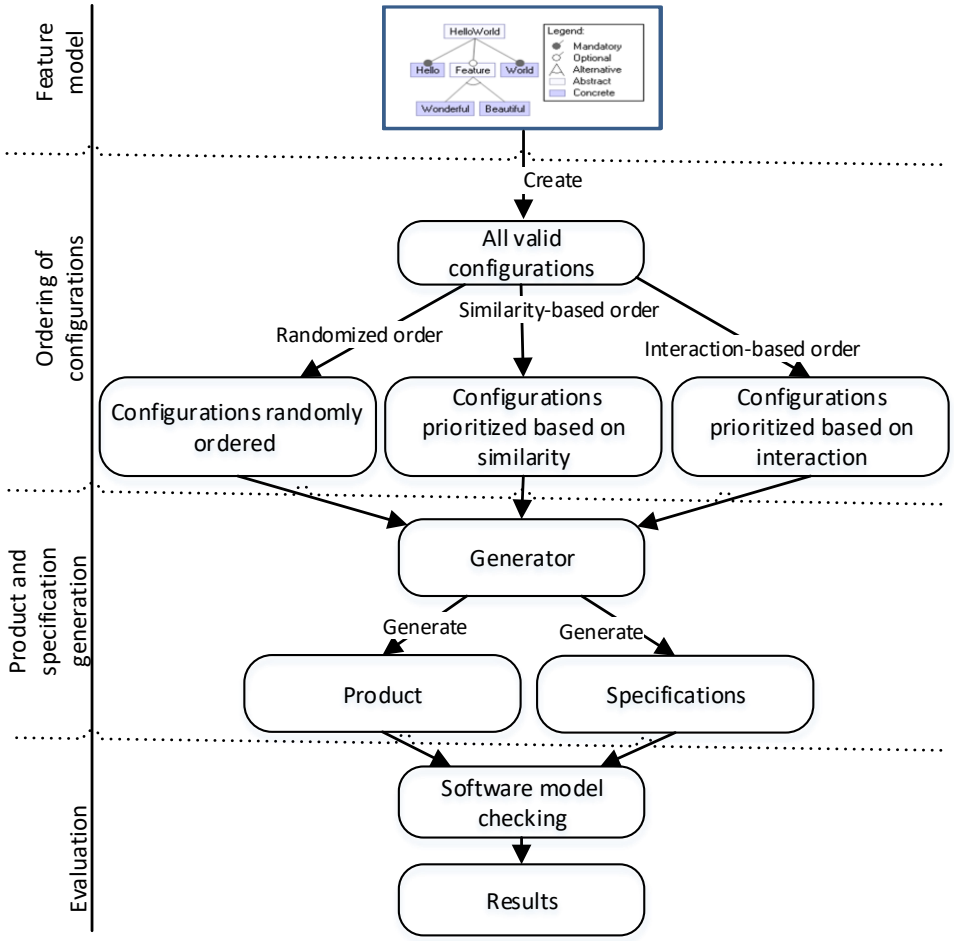


Figure 3.3: Steps of the experiment with code base of product-lines

their feature models, the information about how these faults are caused by the feature interaction are publicly available.[1]

Experiment Design

Figure 3.3 visualizes the experimental steps. First, we use feature models of the subject product lines to generate all valid configurations, as these product lines are small enough. Second, we prioritize these products with configuration-based prioritization and measure the potential improvement of effectiveness compared to random orders as well as to the interaction-based approach [Bryce and Memon, 2007]. We repeat the experiment of random orders 100 times to mitigate the impact of randomness. Third, the products and the specifications for the corresponding configurations are generated using

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/thesis

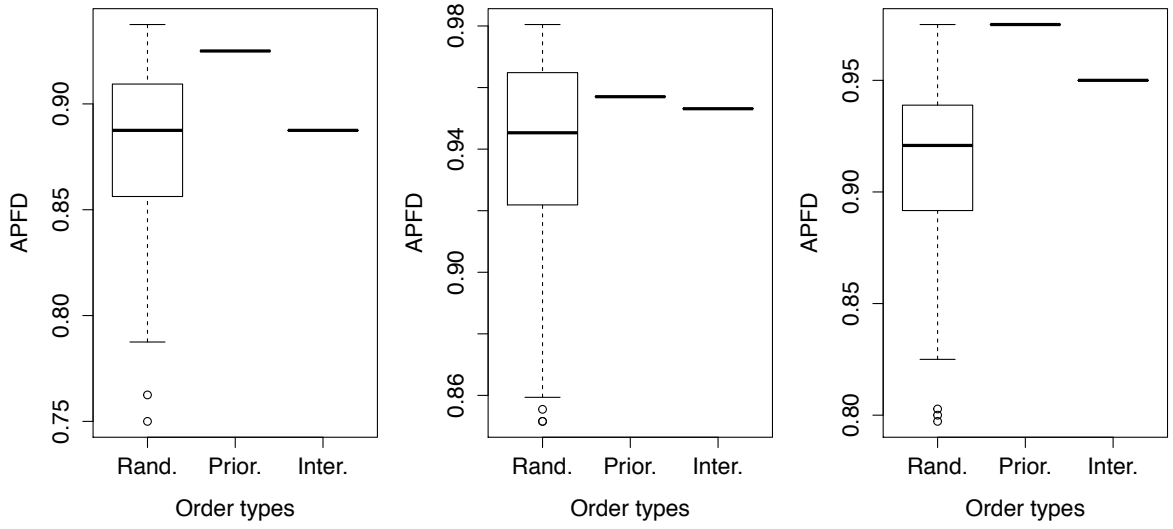
FeatureIDE. Finally, we verify whether the products satisfy the specification using *Java Pathfinder* (JPF) [Visser et al., 2000] as a software model checker. If a product does not satisfy the specification, we consider the violation of specification as a fault. Then, we use the APFD metric (cf. Section 3.3.1) to evaluate the effectiveness of configuration-based prioritization and compare it against interaction-based prioritization and random orders.

Experiment Results

To answer *RQ*1.1, we show in Figure 3.4 the APFD value distribution of random orders, our configuration-based prioritization approach, and interaction-based approach for three product lines. Figure 3.4 shows that the APFD values of our approach are higher than the median APFD values of the distribution of 100 random orders for each product line. However, we observed that in *Mine-pump* (cf. Figure 3.4(b)) some random orders are better than configuration-based prioritization. A possible reason for that is the limited number of detected faults in the product line *Mine-pump* (only four faults) compared to the product lines *Elevator* (eight faults) and *E-mail* (nine faults). In Table 3.4, we show that APFD values for configuration-based prioritization are higher than the average APFD values over 100 random orders for all subject product lines. Using the Mann-Whitney U test, we found that our approach and random orders are significantly better for the three product lines *Elevator* (p-value=0.0), *Mine-pump* (p-value=0.0004), and *E-mail* (p-value=0.0). Although the number of faults and the size of the subject of product lines are small, the reported results show that configuration-based prioritization is better than the random order.

Comparing our approach to the interaction-based approach (cf. Figure 3.4 and Table 3.4), we found that the APFD values of our approach are higher than the values of interaction-based approach for the three product lines. In particular, the APFD values of configuration-based prioritization and the interaction-based approach for the three product lines *Elevator*, *Mine-pump*, and *E-mail* are (0.925, 0.957, and 0.975) and (0.888, 0.953, and 0.950), respectively. The reason for the potential improvement of effectiveness by our approach compared to the interaction-based approach is that most of the faults, which caused by the interaction of selected features, are detected in the first product (*allyesconfig*). In the interaction-based approach, the first product is selected randomly, because each product in the first step covers the same number of feature combinations. Hence, it might be the case that most of the features are not selected. However, the interaction-based approach can be improved by starting with *allyesconfig* as well.

In response to **RQ1.1**, the results show that configuration-based prioritization is significantly better than the random order and slightly better than the interaction-based approach. Although the criteria, which we used to prioritize the products, are based only on selected and deselected features, they give a clue when we applied our approach on product lines with real faults that testing effectiveness of product lines can be improved. To validate our approach further, we conducted other experiments with feature models and simulated test execution and fault detection.



(a) Elevator (b) Mine-pump (c) E-mail

Figure 3.4: APFD value distribution of random orders, our configuration-based prioritization approach, and interaction-based approach for three product lines: Elevator, Mine-pump, and E-mail

Product line	Random	Conf. Prio.	Interaction-based
Elevator	0.881	0.925	0.888
Mine-pump	0.937	0.957	0.953
E-mail	0.912	0.975	0.950

Table 3.4: The average APFD value over 100 random orders, the APFD value of configuration-based prioritization, and the APFD value of the interaction-based approach

3.3.3 Experiments with Feature Models

In this section, we present the feature models, the experiment design, and the results of the experiment to address $RQ1.2$, $RQ2$, and $RQ3$.

Subject Feature Models

We consider a variety of subject systems from academia as well as real-world systems to evaluate our approach. Due to the lack of having open-source software product lines that have test cases and faults in the source code [Sánchez et al., 2014], we conduct our experiment using the feature models of the subject systems. Besides our running example, we select all feature models in the S.P.L.O.T. repository [Mendonça et al., 2009a] that have more than 140 features[2] to evaluate the effectiveness of our approach.

[2] http://splot-research.org/, last accessed February 07, 2017

Feature Model	#Features	#Constraints	CTCR	#Configurations*
BattleofTanks	144	0	0%	459
FM_Test	168	46	28%	44
Printers	172	0	0%	181
BankingSoftware	176	4	2%	42
Electronic Shopping	290	21	11%	22
DMIS	366	192	93%	29
eCos 3.0 i386pc	1,245	2,478	99%	62
FreeBSD kernel 8.0.0	1,369	14,295	93%	77
Automotive1	2,513	2,833	28%	913
Linux 2.6.28.6	6,888	6,847	99%	479
10xAFM15†	15.0	2.6	19%	13.7
10xAFM50†	50.0	9.7	17%	26.4
10xAFM100†	100.0	20.0	17%	56.4
10xAFM200†	200.0	39.0	17%	89.9
10xAFM500†	500.0	100.0	17%	189.6
10xAFM1000†	1,000.0	100.0	14%	343.6
1xAFM5K	5,542	300	11%	685

CTCR: cross-tree constraints representative

*: Numbers of configurations calculated using pairwise sampling algorithm ICPL

†: The values next to the artificial feature models represent the average over 10 feature models

Table 3.5: Feature models used in our evaluation.

Furthermore, we consider feature models of very large-scale product lines, such as the Linux kernel in version 2.6.28.6 with 6,888 features. The feature model of the Linux kernel has been used previously to evaluate the scalability of product-line testing [Johansen et al., 2012a; Henard et al., 2014b]. In addition to the real feature models, we consider 61 artificial feature models generated with S.P.L.O.T. [Mendonça et al., 2009a], which also served previously as a benchmark for evaluation purposes [Henard et al., 2014b]. The feature models that we used in our experiments are shown in Table 3.5, where we present the following data for each feature model: number of features, number of constraints, ratio of the number of distinct features in cross-tree constraints to the number of features (CTCR), and number of valid configurations using the pairwise sampling algorithm ICPL [Johansen et al., 2012a]. For simplification purposes, we show in Table 3.5 the average values over 10 artificial feature models of each model size (i.e., 10 feature models of size 10, 10 feature models of size 50, and etc.). These feature models are highlighted in Table 3.5 with †.

Experiment Design

In Figure 3.5, we show an overview of the experiments with feature models. To evaluate the effectiveness of configuration-based prioritization, we use feature models of

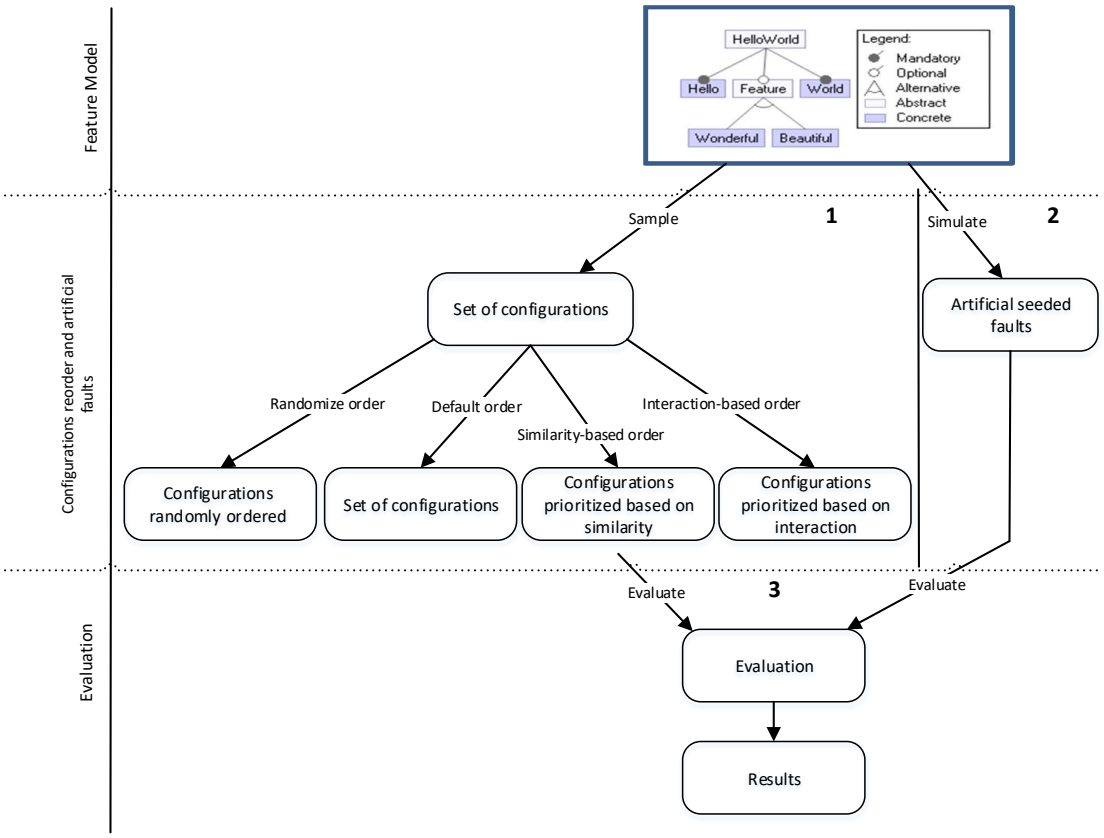


Figure 3.5: Steps of the experiment with feature models

different sizes as inputs for the sampling algorithms. The input to configuration-based prioritization is a set of configurations which are created based on the feature model (cf. Figure 3.5(1)). Generating all valid configurations as in Section 3.3.2 is not feasible for these feature models. Hence, we create these configurations using the algorithms CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979], and ICPL [Johansen et al., 2012a]. These algorithms are well known in the community and they have been used previously to evaluate product-line testing techniques [Henard et al., 2014b; Sánchez et al., 2015, 2014; Henard et al., 2016]. We compare the effectiveness of configuration-based prioritization to the effectiveness of the default order of each sampling algorithm. However, it is not clear whether the default orders of these algorithms are effective or not. The order of the created configurations by the sampling algorithms is mainly influenced by the interaction coverage where these algorithms aim to cover the feature combinations as fast as possible. Thus, we consider the default order of sampling algorithms in our evaluation. In addition, we compare our approach to random orders and interaction-based approach. The outputs of these sampling algorithms are served as inputs for configuration-based prioritization as well as the interaction-based approach, and random orders. Note that the input to the random orders and the interaction-based ap-

proach are configurations created using the sampling algorithm ICPL [Johansen et al., 2012a].

Artificial Faults

For the evaluation purpose, we implemented a fault distribution simulator based on the fault simulators presented by Bagheri et al. [2012] and Ensan et al. [2012] and used by Sánchez et al. [2014] to evaluate the fault detection rate of product-line test suites. Generating these faults is independent of prioritizing products. Applying such a technique is common in the community [Bagheri et al., 2012; Ensan et al., 2012; Sánchez et al., 2014; Devroey et al., 2016], due to the lack of having case studies that have test cases, source code, and numerous real faults.

We assume that faults are equally distributed over features in a product line. The issue with this assumption is that faults are regularly discovered where they are not expected. In any case, we argue that assuming equal distribution is better than to build on non-idealized, yet conceivably non-representative distributions. The input of our fault generator is a feature model and the outputs are valid combinations of features (i.e., partial configurations) (cf. Figure 3.5(2)). We mark those generated combinations as they are faulty. We assume that the faults will be detected if the combination of features causing a fault is covered in a configuration (cf. Figure 3.5(3)). Similar to previous work Sánchez et al. [2014], we simulate $n/10$ faults on each feature model, where n is the number of features. We assume that having a larger feature model leads to potentially more feature interactions. Thus, the number of simulated faults in our experiment is proportional to the feature model size.

We generate two types of faults from the fault simulator, called *exhaustive interaction faults* and *pattern interaction faults*. We use these two types of faults to assess the effectiveness of our approach in terms fault detection ratio with different fault types. Bagheri et al. [2012] simulate faults for 2-wise feature interactions, and Sánchez et al. [2014] simulate faults for 4-wise feature interactions. With *Exhaustive Interaction Faults*, we incorporate up-to 6-wise feature interactions, because previous studies show that almost all faults are caused by the interaction of up-to 6 features [Kuhn et al., 2004; Abal et al., 2014]. Previous work [Bagheri et al., 2012; Sánchez et al., 2014] focused on faults caused by the interaction among the selected features as well, and they do not take deselected features into account. However, with our fault generator, we also consider the deselected features, because deselected features may also cause faults in products [Abal et al., 2014].

In the *exhaustive interaction faults*, the same proportion of each T-wise feature interactions is simulated as faults (i.e., $17\% \approx 100/6$ for each of the following: single feature, 2-wise, 3-wise, 4-wise, 5-wise, and 6-wise feature interaction). We assume the equality of fault numbers for T-wise interaction faults, because we are not aware of a study that reports what the percentage of each T-wise fault is (i.e., 70% of faults are 1-wise interaction faults). Examples of faults simulated for our running example in Figure 2.2 on Page 9 could be as follows: $\{Edges\}$, $\{Algorithm,\ \neg Undirected,\ Cycle\}$ represent

ID	some-selected
1	a
2	$a \wedge b$
3	$a \wedge b \wedge c$
4	$a \wedge b \wedge c \wedge d$
5	$a \wedge b \wedge c \wedge d \wedge e$
ID	combination of selected and not-selected
1	$\neg a$
2	$a \wedge \neg b$
3	$a \wedge b \wedge \neg c$
4	$a \wedge b \wedge c \wedge \neg d$
5	$a \wedge b \wedge c \wedge d \wedge \neg e$
6	$\neg a \wedge \neg b$
7	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$

Table 3.6: Fault patterns [Abal et al., 2014]

two faults, a fault in feature *Edges* and another fault that applies to all configurations where *Algorithm* and *Cycle* are selected and *Undirected* is not selected.

The steps of generating these simulated faults are as follows. We select a value T where $T \in [1 : 6]$. Then, we select T features randomly. We decide for each feature randomly whether it is selected or deselected. After that, we check whether this combination of features is valid according to the feature model using a satisfiability solver (i.e., whether it occurs in at least one configuration) [Mendonça et al., 2009b]. We accept only valid combinations, because invalid combinations are not relevant. We already showed how the valid combinations of features are created in Section 2.2.1.

With *Pattern Interaction Faults*, we base the fault generation on a qualitative study for a set of faults that has been collected from the Linux kernel, BusyBox, Marlin, and Apache repositories [Abal et al., 2014]. Abal et al. [2014] analyze each fault and record their results. They observe that some faults are caused when some features are not selected. They classify their faults as they appear in Table 3.6. All the reported faults are up-to 5-wise feature interactions. We implemented our fault generator to seed faults similar to these reported faults. We generate the same proportion of faults for each pattern due to aforementioned reasons that other distributions are potentially non-representative. We generated 100 sets of faults for each fault type. In each set, as we already mentioned, the number of faults is $n/10$, where n is the number of features.

Experiment Results

In this section, we answer RQ1.2, RQ2, and RQ3.

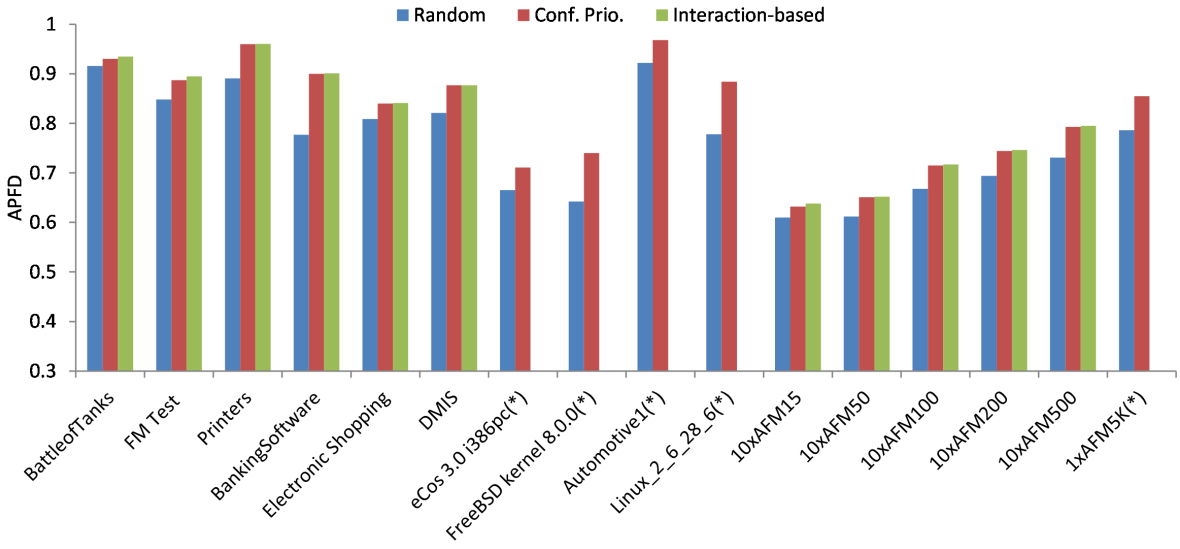


Figure 3.6: Average APFD for random orders, configuration-based prioritization, and interaction-based using *exhaustive interaction faults*.
(*) The computation of interaction-based did not finish within 24 hours.

RQ1.2*: *Can configuration-based prioritization cover* feature interactions *faster than the random orders and the interaction-based approach?

We compare our approach to the average results of 100 experiments for random orders and to interaction-based prioritization using aforementioned fault types. Note that the inputs to configuration-based prioritization, random orders, and the interaction-based approach are configurations created by the sampling algorithm ICPL. In Figure 3.6 and Figure 3.7, we present the average APFD of random orders, configuration-based prioritization, and interaction-based over 100 sets of exhaustive and pattern interaction faults, respectively. We observed that interaction-based prioritization does not scale to feature models larger than 500 features (highlighted with * in Figure 3.6 and Figure 3.7). We abort the process if the computation of interaction-based prioritization did not finish within 24 hours.

As Figure 3.6 and Figure 3.7 reveal, configuration-based prioritization performs better than random orders, as the APFD values of configuration-based prioritization are higher than average APFD of random orders for each product line. We use the Mann-Whitney U test to investigate whether differences between configuration-based prioritization and random orders are significant. We observed that the p-values between APFD values for real and artificial feature models using the exhaustive and pattern interaction faults are approximately zero. That is, the difference between configuration-based prioritization and random orders is significant for all feature models for both fault types. Hence, the evaluation results show that configuration-based prioritization improves the effectiveness of product-line testing compared to random orders.

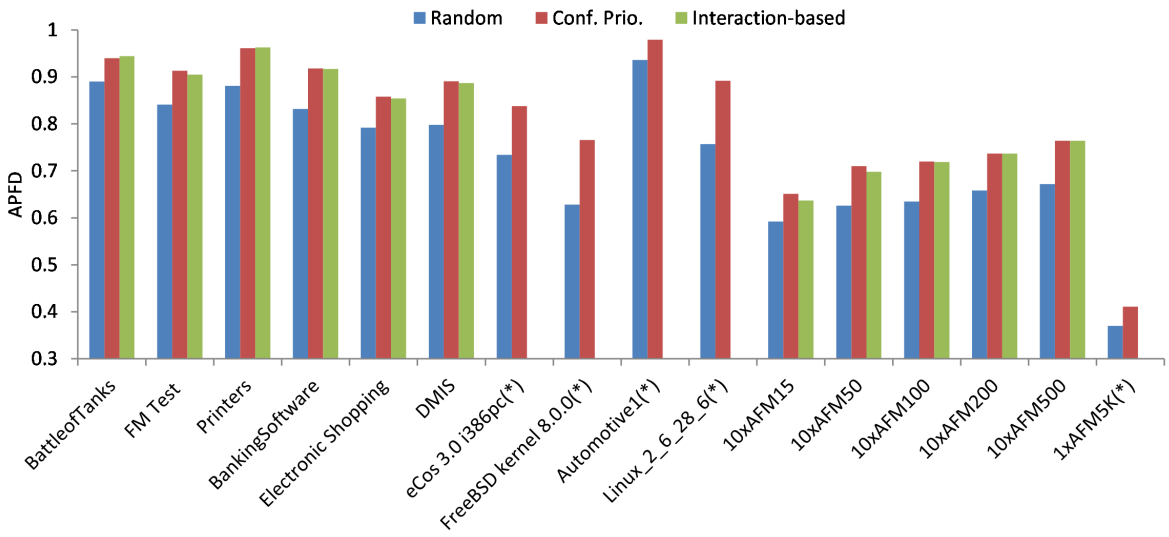


Figure 3.7: Average APFD for random orders, configuration-based prioritization, and interaction-based using *pattern interaction faults*.
(*) The computation of interaction-based did not finish within 24 hours.

Comparing our approach to the interaction-based approach, the results show that both approaches are approximately identical with a slight improvement for one over the other in some product lines. For instance, in Figure 3.6, we observe a slight improvement for the interaction-based approach for some product lines such as *BattleofTank, FM_Test, and BankingSoftware*. In Figure 3.7, we notice that configuration-based prioritization is slightly better than interaction-based approach for some product lines, such as *FM_Test, BankingSoftware, DMIS, 10xAFM15, and 10xAFM50*. The reason for the slight improvement of our approach over the interaction-based approach with the pattern interaction faults is that most of the simulated faults are selected features (cf. Table 3.6 on Page 39). Hence, these faults are detected with the first product (*allyesconfig*). For the first product with the interaction-based approach, it is selected randomly, because any product will cover the same percentage of the feature combinations. Thus, it might be the case that most of the features in the first product are not selected. Another possible reason is that the interaction-based approach is influenced by the T-wise interactions, where T in our case equals 2. On the contrary, configuration-based prioritization is independent of T as it selects the least similar configurations instead of those covering the highest number of interactions.

We use the Mann-Whitney U test to investigate whether the differences between APFD values of configuration-based prioritization and interaction-based approach are significant. In Figure 3.8, we show the distribution of p-values between APFD values for the interaction-based approach and configuration-based prioritization using the exhaustive and pattern interaction faults for real and artificial feature models. In Figure 3.8, we

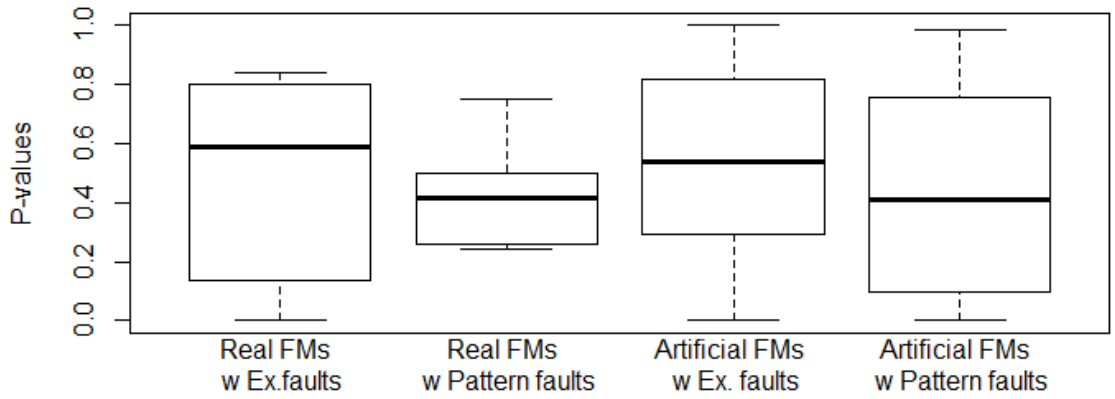


Figure 3.8: The distribution of p-values from Mann-Whitney U Test between interaction-based approach and configuration-based prioritization, (Ex.) exhaustive interaction faults

observe that the median p-values of all box-plots are higher than 0.05. Hence, the difference between our approach and the interaction-based approach is not significant.

Regarding **RQ1.2**, we conclude from the results that our approach performs well compared to an expensive approach such as the interaction-based approach. In addition, the advantage of our approach compared to the interaction-based approach is that it scales well to all used product lines in our evaluation, while it is not the case with interaction-based where it does not scale to product lines larger than 500 features in terms of memory consumption. Using feature models up-to 500 features for which the interaction-based approach scales up, we compare configuration-based prioritization to the interaction-based approach in terms of the computation time. We found that the computation time of configuration-based prioritization is, on average, 11% of the computation time of the interaction-based approach.

RQ2*: *Do sampling algorithms produce samples with an acceptable effective order?

For each sampling algorithm, there is an implicit order influenced by feature coverage. That is, a sampling tool outputs configurations necessarily in a particular order. We investigate whether these algorithms already have good orders as a result of covering as many feature combinations as possible each time a product is generated. Thus, we compare their default orders to configuration-based prioritization.

In Table 3.7 and Table 3.8, we present the average APFD values over 100 sets of faults for each fault type. We highlight higher average values of APFD with a circle. These values are highlighted with an additional background color if the difference between values is significant. The p-values resulting from the Mann-Whitney U test are used

FM	APFD					
	CA-D	CA-P	CH-D	CH-P	ICPL-D	ICPL-P
BattleofTanks	0.915	0.910	0.932	0.931	0.930	0.930
FM_Test	**0.845**	0.833	0.866	**0.887**	0.864	**0.887**
Printers	0.926	0.928	0.961	0.961	0.960	0.960
BankingSoftware	0.849	0.851	0.899	0.896	0.901	0.900
ElectronicShopping	0.829	**0.834**	0.847	0.845	0.841	0.840
DMIS	0.823	0.824	0.876	0.874	0.878	0.877
eCos 3.0 i386pc	*	*	0.723	0.717	**0.718**	0.711
FreeBSD kernel 8.0.0	*	*	0.738	0.739	0.740	0.740
Automotive1	*	*	0.967	0.968	0.968	0.968
Linux 2.6.28.6	*	*	*	*	0.884	0.884
10xAFM15	0.595	**0.605**	0.637	0.640	0.633	0.632
10xAFM50	0.608	**0.616**	0.643	0.644	0.646	**0.651**
10xAFM100	0.698	**0.705**	0.713	0.715	0.714	0.715
10xAFM200	*	*	0.744	0.744	0.743	0.744
10xAFM500	*	*	0.791	0.791	0.793	0.793
10xAFM1000	*	*	0.817	**0.818**	0.817	0.818
1xAFM5K	*	*	*	*	0.854	0.855
Average	**0.709**	**0.711**	**0.797**	**0.811**	**0.817**	**0.818**

CA: CASA algorithm
CH: Chvatal algorithm
D: Default order of each algorithm
P: Configuration-based prioritization approach
*: No result within a whole day of computation
: The corresponding APFD value is higher than the compared value
: The difference between the corresponding APFD value and the compared value is significant

Table 3.7: Average APFD for default order of sampling algorithms and configuration-based prioritization using *exhaustive faults*.

to investigate whether the difference is significant. We report in Table 3.9 the p-values between APFD values of our approach and the default orders of sampling algorithms for both fault types.

In Table 3.7, we show the average APFD values for each feature model using the *exhaustive interaction faults*. We observe that the average APFD values of configuration-based prioritization for all product lines are higher than the average APFD values of the sampling algorithms. If we look at the results for each product line separately, we observe that they are varying. For instance, comparing to the default order of sampling algorithm CASA, we find that our approach is better, on average, in most cases. In particular, our approach is significantly better for product lines *Electronic Shopping*,

FM	APFD					
	CA-D	CA-P	CH-D	CH-P	ICPL-D	ICPL-P
BattleofTanks	0.913	**0.918**	0.944	0.942	0.942	0.940
FM_Test	0.821	**0.870**	0.852	**0.911**	0.850	**0.913**
Printers	0.923	**0.927**	0.961	0.960	0.962	0.961
BankingSoftware	0.838	**0.867**	0.910	0.911	0.916	0.918
Electronic Shopping	0.833	**0.850**	0.856	0.862	0.854	0.858
DMIS	0.825	**0.828**	0.879	0.884	0.885	0.891
eCos 3.0 i386pc	*	*	0.844	0.846	0.835	0.838
FreeBSD kernel 8.0.0	*	*	0.764	0.767	0.764	0.766
Automotive1	*	*	0.967	0.968	0.968	0.968
Linux 2.6.28.6	*	*	*	*	0.890	0.892
10xAFM15	0.573	**0.612**	0.634	**0.658**	0.632	**0.651**
10xAFM50	0.651	**0.684**	0.692	**0.707**	0.686	**0.710**
10xAFM100	0.689	**0.720**	0.713	**0.717**	0.717	0.720
10xAFM200	*	*	0.735	**0.741**	0.735	0.737
10xAFM500	*	*	0.763	**0.764**	0.762	**0.764**
10xAFM1000	*	*	0.783	**0.784**	0.781	**0.782**
1xAFM5K	*	*	*	*	0.411	0.411
Average	**0.707**	**0.728**	**0.812**	**0.828**	**0.799**	**0.807**

CA: CASA algorithm
CH: Chvatal algorithm
D: Default order of each algorithm
P: Configuration-based prioritization approach
*: No result within a whole day of computation
: The corresponding APFD value is higher than the compared value
: The difference between the corresponding APFD value and the compared value is significant

Table 3.8: Average APFD for default order of sampling algorithms and configuration-based prioritization using *pattern interaction faults*.

10xAFM15, *10xAFM50*, and *10xAFM100* with p-values 0.000, 0.001, 0.000, and 0.000, respectively. However, we observe that our approach is significantly worse only for product line *FM_Test* with p-value 0.027. Comparing our approach to the default order of sampling algorithm Chvatal, we observe that the average APFD values over all product lines of configuration-based prioritization and Chvatal are 0.811, and 0.797, respectively. We notice that the average values of the default order of Chvatal are better but not significantly for five product lines, while the average values of our approach are better but not significantly for five product lines and significantly better for product lines *FM_Test* and *10xAFM1000* with p-values 0.000 and 0.004, respectively. The results of configuration-based prioritization compared to sampling algorithm ICPL are as follows. With our approach, the average APFD values for all product lines are higher

FM	Exhaustive faults			Pattern faults		
	CA	CH	ICPL	CASA	CH	ICPL
BattleofTanks	0.059	0.396	0.846	**0.037**	0.195	0.537
FM_Test	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
Printers	0.386	0.8719	0.938	**0.010**	0.971	0.781
BankingSoftware	0.461	0.956	0.970	**0.000**	0.687	0.411
Electronic Shopping	**0.027**	0.906	0.869	**0.000**	0.159	0.159
DMIS	0.395	0.151	0.314	**0.037**	0.195	0.537
eCos 3.0 i386pc	*	0.083	**0.023**	*	0.344	0.425
FreeBSD kernel 8.0.0	*	0.797	0.745	*	0.382	0.580
Automotive1	*	0.579	0.705	*	0.739	0.796
Linux 2.6.28.6	*	*	0.854	*	*	0.094
10xAFM15	**0.001**	0.481	0.398	**0.000**	**0.000**	**0.000**
10xAFM50	**0.000**	0.707	**0.025**	**0.000**	**0.000**	**0.000**
10xAFM100	**0.000**	0.220	0.561	**0.000**	**0.027**	0.099
10xAFM200	*	0.850	0.230	*	**0.002**	0.095
10xAFM500	*	0.448	0.815	*	**0.024**	**0.005**
10xAFM1000	*	**0.004**	0.054	*	**0.0325**	**0.001**
1xAFM5K	*	*	0.643	*	*	0.837

CA: CASA algorithm

CH: Chvatal algorithm

*: No result within a whole day of computation

Table 3.9: P-values of the Mann-Whitney U test between APFD values of configuration-based prioritization and the default orders of sampling algorithms for both fault types.

than the average APFD values of algorithm ICPL. However, in Table 3.7 on Page 43, we notice that APFD values of configuration-based prioritization are higher and lower than the APFD values of ICPL for six product lines, and five product lines, respectively. Except for CASA where our approach outperforms its effectiveness, the APFD values of sampling algorithms default orders are close to the APFD values of our approach. The reason is that these algorithms are greedy (cf. Section 2.2.2) and they cover feature combinations as fast as possible.

Using the *pattern interaction faults*, we show the results of comparing our approach to the default order of sampling algorithms in Table 3.8. We observed that average APFD values over 100 sets of faults of our approach for all product lines are higher than the average APFD values of sampling algorithms. In addition, we report the p-values for each product line in Table 3.9.

Looking closer to the results for each sampling algorithm, we remark that our approach is significantly better than the default order of sampling algorithm CASA for all product lines.

Comparing the default order of sampling algorithm Chvatal to our approach, we observe that our approach is significantly better for seven product lines and, on average, better but not significantly for six product lines. Using sampling algorithm ICPL, the results show that our approach is significantly better for five product lines, and, on average, better but not significantly for eight product lines.

Since our approach is significantly better than random orders and by considering the results of the comparison to the default orders of sampling algorithms (cf. Table 3.7 on Page 43 and Table 3.8 on Page 44), we show that the investigated sampling algorithms have already an acceptable effective order in terms of fault detection rate. With *pattern interaction faults*, we notice that the average APFD values of our approach are higher for most product lines than the default orders of sampling algorithms. However, it is not the same situation with *exhaustive interaction faults*. We assume the reason is that the number of deselected features in faults affects the results. The number of deselected features using *exhaustive interaction faults* is more than the number of deselected features using *pattern interaction faults* (cf. Table 3.6 on Page 39). In contrary to the pattern interaction faults, most of these deselected features in exhaustive interaction faults cannot be detected with the first product, since we select the product with maximum number of features. In response to *RQ*2, although these sampling algorithms have effective orders of products, we show that our approach can be potentially helpful in many cases to increase the rate of early fault detection. In addition, the order of algorithm CASA is non-deterministic for all product lines. For ICPL and Chvatal, we noticed that their orders are non-deterministic for some product lines, such as product lines *BattleofTanks* and *FM_Test*. With our approach, the order is always deterministic.

From the reported results, the default orders of these sampling algorithms already show promising results, which also can be improved with our approach. However, we argue that configuration-based prioritization might have a major impact if the given products generated by end users or sampled randomly. To validate the aforementioned argument, we conduct another experiment where we generated a set of configurations randomly. In particular, we first sampled 20 thousand configurations randomly for each real feature model (cf. Table 3.5 on Page 36). We sampled this large number of configurations to reduce the possibility of generating unrepresentative configurations. Second, we select 100 configurations randomly to be used as an input to our approach. We repeated these two steps 100 times to mitigate the impact of randomness. Note that the difference between this experiment and the random orders mentioned in **RQ1.2** is that the configurations in that experiment are created using the sampling algorithm ICPL and then reordered randomly. The results show that configuration-based prioritization outperforms the orders of these configurations sampled randomly. In particular, using the Mann-Whitney U test, we observe that the differences between configuration-based prioritization and the random orders of these random configurations are significant for all considered feature models with p-values ≈ 0. We also repeat the experiment with different numbers of configurations (e.g., 50 configurations). The reported results of all experiments are very similar.

Feature Model	CA w/ P.	P. %	CH w/ P.	P. %	ICPL w/ P.	P. %
BattleofTanks	25,493.05	1.16%	101.22	83.52%	97.65	91.69%
FM_Test	9,045.40	0.002%	12.63	1.52%	4.46	4.20%
Printers	680.33	1.02%	17.92	34.69%	11.06	56.33%
BankingSoftware	3,656.32	0.004%	15.41	0.97%	3.70	4.73%
Electronic Shopping	3,940.84	0.004%	29.30	0.26%	2.46	3.01%
DMIS**	12,451.14	0.001%	49.75	0.21%	4.54	2.63%
eCos 3.0 i386pc	*	*	1,354.33	0.15%	91.16	2.09%
FreeBSD kernel 8.0.0	*	*	2,184.73	0.053%	110.64	1.09%
Automotive1	*	*	66,431.05	4.60%	12,225.89	23.68%
Linux 2.6.28.6	*	*	*	*	25,102.04	3.94%
10xAFM15	1.81	0.08%	0.06	5.47%	0.10	10.56%
10xAFM50	39.31	0.03%	0.99	1.58%	0.28	5.53%
10xAFM100	1,063.02	0.01%	6.66	2.01%	1.39	10.60%
10xAFM200	*	*	31.10	3.19%	4.96	21.86%
10xAFM500	*	*	1,492.23	1.16%	733.31	2.63%
10xAFM1000	*	*	4,740.29	7.45%	1,196.58	28.66%
1xAFM5K	*	*	*	*	45,498.94	7.80%

CA: CASA algorithm

CH: Chvatal algorithm

w/ P. The computation time of sampling and prioritization processes

P: The computation time percentage of prioritization

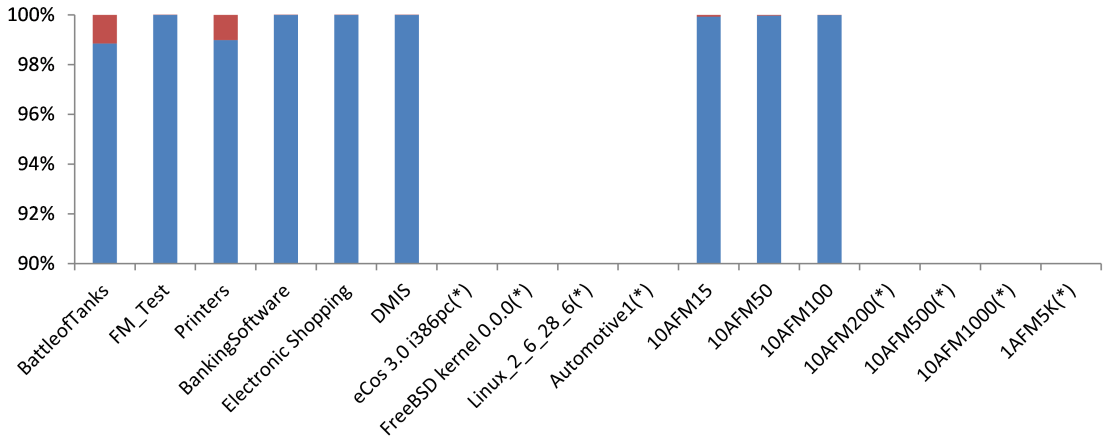
*: No result within a whole day of computation

Table 3.10: The percentage of average execution time in seconds of prioritization to the sampling of each algorithm.

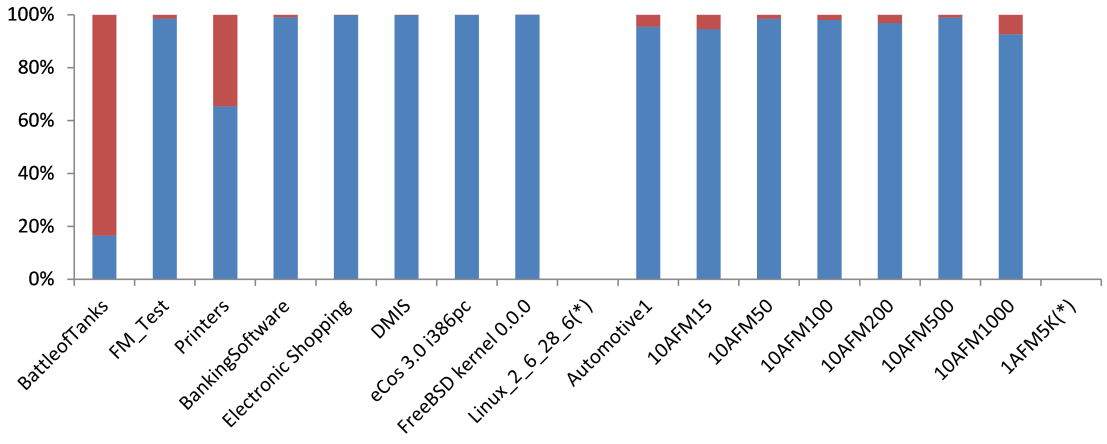
RQ3*: *Is the computational overhead required for configuration-based prioritization negligible compared to the efforts required for sampling?

We computed the average execution time to sample and prioritize products. Then, we compute the percentage of that execution time which is needed to achieve the prioritization process. To accelerate the process, we performed our experiments using two computers. The first one with an Intel Core i5 CPU @ 3.33 GHz, 16 GB RAM, and Windows 7. On this computer, we run the experiment for the following large product lines: *eCose 3.0 i386pc, FreeBSD kernel 8.0.0, Automotive1, Linux 2.6.28.6, and 1xAFM5K.* The second computer with an Intel Core i5 CPU @ 3.33 GHz, 8 GB RAM, and Windows 7 where we conducted experiments with the remaining product lines.

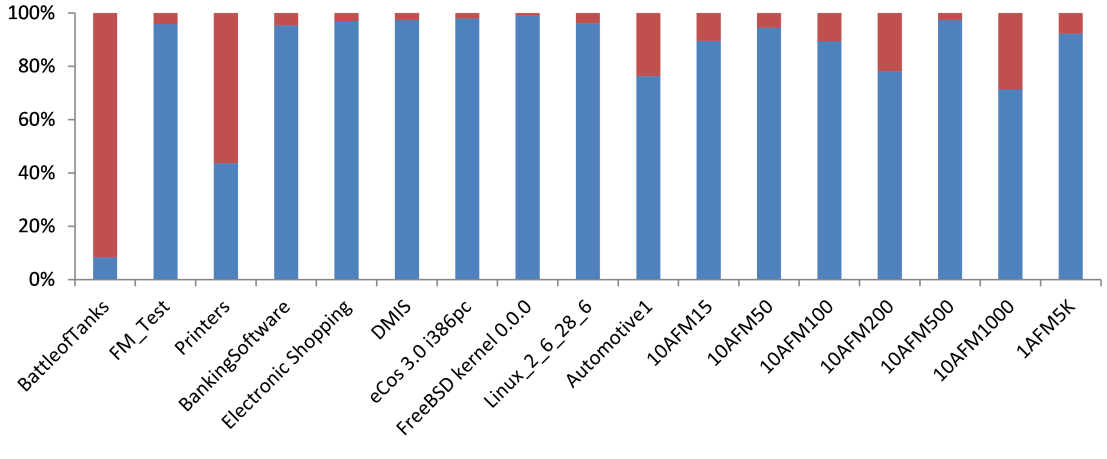
In Table 3.10, we show the average execution time over five runs to sample and prioritize products for the listed product lines. In addition, we present the percentage of prioritization time to the total computed execution time (i.e., time of both tasks: sampling and prioritization). Using the subject product lines, in Figure 3.9, we show



(a) CASA



(b) Chvatal



(c) ICPL

Figure 3.9: The percentage of average execution time of prioritization to the combined time of sampling and prioritization of each algorithm.
(*) The computation of sampling did not finish within 24 hours.

the percentage of prioritization time to the combined time of sampling and prioritization for CASA 3.9(a), Chvatal 3.9(b), and ICPL 3.9(c). In the case of the sampling algorithm CASA, we noticed that the time of prioritization can be negligible since the percentage of prioritization time for each product line is less than 1.2% of the computation time to sample and prioritize products. Compared to sampling algorithm Chvatal, the percentages of prioritization time range between 0.053% and 83.52%.

For ICPL, the percentages of prioritization time range between 1.09% and 91.69%. From the results (cf. Figure 3.9 and Table 3.10), we found only two product lines where the percentage of prioritization is more than 30% of the overall execution time. These product lines are *BattleofTanks* and *Printers*. We investigated these cases, and we found that it occurs when the feature models have no constraints and the number of generated configurations is at least twice as much as the number of features. Thus, the sampling process is computationally cheap. It does not require much time to validate whether these configurations are valid or not. In addition, for these two feature models, our approach takes relatively longer in the range of seconds (cf. Table 3.10), and thus neglectable. Looking closer to the largest product lines (the Linux kernel and the 1xAFM5K), the prioritization process requires 3.94% and 7.8% of the overall execution time, respectively. Regarding **RQ3**, as we show in Figure 3.9 and Table 3.10, the average execution time of configuration-based prioritization for most cases is small compared to the efforts required for sampling, especially for the large product lines.

We conclude that configuration-based prioritization is significantly better than random orders with respect to the rate of early fault detection. Furthermore, the default orders of the compared sampling algorithms already show promising results, which, however, can even be improved cheaply in many cases using configuration-based prioritization.

3.4 Cluster-Based Product Prioritization

In some cases, especially in regression testing, if the diversity between products is large, the cost of testing/analyzing these products will be increased due to the redundancy in test cases execution [Lity et al., 2017] as well as the amount of the required time to set up the testing environment (e.g., in the automotive domain). Furthermore, due to the extreme diversity, it could be the case that different testing techniques or even different prioritization techniques can be applied to each subset of the generated products [Busjaeger and Xie, 2016]. Thus, we investigate a potential solution to overcome the aforementioned challenges. Our idea is to cluster products based on their feature selections to identify those that are syntactically similar. Clustering products can be useful to optimize the testing setup time by clustering products based on a specific parameter into groups. That is, the same testing setup can be used to test the whole products in a group. Moreover, clustering products facilitates testing if developers want to focus on a set of products (i.e., the most demanded products). If they want to cover dissimilar products, testers may select a subset of products from each cluster. In addition, clustering can be used to test products based on the testing history. For instance,

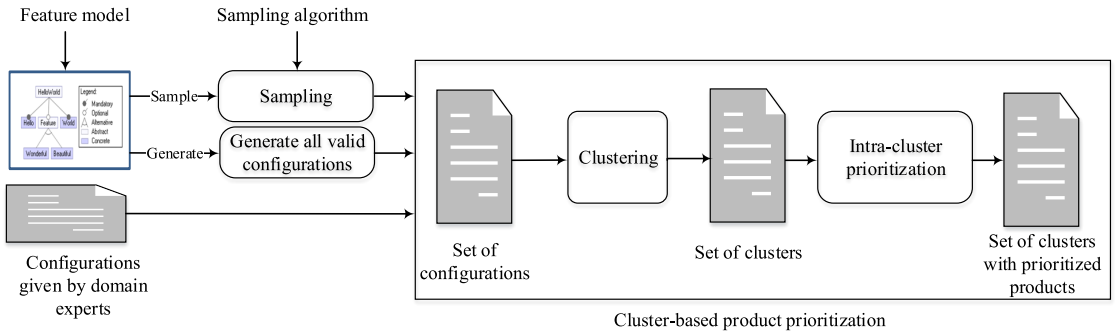


Figure 3.10: An overview of clustering-based product prioritization

testers may want to focus on products that contain certain artifacts in which many faults have been found in previous testing runs.

We evaluate our approach using product lines of different sizes and compare it against random orders and our configuration-based prioritization approach (cf. Section 3.1). The results for cluster-based prioritization show potential improvement in the effectiveness of product-line testing (i.e., increasing the early rate of fault detection).

More precisely, we contribute the following:

- We propose a cluster-based prioritization approach to cluster products. This allows us to group products and prioritize them.
- We evaluate our approach compared to a configuration-based prioritization (cf. Section 3.1) and random orders.
- We assess the impact of having different cluster numbers.

The rest of this section is organized as follows. In Section 3.4.1, we introduce the cluster-based prioritization approach. We present and discuss the results in Section 3.4.2.

3.4.1 Overview on Cluster-Based Prioritization

The main goal of our approach is to cluster products into different subsets such that products in each group share common properties. As we illustrate in Figure 3.10, the input for our approach is a set of configurations. In the following, we describe the two main steps, clustering and prioritization, of our approach, which we display in Figure 3.10. Clustering enables testers to cluster products into groups (e.g., cluster with the most demanded products). In addition, testers may wish to prioritize products within clusters based on certain criteria, such as the coverage, to find faults faster. For the latter, we use configuration-based prioritization (cf. Section 3.1) to prioritize products.

Clustering

The commonality of configurations is measured with a clustering criterion. In this thesis, we consider the similarity between products in terms of the selected features (their *configuration*, cf. Table 2.1 on Page 15) as criterion. In particular, we identify the similar products syntactically, in terms of features, and group them into clusters. The resulting clusters allow testers to select a sample of products from each or only a particular cluster. While we consider the simple *K-means* algorithm to cluster products, other clustering algorithms can be considered in future to investigate whether they may influence the results.

Prioritization

In order to prioritize clustered products, we recognize two layers of prioritization:

1. *Intra-cluster* prioritization addresses the order of products in a cluster.
2. *Inter-cluster* prioritization addresses the order of clusters themselves.

In our work, we only consider *intra-cluster* prioritization to prioritize products. Considering *inter-cluster* prioritization (i.e., from which clusters products are tested first) requires additional domain knowledge, for instance, which cluster contains more demanded products than others. Hence, during our evaluation, we rely on the cluster ordering that is given by the clustering algorithms.

To prioritize products in a cluster, we use configuration-based prioritization (cf. Section 3.1), that is, products are prioritized based on the similarity of their feature selections. The product that is least similar to *all* previously tested ones is selected to be tested next. The goal of considering the configuration-based prioritization approach is to increase the interaction coverage for products under test inside a particular cluster as soon as possible.

3.4.2 Evaluation of Cluster-Based Prioritization

In this section, we formulate *research questions*, introduce our *subject systems*, and explain the *experiment settings* of our evaluation. Finally, we present and discuss the *results*.

We assess the effectiveness of cluster-based prioritization in terms of its fault detection rate compared to configuration-based prioritization (cf. Section 3.1) and random orders. With configuration-based prioritization, we handle all products as they are all in one cluster. To evaluate the impact of considering the *intra-cluster* prioritization, we compare it to the default order given by the clustering algorithm. In particular, we answer the following questions:

RQ1 How does cluster-based prioritization perform compared to configuration-based prioritization and random orders?

RQ2 How does *intra-cluster* prioritization influence the effectiveness of testing compared to using the default order provided by clustering algorithms?

RQ3 How does the number of clusters influence the effectiveness of testing?

We consider a variety of subject systems from academia as well as real-world systems to evaluate our approach. Due to the lack of having open-source software product lines that have test cases and faults in the source code, we conduct our experiment using the feature models of the subject systems. We considered all real feature models in Table 3.5 (cf. Page 36) as well as an artificial feature model with 5,542 feature and excluded all other artificial feature models as they are relatively small.

Experiment Settings

Given the feature models, we now explain our methodology for conducting the evaluation. In particular, we provide details about the faults to be detected and the applied clustering algorithm.

Artificial Faults Due to the aforementioned reason of the lack of real-world product lines, we use simulated faults to evaluate our approach. For this, we applied a technique that has been used in previous studies on product-line testing [Ensan et al., 2012; Sánchez et al., 2014; Devroey et al., 2016]: we randomly selected and marked features as containing faults. To simulate reality, we used the *pattern interaction faults* that have been generated to evaluate the configuration-based prioritization (cf. Section 3.3.3).

K-Means Clustering Algorithm For our evaluation, we use the Waikato Environment for Knowledge Analysis (Weka) [Hall et al., 2009] version 3.8, an open source tool for machine learning and data mining. It provides several clustering algorithms, such as simple *K-means* and Hierarchical Cluster. In this thesis, we use the simple *K-means* algorithm to cluster products, as it needs no deeper knowledge about clustering algorithms, thus, not biasing the results, and also can serve as a baseline for applying other algorithms in the future. To answer our third research question (RQ-3), we consider three values for K (i.e., a different number of clusters): $K = 5$, $K = 10$, and $K = 15$. To measure the proposed approach, we use the APFD metric developed by Elbaum et al. [Elbaum et al., 2000] to evaluate the effectiveness of fault detection (cf. Section 3.3.1).

Results and Discussion

Regarding **RQ1**, we compare cluster-based prioritization with $K = 5$, $K = 10$, and $K = 15$ to the configuration-based prioritization (cf. Section 3.1) and random orders with respect to the fault detection rate. We show in Table 3.11 that the average APFD values of cluster-based prioritization are higher than these of random orders for all

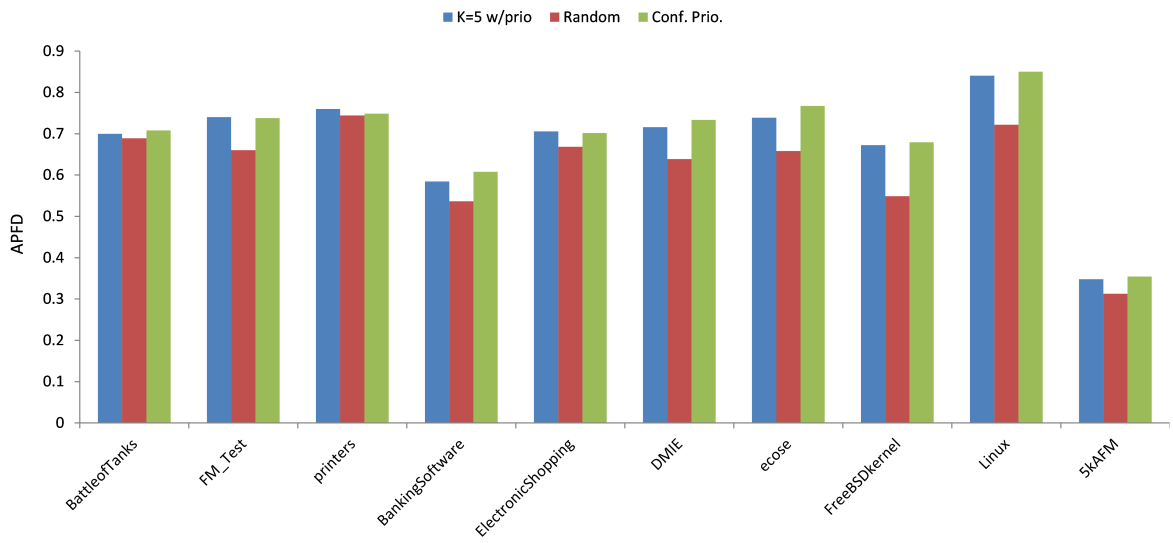


Figure 3.11: Average APFD for random orders, cluster-based prioritization, and configuration-based prioritization

	Clustering			Random	Conf. Prio.
	$K = 5$	$K = 10$	$K = 15$		
APFD Average	0.681	0.667	0.669	0.618	0.689

Table 3.11: APFD average values for clustering-based prioritization, Random order, and configuration-based prioritization

product lines, regardless the number of clusters. In particular, the APFD values for cluster-based prioritization with $K = 5$, $K = 10$, and $K = 15$ are, receptively, 0.681, 0.667, and 0.669, while the average APFD value of random order is 0.618. Hence, the improvements are 10.2%, 9.5%, and 8.3%.

If we look at the results for each feature model separately (cf. Figure 3.11), we observe that for each product line, cluster-based prioritization is, on average, better than random orders. To support our observation, we apply the Mann-Whitney U test, a non-parametric statistical test, to investigate whether the differences to our approach are significant. We observe that the difference between cluster-based prioritization and random orders is significant for all product lines, except *printers* and *BattleofTanks*, with p-values 0.30 and 0.34, respectively. The detailed results of the experiment, including all p-values, are listed in the Appendix Section A.2.

Furthermore, from Figure 3.11 and Table 3.11, we observe that the average APFD value of heuristic configuration-based prioritization (0.689) is slightly higher than the values for cluster-based prioritization. This applies especially with $K = 5$ (0.681) for which the percentage of decrease is 1.2%.

	W/prioritization	Wo/prioritization	Improvement (%)
$K = 5$	0.681	0.668	3.18%
$K = 10$	0.677	0.663	2.10%
$K = 15$	0.669	0.650	2.90%

Table 3.12: APFD average values for clustering-based with and without intra-cluster prioritization for different number of clusters

Regarding **RQ2**, we compare our approach to cluster-based prioritization without considering *intra-cluster* prioritization (i.e., taking the default order given by the clustering algorithm). In Table 3.12, where we show the average value of APFD for $K = 5$, $K = 10$ with and without considering the intra-cluster prioritization inside clusters, we found that the average APFD values are higher when applying *intra-cluster* prioritization for $K = 5$, $K = 10$, and $K = 15$ with 0.681, 0.677, and 0.669, respectively than without *intra-cluster* prioritization, where the values decrease to 0.660, 0.663, and 0.650. Hence, the percentages of improvement of considering *intra-cluster* prioritization are 3.18%, 2.1%, and 2.9%. Still, we notice that the improvement of prioritizing products within clusters is relatively small.

Regarding **RQ3**, we see that for our cluster-based prioritization, fewer clusters result in higher APFD values. As we show in Table 3.12, the average APFD values for $K = 5$, $K = 10$, and $K = 15$ are 0.681, 0.677, and 0.669 respectively. The percentages improvement with $K = 5$ compared to $K = 10$ and $K = 15$ are 0.5% and 1.8%. The reason for the slight improvement is the impact of *intra-cluster* prioritization on the results. Fewer clusters result in more products within each cluster and, thus, a higher impact of *intra-cluster* prioritization. The following supports the aforementioned reasoning: Without considering *intra-cluster* prioritization the results are varying. For instance, we observe that the average APFD values for $K = 5$, $K = 10$, and $K = 15$ are 0.660, 0.663, and 0.650 respectively. In particular, the percentages of improvement of $K = 10$ compared to $K = 5$ and $K = 15$ are 0.4%, and 2.0%. We conclude that fewer clusters improve the results if *intra-cluster* prioritization is considered.

To summarize our findings, we answer our research questions as follows:

RQ1 On average cluster-based prioritization performs better than random orders but slightly worse than configuration-based prioritization. However, cluster-based prioritization enables testers to select subsets of all products (e.g., select the cluster with the most demanded products), which cannot be done easily with configuration-based prioritization, as it requires to compare all products instead of clusters.

RQ2 Considering the default order of clusters is slightly worse than prioritizing products overall or in a cluster. Still, clustering provides comparable results and further investigations seem promising.

RQ3 A higher number of clusters decreases the APFD value on average. Hence, increasing the number of clusters influences badly on the testing effectiveness as a result of the randomness of selecting the center point of each cluster.

That is, clustering with and without intra-cluster prioritization shows promising results compared to configuration-based prioritization and random orders. However, further investigation using other clustering algorithms as well as different criteria is required.

3.5 Threats to Validity

In this section, we discuss the threats to validity of our experiments that may affect our results and explain the steps that we took to mitigate those threats.

Experiment with Code Base of Product Lines

Our implementation of configuration-based prioritization could contain faults itself, which may affect the *internal validity*. To overcome this threat, we applied our algorithm on small product lines and analyzed the results manually. Our implementation and experiment data are publicly available for inspection and reproduction purposes.[3] Furthermore, configuration-based prioritization is integrated into testing functionalities of FeatureIDE.

A potential *external threat* is that we acquired an intimate knowledge of the subjected product lines for reproduction purposes. However, as configuration-based prioritization is a push-button approach and only based on selection and deselection of features, this threat cannot affect the reported results. A further potential threat that may affect the *external validity* is related to the nature of the subject product lines. We are not able to generalize that the proposed approach will provide the same results for other product lines. We were restricted to these three product lines, because we are not aware of other product lines publicly available with their source code and faults in their code (i.e., interaction faults) and at the same time, test cases to detect these faults. In particular, results may depend on fault distribution, test coverage, size of the code base, number of features, and number of products. However, all these product lines served as benchmarks previously to evaluate verification strategies of product lines [Apel et al., 2013c; Meinicke et al., 2016b]. To reduce these threats, we conducted another experiment with feature models of product lines and simulated test execution and fault detection.

The main *construct validity threat* is the APFD metric that we used to evaluate the testing effectiveness of the compared approaches. To our knowledge, APFD is widely accepted metric for evaluating prioritization approaches [Kuhn et al., 2013; Rothermel et al., 2001; Elbaum et al., 2002; Li et al., 2007; Yoo and Harman, 2012; Qu et al., 2007; Henard et al., 2016; Sánchez et al., 2014; Walcott et al., 2006]. A potential limitation of

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/thesis

the APFD metric is that it does not assign values for the subsequent products that may contain the already detected faults, which, however, may be helpful, especially in the debugging process. Note that we use the Mann-Whitney U test to investigate whether the differences between the APFD values of the compared approaches are significant (confidence level 95%).

Experiments with Feature Models

There is a potential threat that may affect the *internal validity* related to the distribution of artificial faults in the experiments with feature models. We assume that the faults are equally distributed over the features in a product line. This is rather problematic in testing as faults are often found where they are not expected. Still, assuming equally distributed faults seems to be better than to build on non-idealized, but potentially also non-representative distributions.

Another threat to *internal validity* regarding the artificial faults is that we defined these faults as valid combinations of features. We assume that, if a combination of features is covered in a configuration, the faults will be detected. However, in practice, faults may exist, but are not found. With *exhaustive interaction faults*, we assume the equality of proportion of faults for T-wise interaction faults, because we are not aware of a study that reports the percentage of each T-wise fault. Although Abal et al. [2014] list the number of faults for each pattern, we assume the equality of fault numbers with *pattern interaction faults*, for the aforementioned reasons that other distributions are likely non-idealized distributions. In addition, these pattern faults are based on an analysis of real interaction faults in the Linux kernel, which may raise threats related to the applicability of these patterns to all product lines. Despite the distribution of the artificial faults in the experiment with feature models may not realistic, we are now more confident because we got the similar results for realistic fault distribution in the experiment with code base of product lines.

An *internal validity* threat related to the default orders of sampling algorithms is that they are non-deterministic. For instance, different runs of sampling algorithm CASA with the same input often lead to different orders and even different numbers of configurations. In contrast, for Chvatal and ICPL, the results for the same input differ only in slight changes to the order. To reduce these random effects, we conducted those experiments five times and we showed average results of these experiments. Since we compare configuration-based prioritization to random orders, there are also random effects, which we mitigate by 100 repetitions for each run.

A potential *external validity* threat related to the nature of feature models is that configuration-based prioritization may not provide similar results for different feature models. To alleviate this threat, we use all feature models in the S.P.L.O.T. repository that have more than 140 features. In addition to 61 artificial feature models of different sizes, we include large feature models consisting of up-to 6,888 features. The number of configurations for some feature models is large, for example, using ICPL, the numbers of configurations for *BattleofTanks* and *Automotive1* are 459 and 913, respectively.

Another external threat related to the nature of the features in the subject product lines and feature models is that the results cannot be generalized to the non-functional features, since our focus was on functional testing. However, future research should investigate how to consider non-functional features in prioritizing products is required as it may require additional effort to model and compute these features.

Furthermore, the selected clustering algorithm may influence the results. Thus, other algorithms in future should be considered to investigate whether different clustering algorithms influence the testing effectiveness. In addition, the number of clusters could also influence the results. To mitigate this threat, we considered different numbers of clusters.

3.6 Related Work

In this section, we discuss existing works that are related to product prioritization based on different criteria.

Product prioritization based on feature selection

Several approaches have been proposed to prioritize products based on different criteria. Sánchez et al. [2014] propose five prioritization criteria based on common feature model metrics to prioritize products and compare between these criteria with respect to the rate of fault detection. They observe that different orderings of the same products of a product line may lead to a significant difference in the rate of early fault detection. In their criteria, they do not take the deselected features into account, which, however, is crucial according to Abal et al. [2014]. In addition, in their evaluation part, they generate T-wise faults where $T \in [1:4]$. In our evaluation, we propose two types of faults, one type up-to 6-wise faults (i.e., exhaustive faults). The other type of faults (i.e., pattern interaction faults) is based on an empirical study of the Linux kernel and BusyBox [Abal et al., 2014]. Moreover, we propose cluster-based prioritization to enable testers to focus on a subset of products by clustering them into groups.

Henard et al. [2014b] sample and prioritize products at the same time by employing a search-based approach to generate products based on similarity among them. Moreover, they also propose to prioritize a given set of existing configurations, which can be obtained elsewhere, such as sampling algorithms [Henard et al., 2013b]. Configuration-based prioritization differs from the previous work, as follows:

- the selection of the initial (allyesconfig) configuration, which is the one with the maximum number of selected features,
- the choice of the distance function (Hamming, an edit-based distance vs Jaccard a token-based one),

- the way prioritization is computed when more than two products are already selected (e.g., taking the maximum of the minimum distances instead of taking sums into account for a more fine-grained evaluation with respect to a coarse-grained approach adopted by Henard et al. [2014b]), and
- the way of ordering (i.e., our proposed approach is deterministic while their approach is not due to the nature of the search-based approaches).

In addition, in case testers wish to focus on a subset of configurations, we propose to cluster products into groups based on feature selections.

Devroey et al. [2016] propose a search-based approach that considers the similarity between feature selections to increase the diversity of the selected test cases for behavioral product lines. Although we do not consider test case selection in our work, we argue that our approach can increase the diversity of the selected test cases in case they are mapped to the products or to the selected features in a product. Henard et al. [2013c] mutate feature models to validate the observation that testing dissimilar products has the ability to detect more faults than the similar ones. In their approach, they propose two mutation operators to mutate feature models and they evaluate the generated products against these mutants. They report that considering dissimilar products to be tested can detect more faults than similar ones. In our work, we confirm the previous observation, but by evaluating our approaches using product lines with real faults as well as simulated ones.

Product prioritization using domain knowledge

Several approaches have been proposed to prioritize products based on additional domain knowledge. Johansen et al. [2012b] prioritize products by giving a weight on T-wise interactions based on market knowledge. The idea behind their approach is to generate and prioritize only a set of products that is required in the market. However, such market knowledge of product lines is often not available. Similarly, Ensan et al. [2011] consider the expectations of the domain experts in product prioritization. The experts expect the desirable features and as a result, they give a priority to products that have these desirable features over other products. With our approaches, we do not need domain knowledge as well as domain experts as we follow black-box approach, where we only need the feature selections of each product.

Baller et al. [2014] introduce a framework to prioritize products under test based on the selection of adequate test suites with regard to cost and profit objectives. The limitation of this approach is that it requires in advance the set of all products and its relation to test cases and test goals. To tackle this limitation, Baller and Lochau [2014] propose an incremental test suite optimization approach for product-line testing that uses a symbolic representation in terms of feature constraints. However, further experiments are required to evaluate the effectiveness of their approach using real-world product lines. In our approach, we use the similarity among configurations in terms of features as criteria to prioritize them.

Devroey et al. [2014] perform statistical analysis of a usage model for a product line in order to select the products with high probability to be executed. A drawback of these approaches is that they require analysis of a usage model to be given a priori. Sánchez et al. [2015] prioritize products based on their non-functional attributes, such as feature size and the number of changes in a feature. In a following work, Parejo et al. [2016] consider the aforementioned criteria as inputs to their search-based algorithm by modeling the product prioritization as a multi-objective optimization problem. However, some of these required information are often not given in practice, especially in black-box testing. In addition, their approach is non-deterministic, as it is the nature of the search-based algorithm. Configuration-based and cluster-based prioritization does not require more domain knowledge than the selected and deselected features of each configuration. That is, we follow a purely model-based black-box approach solely requiring feature selection as a basis. Furthermore, configuration-based product prioritization can be applied to other product-line analysis strategies beyond testing, such as statistical analysis, type checking, and theorem proving.

Test-case prioritization

In product-line testing, test-case prioritization is used to reschedule test-case executions in order to increase the rate of fault detection. Baller et al. [2014] present a multi-objective approach to optimize the test suites by considering certain objectives such as cost of test cases, cost of test requirements, and products. With their approach, a domain knowledge is required, which is not often available. In our approach, we only need the feature selections of products under test. Lachmann et al. [2015] propose an integration testing approach for product lines based on delta-oriented test specifications. They reduce the number of executed test cases by selecting the most important test cases for each product. In following work, Lachmann et al. [2016] combine a prioritization based on structural and behavioral deltas with a dissimilarity approach to prioritize message sequence chart test cases. In our work, we focus on product prioritization. However, since these approaches mainly target test-case prioritization, combining these approaches to configuration-based as well as cluster-based prioritization may enhance the effectiveness of product-line testing.

In single-system testing, Henard et al. [2016] compare different black-box prioritization approaches with respect to the detection fault rate. They report that combinatorial interaction testing and similarity-based techniques perform best among the black-box approaches. Bertolino et al. [2015] exploit the notion of similarity to prioritize test cases in the context of access control systems. They conclude that considering the similarity outperforms the random order with respect to the early rate of fault detection. Zhang et al. [2017] propose a search-based approach that exploits the notion of similarity to identify a subset of scenarios for inspecting the requirements of a large-scale system. Their aim is to maximize the diversity of those selected scenarios. While we have the same goal with the aforementioned approaches that is representing by finding faults faster, we use the notion of similarity in terms of feature selection to increase the diversity; but for prioritizing products in product-line testing.

In addition, as surveyed by Yoo and Harman [2012], efforts have been made to prioritize test cases. For instance, Rothermel et al. [2001] describe several techniques for using test execution information to prioritize test cases in regression testing. This information includes code coverage and estimation on the ability of test cases to detect faults. Walcott et al. [2006] use a genetic algorithm to prioritize test cases which can be executed within a given time budget.

With respect to T-wise testing, Bryce and Colbourn [2007] propose a one-test at-a-time approach to cover more T-way interactions in earlier tests. In the context of product-line testing, it might be that some of these approaches can be applied to prioritize either the products or the test cases of a product line. Yoo et al. [2009] prioritize test cases by classifying the test cases into clusters. Then, they prioritize the clusters by utilizing the domain expert judgment. We follow a similar approach, but for products and do not consider domain expert judgment, which may be utilized in future work.

3.7 Summary

Testing product lines takes a considerable amount of time. Testers wish to increase the probability of detecting faults as soon as possible for the product line under test. Hence, several approaches have been proposed to prioritize products, such that testing products have a higher probability of containing faults earlier.

With configuration-based prioritization, we prioritize products based on similarity of their feature selection and deselection. Products are incrementally prioritized among the already tested and the remaining products from a given product set. We evaluate configuration-based prioritization using three product lines with real faults in their source code.

The results show that our approach can potentially accelerate the detection of faults. In addition, we use several sampling algorithms to generate a subset of all valid configurations. We evaluated configuration-based prioritization against random orders, interaction-based order, and default orders of the sampling algorithms CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979], and ICPL [Johansen et al., 2012a]. We performed our experiments on product lines of different size. The results show that the rate of early fault detection of configuration-based prioritization is significantly better than random orders. The default orders of existing sampling algorithms already show promising results, which can even be improved in many cases using configuration-based prioritization.

Comparing configuration-based prioritization to the interaction-based approach, the results show that the difference between the effectiveness of both approaches, with respect to the APFD, is not significant. However, our evaluation indicates that the interaction-based approach did not scale to models larger than 500 features (already for T=2). That is, we stopped computation after 24 hours, whereas configuration-based prioritization finished in all cases within 48 minutes.

In addition, we propose cluster-based product prioritization to classify products into different subsets such that products in each group share common features. This allows sampling these products and, thus, reducing the costs of testing, especially when the diversity among products is large. The results show that cluster-based product prioritization performs better, on average, than random orders.

In this chapter, we considered only problem-space information in terms of feature selection in product prioritization. The results show that prioritizing product can potentially enhance the product-line testing effectiveness. How including more information (i.e., solution-space information) in product prioritization influences the product-line testing effectiveness is discussed in the next chapter.

4. Delta-Oriented Similarity-Driven Product Prioritization

This chapter shares material with the VACE'17 paper "Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing" [Al-Hajjaji et al., 2017c].

In the previous chapter, we proposed configuration-based prioritization to prioritize products based on the similarity between them with respect to problem-space information (i.e., feature selection). In addition to configuration-based prioritization (cf. Chapter 3), other approaches considered prioritization in product-line testing to increase the effectiveness of testing [Devroey et al., 2014; Baller et al., 2014; Henard et al., 2014b; Johansen et al., 2012b; Sánchez et al., 2015]. However, these approaches focus mainly on prioritizing products based on the selection of features only. A potential limitation of all those approaches is that considering the feature selection only may not represent all actual differences between products. For instance, some features have more impact on solution-space artifacts than others. Hence, considering solution-space information for prioritization could improve the effectiveness of product-line testing.

In this chapter, we first introduce the variability modeling technique *delta modeling* [Clarke et al., 2015], which has been proposed to develop product lines (cf. Section 4.1) by applying operations, e.g., adding an element, to a designated core model. Second, we introduce delta-oriented prioritization, which allows us to reason about differences between solution-space artifacts, without computing and comparing complete products (cf. Section 4.2). Using delta modeling to prioritize products provides more detailed information about products than the feature selection, which may increase the rate of early fault detection. In case testers wish to consider also the feature selection in product prioritization, we third provide a combined approach of delta-oriented and configuration-based prioritization, where their impact can be adjusted using weighting

factors (cf. Section 4.3). Fourth, we evaluate the effectiveness of delta-oriented prioritization using an automotive product line and compare it against a default order of a sampling algorithm and random orders (cf. Section 4.5). Furthermore, we examine the influence of using two different distance measurements, namely Hamming and Jaccard distance, in product prioritization. The results of our product prioritization show an improvement in the effectiveness of product-line testing in terms of the fault detection rate. Finally, in Section 4.6, we discuss the related work that focuses on using solution-space information in product prioritization.

4.1 Delta Modeling

Various variability modeling techniques have been proposed to develop product lines [Schaefer et al., 2012], such as the transformational approach delta modeling [Clarke et al., 2015]. In our work, we focus on delta modeling, because detailed information about products and their differences can be extracted without need to compute and compare the corresponding products. Delta modeling has been proposed as an artifact-independent and modular variability modeling technique to develop software product lines [Clarke et al., 2015]. Based on a preselected *core product* $p_{core} \in P_{SPL}$, where P_{SPL} represents products of a product line SPL, differences are specified to transform the *core model* $m_{p_{core}}$ into product-specific models m_{p_i} for the remaining products $p_i \in P_{SPL}$ of the corresponding product line under consideration.

Selecting an appropriate *core product* may be guided by coverage criteria. For instance, the *core product* can be the smallest product (a.k.a. simple core product) or the largest possible product (a.k.a. complex core product) [Schaefer et al., 2010]. With the simple core product, we consider only, if applicable, the mandatory features and a minimal number of required alternative features, while with the complex core product, we consider, if possible, the largest set of features. In practice, selecting a *core product* is often influenced by project-specific considerations such as the existence of a legacy system to be used as a basis for the product line. In addition, other factors may affect the core product selection, such as focusing on the most desirable product by customers.

Each difference to the core product is defined by a change operation captured by means of a *delta* $\delta \in \Delta_{SPL}$. Thus, each delta δ comprises one particular operation, i.e., either an *addition* (add e) or a *removal* (remove e) of a model element e. In particular, we abstract the differences between products from modifications of elements. By Δ_{SPL}, we refer to the set of all possible deltas for the current product line that are extracted from the delta model of a product line $\Delta_{M_{SPL}}$. Hence, for each product $p_i \in P_{SPL}$, a delta set $\Delta_{p_i} \subseteq \Delta_{SPL}$ exists solely comprising the transformations to obtain the corresponding model m_{p_i} by applying all deltas $\delta \in \Delta_{p_i}$ subsequently to the core model. The product-specific delta sets are either predefined or can be determined based on application conditions, i.e., logical expressions using features $f \in F_{SPL}$ as variables, specified for each delta $\delta \in \Delta_{SPL}$. By giving a feature configuration F_p for product p, all application conditions are evaluated and those deltas are selected in the corresponding delta set Δ_p for which the evaluation results in `true`. The concept of delta modeling [Clarke et al.,

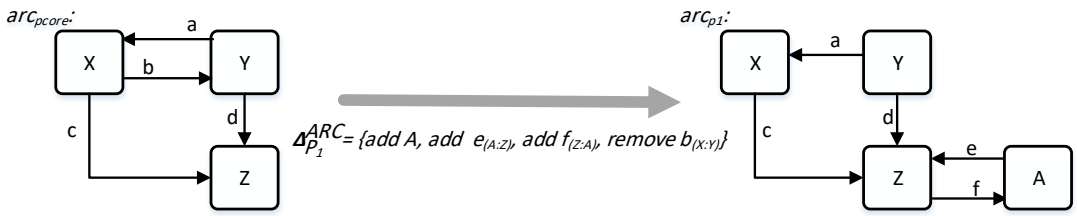


Figure 4.1: The principle of delta-oriented software architectures

2015] is instantiated for various types of artifacts such as JAVA code [Schaefer et al., 2010], finite state machines [Lochau et al., 2012], or architectures [Lochau et al., 2014]. In this chapter, we propose a prioritization technique independent from a concrete delta modeling instantiation. Nevertheless, we use delta-oriented architectures for illustration and evaluation.

Software architectures define the high-level structure of a software system [Szyperski, 2002] by describing the overall system *components* $C = \{c_1, \ldots, c_m\}$ and specifying the communication between them via *connectors* $CON = \{con_1, \ldots, con_l\}$ transmitting *signals* $\Pi = \{\pi_1, ..., \pi_k\}$. In the context of delta-oriented product lines, for each product $p_i \in P_{SPL}$ a corresponding architecture model arc_{p_i} exists. Thus, based on a *core architecture model* $arc_{p_{core}}$, *architecture deltas* $\delta \in \Delta^{ARC}_{SPL}$ are defined and grouped in product-specific delta sets $\Delta^{ARC}_{p_i}$ to transform the core into an architecture model arc_{p_i}. The operations captured by a delta add/remove components, connectors, or signals.

Example 4.1. *In Figure 4.1, a sample core architecture model* $arc_{p_{core}}$ *is shown on the left-hand side comprising the three components* $C = \{X, Y, Z\}$ *communicating via the four connectors* $CON = \{a_{(Y:X)}, b_{(X:Y)}, c_{(X:Z)}, d_{(Y:Z)}\}$ *(i.e.,* $a_{(Y:X)}$ *represents the connector a between components* Y *and* X*) by transmitting the four signals* $\Pi = \{a, b, c, d\}$*. To transform* $arc_{p_{core}}$ *into* arc_{p_1} *on the right-hand side, we apply the delta set* $\Delta^{ARC}_{p_1}$*, i.e., we remove one connector, add one component as well as two connectors.*

Delta modeling [Clarke et al., 2015] has been further adapted to improve product-line testing by reducing and prioritizing the number of executed test cases for components as well as integration testing [Lochau et al., 2012, 2014; Lachmann et al., 2015, 2016; Lity et al., 2016]. In this thesis, we exploit the concept of delta modeling to improve product-line testing effectiveness by prioritizing products under test. Therefore, we identify the commonalities and the differences between products by means of deltas to reason about their similarity on the solution-space level, e.g., between their product-specific architectures. Similarity-based testing is a technique used to select a subset of test cases, which aims to increase the early rate of fault detection [Hemmati et al., 2013]. Achieving this aim can be done by maximizing the diversity among test cases. In the context of product-line testing, numerous approaches apply similarity-based testing to sample and prioritize products, with respect to feature selections [Sánchez et al., 2015;

Henard et al., 2014b, 2013c]. However, in this chapter, we prioritize products based on the similarity between them with respect to deltas. In particular, we present delta-oriented prioritization and the combined approach of configuration-based (cf. Chapter 3) and delta-oriented prioritization in the next sections.

4.2 Delta-Oriented Prioritization

The input to our approach is a set of products which can be all valid products if the product line is small, sample products that are generated using sampling algorithms, or products given by domain experts. In addition to products, the delta model of the software product line is required. With delta-oriented prioritization, we perform three main steps to prioritize products to be tested by incorporating their differences by means of deltas to reason about their similarity (cf. Algorithm 4.1). Therefore,

1. we select the first product which has the highest capability to detect the most faults at the beginning of the testing process,
2. we select the second product which is most dissimilar compared to the first product by means of deltas to be applied, and
3. the remaining products are incrementally selected by taking the delta-oriented similarity to all already tested products into account.

The aforementioned steps are mainly based on Algorithm 3.1 on Page 22, Algorithm 3.2 on Page 24, and Algorithm 3.3 on Page 26 of the configuration-based prioritization (cf. Section 3.1) with the following differences and improvements. Regarding the differences, in addition to the feature model, which is also needed in configuration-based prioritization, the delta model is required as input of the delta-oriented prioritization approach. With respect to the improvements, with configuration-based prioritization (cf. Section 3.1), we calculate the distances between all products in the initialization phase. In the enhanced algorithm, the distances between products are calculated only when it is needed, as it could be the case that testing time is finished before testing all products. Once we calculate the needed distances between each untested product and the new tested products, we store them to be used in follow-up steps. The distances between the already tested products can be removed, as there is no need anymore to store them. Postponing the distance calculation has the advantage that it improves scalability and the performance of the algorithm. With respect to the scalability, only the needed distances are stored in the memory, which means the approach can scale to large product lines in terms of memory consumption. Regarding the performance, postponing the distance calculation will accelerate the process of finding the distance between particular products, especially at the beginning, because we iterate distances of a small set of products rather than the whole set of products. In the following, we describe the three aforementioned steps in detail.

Algorithm 4.1 Delta-Oriented Prioritization.

Require: $\mathcal{FM}$ ▷ feature model
P_{SPL} ▷ a set of products
$\Delta_{M_{SPL}}$ ▷ a delta model of a product line
Return: P_{tested} ▷ list of tested products

```
1: function PRIORITIZATION(P_SPL, Δ_MSPL)
2:     Δ_SPL ← GETDELTAS(Δ_MSPL)
3:     P_tested ← EMPTYLIST
4:     AllDistances[ ] ← EMPTYARRAY
5:     p_core ← GETCOMPLEXCOREPRODUCT(P_SPL)
6:     P_tested ← P_tested ∪ {p_core}
7:     P_SPL ← P_SPL \ {p_core}
8:     TESTPRODUCT(p_core)
9:     while |P_SPL|>0 do
10:        p_i = SELECTFURTHERCONFIG(Δ_SPL, P_tested, AllDistances)   ▷ Algorithm 4.2
                                                                      select further product
                                                                      to be tested
11:        P_tested ← P_tested ∪ {p_i}
12:        P_SPL ← P_SPL \ {p_i}
13:        TESTPRODUCT(p_i)
14:    end while
15:    return P_tested
16: end function
```

4.2.1 Choosing First Product to Test

As the overall goal of our product prioritization is to detect faults as early as possible during product-line testing, we follow a strategy used for the analysis of the Linux kernel, where a specific product called *allyesconfig* is selected to be analyzed or tested first [Dietrich et al., 2012]. The *allyesconfig* is one of the largest products in the number of selected features. Testing a large product first has the advantages that many faults are caused by the selection of a single feature or the selection of two or more features can be detected [Abal et al., 2014; Medeiros et al., 2016]. In contrast, we focus on the detection of faults in solution-space artifacts such as architectures and, therefore, reason about model size when selecting the first product to be tested. The first product builds the basis for the selection of the next products to be tested by incorporating their similarity to each other by means of model differences.

In delta modeling [Clarke et al., 2015], we observe a similar scenario, where the core product builds the basis for the specification of deltas to create the remaining product-specific models. We combine both scenarios by (1) choosing a *complex core*, i.e., a certain product $p_{core} \in P_{SPL}$ with the largest set $ME_{p_{core}}$ of model elements comprised in the respective product-specific model $m_{p_{core}}$ and (2) selecting the core as first product to be tested (Algorithm 4.1, Line 5). For architectures, the set ME_p of model elements

Algorithm 4.2 Select Further Configuration.

Require: P_{SPL} ▷ a set of products
Δ_{SPL} ▷ a delta set of a product line
P_{tested} ▷ a list of tested products
$AllDistances[\,]$ ▷ The calculated needed distances between products
Return: $NextConfig$ ▷ product with the largest distance to the tested ones

```
1: function SelectConfigurations(Δ_SPL, Δ_P_core, AllDistances)
2:     NextConfigDistance ← 0
3:     for each p_i ∈ P_SPL do
4:         TempDistance ← 1
5:         for each p_j ∈ P_tested do
6:             AllDistance[i, j] ← getDistances(Δ_SPL, Δ_p_i, Δ_p_j))
7:             if Distance_p_i p_j < TempDistance then
8:                 TempDistance ← Distance_p_i p_j
9:                 P_New_Temp ← p_i
10:            end if
11:        end for
12:        if TempDistance > NextConfigDistance then
13:            NextConfigDistance ← TempDistance
14:            NextConfig ← P_New_Temp
15:        end if
16:    end for
17:    return NextConfig
18: end function
```

is defined by the union of components C_p and connectors CON_p contained in the architecture arc_p. To find the complex core, we assume that either (1) the model size corresponds to the feature configuration size by means of selected features and, thus, the complex core has also the largest feature configuration, or (2) the developer/test engineer has this information based on domain and system knowledge. If there exist more than one product comprising the maximum number of elements, we have to choose one of these products as a complex core for delta specification and first product to be tested. For testing a product, we apply standard testing techniques from single systems engineering, e.g., integration testing in the context of software architectures [Bertolino et al., 1997].

Example 4.2. *For instance, assume we want to test the five products* $P_{SPL} = \{p_1, \ldots, p_5\}$ *listed in Table 2.1 on Page 15. We define* p_2 *as complex core selected to be tested first as it contains the maximum number of selected features and we assume that it has the largest set* $ME_{p_{core}}$ *of model elements.*

4.2.2 Choosing Second Product to Test

After testing the core, we select the next product to be tested by incorporating the similarity compared to the core by means of model differences specified by deltas (Al-

gorithm 4.2). Selecting the second product to be tested is a special case of Algorithm 4.2, as the inner loop (Lines 5–11) will be iterated once, because we only have one tested product (core product p_{core}) in the list P_{tested}. By selecting the most dissimilar product, we facilitate the coverage of solution-space artifacts to be tested and, therefore, support early fault detection. In order to measure the similarity between products, we require a suitable distance metric and meaningful properties on which the chosen metric is applicable. Devroey et al. [2016] investigated the impact of using different types of similarity measurements in test case prioritization such as Hamming, Jaccard, Dice, Anti-dice, and Levenshtein distance. They report that Hamming and Jaccard distances outperform the other distance types. However, they do not conclude whether using Hamming outperforms using Jaccard distance or vice versa. Hence, we build on and complement their work by investigating only the impact of Hamming and Jaccard distances in product prioritization. Furthermore, as we focus on solution-space artifacts such as architectures, we take the product-specific models and their differences by means of model elements into account to determine their similarity.

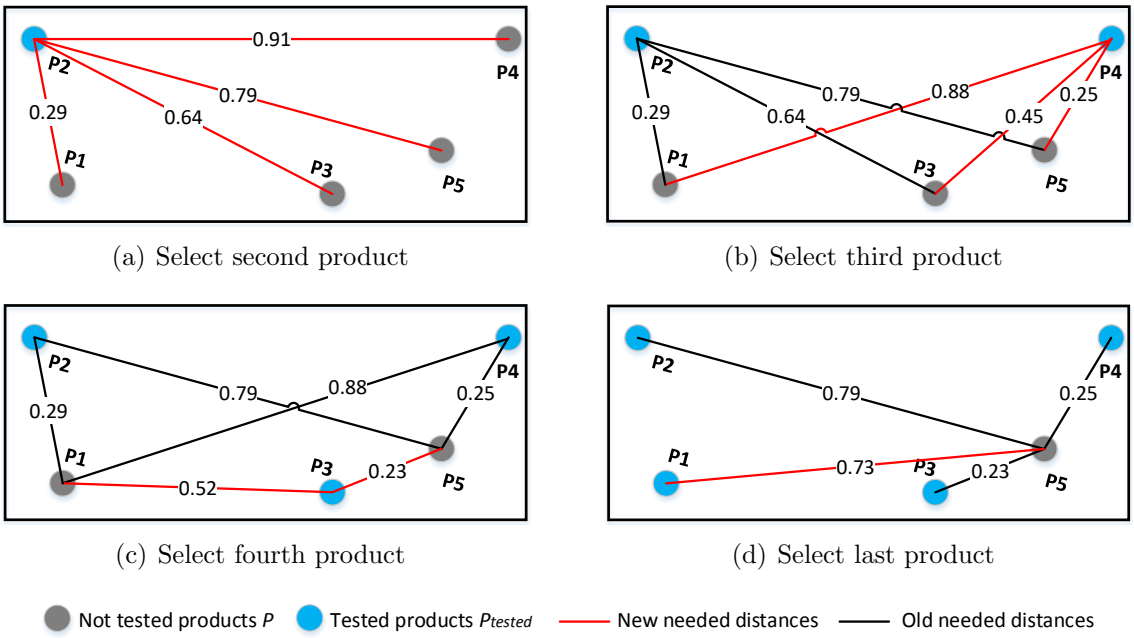
Following a naive approach, we have to create every product-specific model to determine the differences for the distance computation. Based on delta modeling [Clarke et al., 2015], we already have the explicit knowledge about those differences specified by deltas and, thus, are able to determine the distance between two products by comparing their delta sets without generating each individual model. Each product-specific delta set comprises the particular deltas, i.e., change operations by means of additions and removals of elements, required for transforming the core into the corresponding model. The more deltas and, thus, change operations are common in both delta sets, the more similar are the respective products. In addition, the usage of delta modeling allows for a generalized definition of delta similarity, because it is independent of artifacts. Therefore, the following generalized delta similarity functions build the basis for the application of our approach in the context of the different delta modeling instantiation such as delta-oriented architectures.

The Hamming distance is a well-known distance measurement [Hemmati et al., 2013], which we used as a similarity measurement in the previous chapter. In order to handle product-specific delta sets for distance computation and, thus, for the comparison of two products p_i and p_j, we compare their corresponding delta sets Δ_{p_i} and Δ_{p_j}, and further the sets of deltas not applicable for both products. We define the delta similarity function $deltaDist_H$ based on the Hamming distance as follows

$$deltaDist_H(\Delta_{p_i}, \Delta_{p_j}, \Delta_{SPL}) = \quad 1 - \frac{|\Delta_{p_i} \cap \Delta_{p_j}| + |(\Delta_{SPL} \setminus \Delta_{p_i}) \cap (\Delta_{SPL} \setminus \Delta_{p_j})|}{|\Delta_{SPL}|}, \quad (4.1)$$

where $|\Delta_{p_i} \cap \Delta_{p_j}|$ denotes the number of common deltas applicable for p_i as well as p_j, and $|(\Delta_{SPL} \setminus \Delta_{p_i}) \cap (\Delta_{SPL} \setminus \Delta_{p_j})|$ represents the number of common deltas which are not applicable to obtain their product-specific models.

The Jaccard distance measurement was already proposed to measure the similarity between products and test cases [Henard et al., 2014b; Devroey et al., 2016]. In contrast



(a) Select second product

(b) Select third product

(c) Select fourth product

(d) Select last product

Figure 4.2: Delta-oriented prioritization with distance minimum

to the Hamming distance, it defines a different comparison of products by incorporating their product-specific delta sets. For the Jaccard distance, the number of common deltas between two delta sets is compared to the number of all deltas captured in both delta sets. Thus, the delta similarity function $deltaDist_J$ is defined based on the Jaccard distance as follows

$$deltaDist_J(\Delta_{p_i}, \Delta_{p_j}, \Delta_{SPL}) = 1 - \frac{|\Delta_{p_i} \cap \Delta_{p_j}|}{|\Delta_{p_i} \cup \Delta_{p_j}|}, \tag{4.2}$$

where $|\Delta_{p_i} \cap \Delta_{p_j}|$ denotes the number of common deltas for p_i and p_j, and $|\Delta_{p_i} \cup \Delta_{p_j}|$ represents the number of all deltas specified for both products.

In the context of delta architectures, both functions compare deltas specifying the addition and removal of components and connectors. For both functions, their result ranges between 0 and 1, where a value close to 0 indicates that compared products have similar delta sets and, thus, similar product models are created by applying the respective delta sets, whereas a value close to 1 indicates different products.

Example 4.3. *After selecting p_2 as complex core product to be tested first, we follow an incremental process by determining the distances to the four untested products p_1, p_3, p_4, and p_5 as shown in Figure 4.2(a), e.g., based on the delta similarity function $deltaDist_H$ using the Hamming distance measure. Then, we select the most dissimilar product to p_2 in terms of deltas. The distances between product p_2 and products p_1, p_3, p_4, and p_5 are 0.29, 0.64, 0.91, and 0.79, respectively. As p_4 has the largest distance (0.91) to p_2, we select p_4 as second product to be tested.*

4.2.3 Choosing Further Products to Test

For the selection of the third and further products to be tested from the set of remaining products $P_{SPL} \setminus P_{tested}$, we must take the similarity to all already tested products P_{tested} into account (cf. Algorithm 4.2). By incorporating the distances to all previously tested products, the selection of the next product ensures the fast coverage of solution-space artifacts such as architectures and their elements and, therefore, facilitates the early fault detection.

In Section 3.1.3, we already introduced two strategies to calculate distances between more than two products in product-line testing, namely maximum over (1) distance summation [Henard et al., 2014b] (cf. Equation 3.2) and (2) distance minimum (cf. Equation 3.3 and Equation 3.4). In this chapter, we will also investigate the impact of using aforementioned strategies when the similarity between products is calculated.

After testing the third product that is less similar to the previous two products, we repeat the product selection until all products are tested or the provided testing resources are exhausted.

Example 4.4. *Depending on the chosen strategy, the order in which products are selected differs. For instance, consider again our running example depicted in Figure 4.2. After selecting and testing product p_4 (cf. Example 4.3), we first update the distances between all untested and tested products shown in Figure 4.2(b). For the summation distance, we select p_1 as next product to be tested, because the sum of its distances (1.17) compared to p_5 (1.09) and p_3 (1.04) is larger. After testing p_1, we update the distances and select p_5 (cf. Figure 4.2(c)). In the end, we select p_3 as last product to be tested as shown in Figure 4.2(d) and obtain p_2, p_4, p_1, p_5, p_3 as testing order. In contrast, for minimum distance, we would select p_3 as third product under test as the pair (p_3, p_4) has the maximum of minimum distances (0.45) compared to (p_1, p_2) (0.29) and (p_5, p_4) (0.25). The corresponding testing order results in p_2, p_4, p_3, p_1, and p_5.*

Moreover, we investigate In this chapter whether the distance functions (Hamming and Jaccard distance) and the way the distance between more than two products is computed (summation and minimum distance) influence the results.

4.3 Combining Configuration-Based and Delta-Oriented Prioritization

In Section 3.1, we show how products can be prioritized using configuration-based product prioritization. Furthermore, we measure the similarity between products using the Hamming distance functions as follows:

$$confDist(F_{p_i}, F_{p_j}, F_{SPL}) = \quad 1 - \frac{|F_{p_i} \cap F_{p_j}| + |(F_{SPL} \setminus F_{p_i}) \cap (F_{SPL} \setminus F_{p_j})|}{|F_{SPL}|}, \qquad (4.3)$$

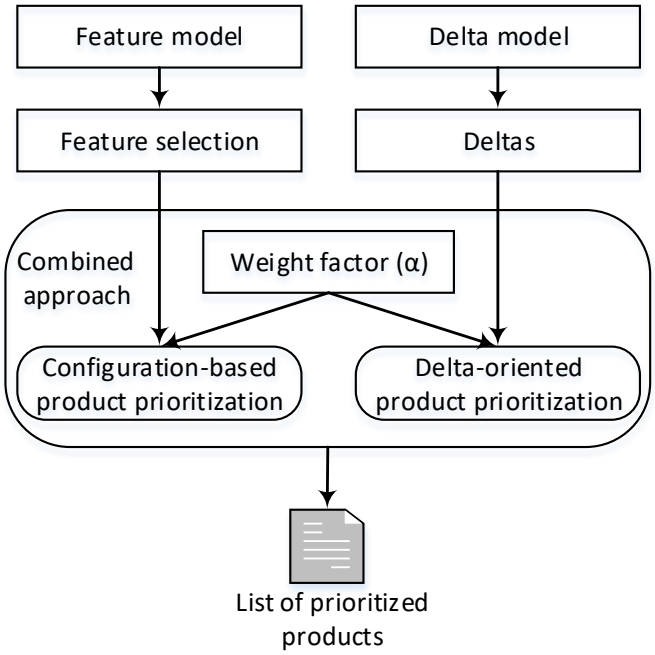


Figure 4.3: Overview of the combined approach where α represent the weighting factor of the involved approaches

where $|F_{p_i} \cap F_{p_j}|$ is the number of included features in products p_i and p_j, $|(F_{SPL} \setminus F_{p_i}) \cap (F_{SPL} \setminus F_{p_j})|$ is the number of not included features in products p_i and p_j, and F_{SPL} is a set of features of the SPL under test.

In Section 4.2, we propose delta-oriented prioritization to prioritize products based on the similarity between them with respect to deltas. In this section, we present the combined approach of delta-oriented and configuration-based prioritization, where the impact of each can be adjusted using a weighting factor.

Combined Approach

As illustrated in Figure 4.3, the inputs to the combined approaches are the feature model and delta model for the current product line under test, where the feature selections and deltas are extracted, respectively. As mentioned above, the feature selections and deltas are the inputs of configuration-based and delta-oriented prioritization. We control the impact of each approach by introducing α as a weighting factor. Therefore, the value of α adjusts the weight of the delta-oriented prioritization, whereas the $1 - \alpha$ represents the weight for the configuration-based prioritization as counterpart. In particular, the α value adjusts the impact of the feature selections and deltas when the total distance is calculated. The value of α can be adjusted based on the availability and quality of

the given information. The total combined distance for product prioritization is defined as follows:

$$totalDist(F_{p_i}, F_{p_j}, F_{SPL}, \Delta_{p_i}, \Delta_{p_j}, \Delta_{SPL}) = \frac{\alpha \cdot deltaDist_{H/J} + (1-\alpha) \cdot confDist}{2} \quad (4.4)$$

Example 4.5. *Consider our running example and the testing orders of Example 3.4 and Example 4.4 again, where the testing orders are determined based on* $\alpha = 0.0$ *and* $\alpha = 1.0$*, respectively. For* $\alpha = 0.0$*, we obtain* p_2*,* p_3*,* p_1*,* p_4*, and* p_5 *as testing order, while for* $\alpha = 1.0$*, we obtain the following testing order* p_2*,* p_4*,* p_3*,* p_1*, and* p_5*. In contrast, for* $\alpha = 0.5$*, we derive* p_2*,* p_3*,* p_5*,* p_4*, and* p_1 *as testing order showing the impact of the variation of* α*.*

An alternative way to combine the configuration-based and delta-oriented prioritization approaches is that we ignore the α value and, instead, we consider the product with the maximum distance of both approaches. A potential limitation is that the used information that causes the maximum distance may not represent the actual differences between products. Note that, in the evaluation, we use different values of α to assess our approach (cf. Section 4.5).

4.4 The Implementation of Delta-Oriented Prioritization

We built our implementation to prioritize products based on an existing framework, where we reuse the modeling part of it [Lochau et al., 2014]. In particular, the tool is an Eclipse plug-in that consists of Deltarx editor that is used to model the core architecture as well as the deltas. Using the editor, the user is able to define the core architecture (cf. Appendix, Section A.3) as well as a set of deltas based on the Deltarx grammar [Lity et al., 2013]. This Deltarx grammar is defined using Eclipse Modeling Framework [1] and Xtext Framework[2] as they are compatible with Eclipse.

Each architecture model delta is recognized by a unique name and mapped to a feature/partial configuration (i.e., a combination of two or more features). These partial configurations are part of a set of configurations generated using the sampling algorithm *MoSo-PoLiTe* [Oster et al., 2011c]. This mapping is achieved by a declaration of a condition as a Boolean statement using the keyword *when*. In case delta dependencies are needed to be applied in advance (i.e., some deltas are needed to be applied before others), those dependencies are indicated using the keyword *after*. As mentioned in Section 4.1, generating a new product is achieved by transforming of the core, i.e., addition and/or removal of the architecture elements using the listed keywords in Table 4.1. The process of these plug-ins works as follows. First, the architecture of the core product p_{core} is defined (cf. Appendix Section A.3). Second, the architecture model deltas are

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.eclipse.org/Xtext/

	Add	Remove
Component	addcomponent	removecomponent
Connector	addconnector	removeconnector
Signal	addsignal	removesignal

Table 4.1: Overview of Deltarx Transformation Keywords

defined. Then, once the configuration is selected, the architecture model deltas that are required to generate the product are extracted. We use those deltas as inputs to our implemented approach to differentiate between products. In the following section, we evaluate the effectiveness of the delta-oriented prioritization as well as the combined approach.

4.5 Evaluation of Delta-Oriented Prioritization

To show the effectiveness of our approach, we compare it against random orders as well as the default order of the sampling algorithm *MoSo-PoLiTe* [Oster et al., 2011c], where an implicit order as part of its output is given by the positioning of products within the output data structure. While the default order of a set of sampling algorithms, such as CASA, Chvatal, and ICPL have been already evaluated (cf. Chapter 3), the default order of the sampling algorithm *MoSo-PoLiTe* has not evaluated yet. We did not consider the sampling algorithm *MoSo-PoLiTe* in the evaluation of Chapter 3, because we could not have access to the tool. Even with the evaluation of delta-oriented prioritization, we do have the access to the configurations of the subject product line generated by *MoSo-PoLiTe*, not the tool itself. In particular, *MoSo-PoLiTe* is the main algorithm used to sample the subject product line. In addition, we assess whether adding more solution-space information enhances the effectiveness of product-line testing. Furthermore, we evaluate whether using different distance functions influences the results. In particular, we answer the following research questions:

RQ1 How does delta-oriented prioritization using delta modeling *perform* compared to random orders and the default order of the sampling algorithm *MoSo-PoLiTe* [Oster et al., 2011c]?

RQ2 How does the combination of delta-oriented and configuration-based prioritization in product prioritization *influence* the effectiveness of product-line testing?

RQ3 How does the way the distance between products is measured *influence* the effectiveness of product-line testing?

In the following, we introduce the subject product line and the fault injection that we used in our evaluation. Then, we present and discuss the reported results.

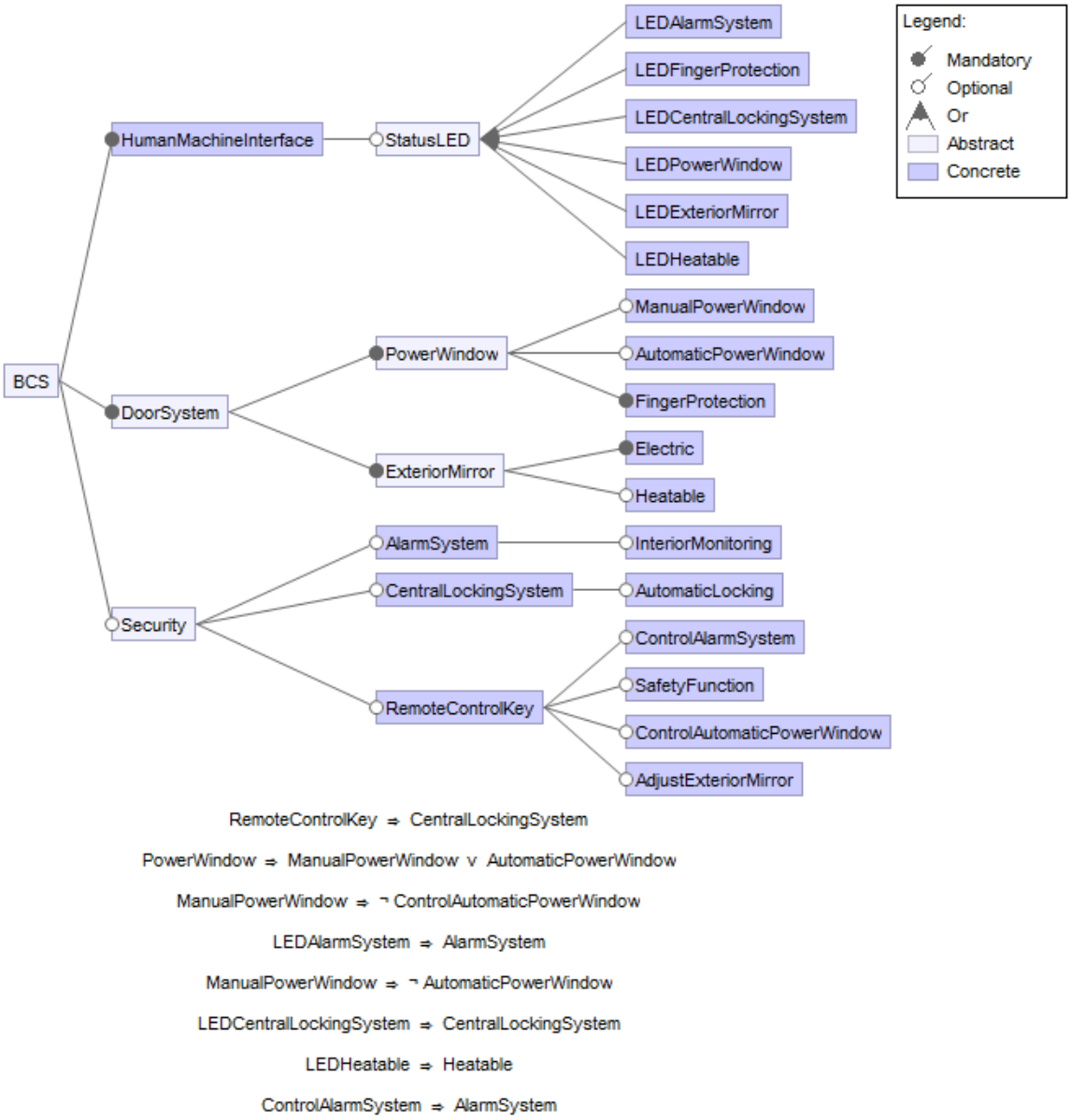


Figure 4.4: The feature model of the Body Comfort System

4.5.1 Subject Product Line: Body Comfort System (BCS)

The amount of software embedded in the automotive domain is steadily increasing. An example of this software is the implementation of the driver assistance systems in cars. To assure that these systems work as expected, software testing becomes a critical task in the automotive development. Furthermore, the variability of such systems is needed to be taken into account during testing, as it makes the system more complex [Oster et al., 2011a].

In our work, we evaluate our approach by means of a product line from the automotive domain representing a Body Comfort System (BCS) [Lity et al., 2013]. Oster et al.

```
1 DAutomaticPW when 'Automatic Power Window' {
2 removeconnector{
3 fp1
4 fp2
5 pw1
6
7 #endif /* FEAT_VISUAL */
```

Listing 4.1: Delta DAutomaticPW for the Feature AutomaticPowerWindow [Lity et al., 2013]

[2011b] adapted the original version of the BCS to a product line. As illustrated in Figure 4.4, the BCS comprises a number of mandatory and optional features, such as a *human machine interface* that serves as a point of interaction with a driver, an *alarm system* with the option of having *interior monitoring*, a *central locking system* with the optional of *automatic locking*, a *manual/automatic power window*, and a *remote control key* enables the locking and unlocking of the car as well as the controlling of the window movement. The BCS product line comprises a total of 11,616 possible products. Applying pairwise sampling using the algorithm *MoSo-PoLiTe*, the number of generated products is reduced to 17 products, where we add the core as product 18 [Lity et al., 2013].

For evaluation, we focus on the delta-oriented architectures specified for the BCS product line [Lity et al., 2013]. The range size of all the designed architecture models is between 4 and 19 components. The average number of connectors in each architecture is 72. These connectors transfer an average of 60 different signals between components. We refer to Appendix, Section A.3 Listing A.1 for the architecture definition of the core product of the BCS. In Listing 4.2, we show an example of the deltas that is required to be applied in order to add the feature *AutomaticPowerWindow* to a product. It looks that in order to add feature *AutomaticPowerWindow*, we need to remove eight connectors, one component, and two signals from the core product. Furthermore, we need to add nine connectors, one component, and three signals. For all deltas of the BCS product line, we refer the reader to the Appendix, Section A.3 Listing A.2.

4.5.2 Fault Injection

To measure the effectiveness of the proposed approach, we use seeded faults using a well-known method to assess the fault detection of a test suite [Mathur, 2008]. Following the same concept of our work in Section 3.3.3, but considering the delta modeling instead of feature selections, we randomly select architecture model elements and mark them as containing faults. We assume that if the products contain these elements, the faults will be detected. Abal et al. [2014] report that some faults can be triggered because some features are *not* selected. Thereupon, we consider the involved and the non-involved model elements when we seed faults. We consider two types of faults: single faulty elements to simulate faults in a single element and pairwise combinations of faulty elements to simulate faults caused by the interaction between elements. The reason for

```
1  DAutomaticPW when 'Automatic Power Window' {
2  removeconnector{
3  fp1
4  fp2
5  pw1
6  pw2
7  hmi5
8  hmi6
9  env13
10 env14
11 }
12 removecomponent {
13 ManPW
14 }
15 removesignal {
16 pw_mv_dn
17 pw_mv_up
18 }
19 addsignal {
20 pw_auto_mv_up boolean
21 pw_auto_mv_dn boolean
22 pw_auto_mv_stop boolean
23 }
24 addcomponent{
25 AutoPW{
26 }
27 }
28 addconnector{
29 fpautopw1(FP,fp_on,fp_on,AutoPW)
30 fpautopw2(FP,fp_off,fp_off,AutoPW)
31 hmiautopw1(HMI,pw_but_up,pw_but_up,AutoPW)
32 hmiautopw2(HMI,pw_but_dn,pw_but_dn,AutoPW)
33 autopwenv1(AutoPW,pw_auto_mv_up,pw_auto_mv_up,ENV)
34 autopwenv2(AutoPW,pw_auto_mv_dn,pw_auto_mv_dn,ENV)
35 autopwenv3(AutoPW,pw_auto_mv_stop,pw_auto_mv_stop,ENV)
36 envautopw1(ENV,pw_pos_up,pw_pos_up,AutoPW)
37 envautopw2(ENV,pw_pos_dn,pw_pos_dn,AutoPW)
38 }
39 }
```

Listing 4.2: Delta DAutomaticPW for the Feature AutomaticPowerWindow [Lity et al., 2013]

considering only up-to pairwise combinations is the nature of the subject product line as it consists of only 27 features. Hence, we do not expect to find faults caused by a high degree of combinations.

In the following steps, we show an example of how the pairwise faults are seeded. At first, we select two elements (i.e., components) randomly. Assume that we select components c_3 and c_5. The decision of involving the component in a faulty combination or not ($\neg$) is decided randomly. In our case, assume that component c_3 is selected to be

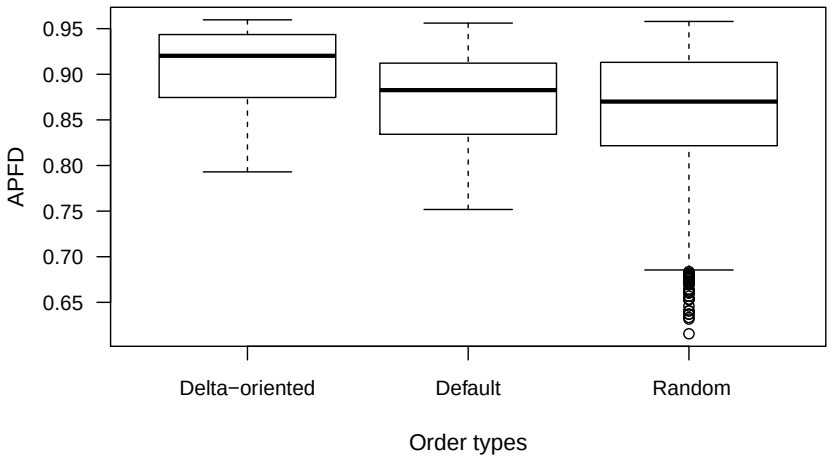


Figure 4.5: The APFD distribution for delta-oriented prioritization, default order of sampling algorithm *MoSo-PoLiTe [Oster et al., 2011c]* (Default), and random orders (Random)

	Delta-oriented	Default	Random
Avg. APFD	0.907	0.871	0.863

Table 4.2: The average APFD for delta-oriented prioritization, default order of sampling algorithm *MoSo-PoLiTe [Oster et al., 2011c]* (Default), and random orders (Random)

involved and component c_5 is selected to be not involved. The combination of the two components is $c_3 \wedge \neg c_5$. We ensure the validity of the generated combination with respect to the feature model and delta model. We generate 200 sets of faults (100 sets for single-wise interactions and 100 sets for pairwise interactions). In each set, we randomly select 10% of the faulty elements. We also considered different percentage of faulty elements; however, the results show no significant difference. In our evaluation, we report the results of considering 10% of the faulty elements. We use average percentage of faults detected (APFD) as a metric to measure the effectiveness of our approach. APFD is a metric developed by Elbaum et al. [2000] to evaluate how fast faults are detected during testing. For more details about the APFD, we refer the reader to Section 3.3.1.

4.5.3 Results and Discussion

To answer **RQ1**, we used Hamming distance to measure the similarity between products. For computing the distance between more than two products, we considered the minimum distance strategy.

The boxplots in Figure 4.5 show the APFD distribution for each approach over 200 sets of faults. The X-axis represents the different type of orders (delta-oriented product prioritization, random, and default order) and the Y-axis represents the APFD value. To reduce the impact of the random order, the boxplot shows the distribution of APFD values of 100 random orders. Each random order is executed on 200 different sets of faults.

From Figure 4.5, it is obvious that delta-oriented prioritization outperforms the random ordering as well as the default order of the sampling algorithm *MoSo-PoLiTe* [Oster et al., 2011c]. In Table 4.2, we show the average APFD values for each approach over 200 sets of faults, where we observe that delta-oriented approach has a higher APFD value (0.907) than the random ordering (0.863) and the default order of *MoSo-PoLiTe* [Oster et al., 2011c] (0.871). To test whether the difference between our approach and the default order of *MoSo-PoLiTe* as well as the random orders is significant, we use the Mann-Whitney U test (cf. Section 3.3.1). In our results, we observe that the differences between our approach and the default order of *MoSo-PoLiTe* as well as the random orders are significant with p-value 0.0 for both approaches. Hence, regarding **RQ1**, delta-oriented prioritization outperforms the random ordering as well as the default order of the sampling algorithm *MoSo-PoLiTe* [Oster et al., 2011c]. Figure 4.5 also shows that the default order of *MoSo-PoLiTe* is slightly better than the random orders. While, in some cases, it shows that random orders are better than the *MoSo-PoLiTe* default order, we observed that in random ordering numerous worse outliers appear, which is not the case in the default order.

Regarding **RQ2**, in order to show whether the combination of configuration-based and delta-oriented prioritization influences the results, we consider different cases which represent different values of α. The range value of α is between 0.0 and 1.0. If the value of α is 0.0, we just apply configuration-based prioritization and if the α value is 1.0, we apply delta-oriented prioritization. In the other cases where $0.0 < \alpha < 1.0$, we apply the combined product prioritization. These different values adjust the impact of deltas and feature selections (*cf. Equation 4.4*). In particular, we consider the three cases that represent three values as follows.

- Case 1: $\alpha = 0.0$, $1 - \alpha = 1.0$. (configuration-based prioritization)
- Case 2: $\alpha = 0.50$, $1 - \alpha = 0.50$.
- Case 3: $\alpha = 1.0$, $1 - \alpha = 0.0$. (delta-oriented prioritization)

The proposed α values in this chapter are for evaluation purposes. However, the α value can be adjusted by testers based on the availability and the quality of the provided information of product lines.

Figure 4.6 shows the distribution of APFD values, where the X-axis denotes different values of α, which represent different weights for the delta-oriented prioritization,

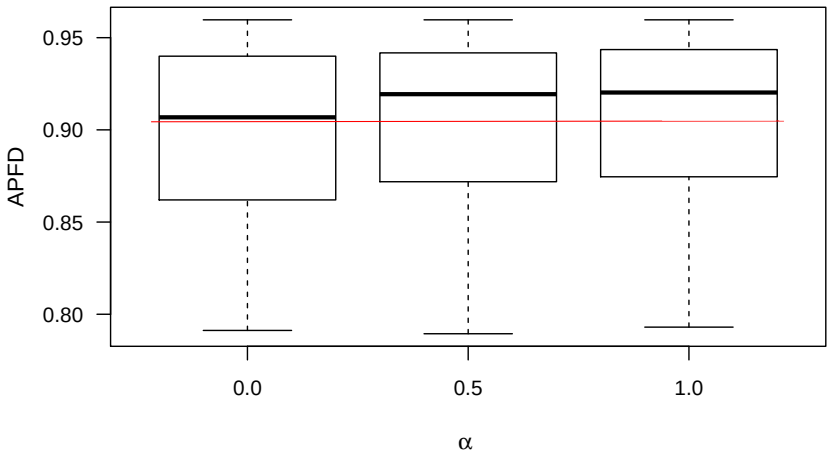


Figure 4.6: The APFD distribution for delta-oriented prioritization with considering different weights of delta-oriented prioritization represented by the value of α

	$\alpha = 0.0$	$\alpha = 0.5$	$\alpha = 1.0$
Avg. APFD	0.900	0.905	0.907

Table 4.3: The average APFD distribution for delta-oriented prioritization with considering different weights of delta-oriented prioritization represented by the value of α

and the Y-axis denotes the APFD values. From Figure 4.6, we observe that including solution-space information (delta modeling) to problem-space information (feature selection) improves the effectiveness of product-line testing. Furthermore, the results in Table 4.3 show that the average APFD value over 200 sets of faults when $\alpha = 0.0$ is 0.900, while it is 0.907 when $\alpha = 1.0$. Using the Mann-Whitney U test, we found that the difference between the orders of the generated products, where $\alpha = 0.0$ and $\alpha = 1.0$, is significant with p-value 0.04. We envision that adding more solution-space information will enhance the product-line testing effectiveness, because it can give us information about the solution-space impact and the functionality of each feature, which can be used to weight each feature during product prioritization. The results indicate that including other solution-space information (i.e., source code), may enhance the SPL testing effectiveness compared to configuration-based prioritization. However, adding problem-space information does not seem to give advantages.

In order to answer **RQ3**, we compared the Jaccard distance to the Hamming distance. For each distance type, we used two different ways of calculating the distance, namely maximum over *distance minimum* and *distance summation*. In Figure 4.7 and Ta-

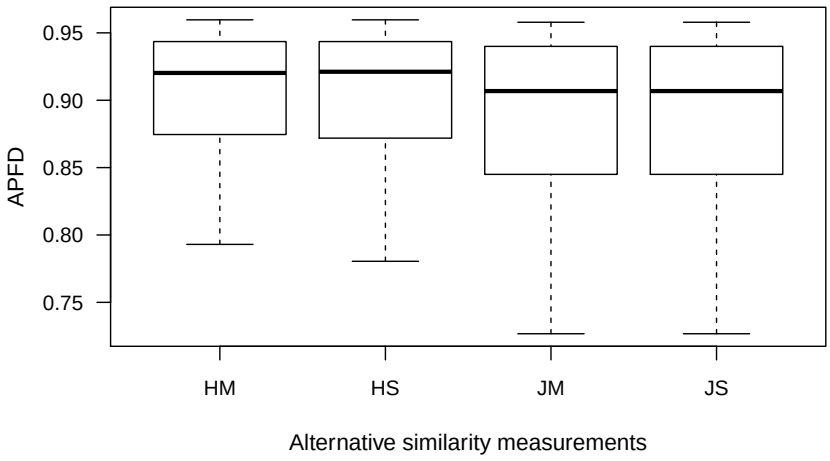


Figure 4.7: The APFD distribution for delta-oriented prioritization by considering combinations of different distance types, Hamming distance (H) and Jaccard distance (J) with minimum distance (M) and summation distance (S)

	HM	HS	JM	JS
Avg. APFD	0.907	0.906	0.891	0.891

Table 4.4: The average APFD for delta-oriented prioritization by considering combinations of different distance types, Hamming distance (H) and Jaccard distance (J) with minimum distance (M) and summation distance (S)

ble 4.4, we show the distribution of APFD values and their average over 200 sets of faults for a combination of Hamming and Jaccard distance with minimum distance and summation distance. By comparing the two left boxplots to the two right boxplots, which represent the two distance types: Hamming and Jaccard distance, respectively, the boxplots show that using Hamming distance is better than the Jaccard distance. In addition, the average APFD values over sets of faults for different similarity alternatives (cf. Table 4.4), confirm the aforementioned observation that Hamming distance (0.907, 0.906) outperforms the Jaccard distance (0.891, 0.891). With the Mann-Whitney U test, we observed that the difference between the Hamming and Jaccard distance for the minimum distance and the summation distance is significant with p-values 0.00084 and 0.018, respectively. Hence, answering **RQ3**, we show that distance measurements affect the results.

Furthermore, in the first two left boxplots from Figure 4.7, we observed a slight improvement, on average, when combining the Hamming distance with distance minimum

strategy over the combination of Hamming distance with distance summation strategy. This slight improvement can be observed from the average values (cf. Table 4.4) of distance minimum (0.907) and distance summation (0.906). However, by considering the Mann-Whitney U test, we observed that the difference is not significant with p-value 0.74.

In the two right boxplots in Figure 4.7, we show the results of using Jaccard distance with distance minimum and with distance summation. The boxplots show that they are approximately identical. The obtained p-value from the Mann-Whitney U test (1.0) approves that they behave similarly. Hence, we show that there is only a slight impact of using distance minimum or using distance summation when we calculate the distance between more than two products. This slight impact is observed only when we used Hamming distance. However, in order to generalize the aforementioned observation, more experiments using other case studies are required to be conducted.

4.5.4 Threats to Validity

In the following, we discuss the internal and external threats to validity that may affect our results.

There is a potential threat to the internal validity that may affect the results related to the random distribution of the seeded faults. To mitigate this threat, we generated 200 sets of faults for the single-wise and pairwise interactions. In each set, we select 10% of architecture model elements and mark them as they are faulty. We assume that the random distribution of the seeded faults is better than building on non-representative distributions. Considering 10% of the elements as seeded faults raises an internal fault that may affect the results. To alleviate this threat, we ran different experiments with other percentages. The experiments led to very similar results and, thus, to same interpretations. The raw results of all experiments are available online[3]. Another internal threat is that we compared our approach to random orders. To mitigate the random effects, we repeated those experiments 100 times.

Regarding external validity, we cannot ensure that our product line is representative for real-world software product lines. The product line BCS already served as a benchmark to evaluate product-line testing techniques [Lachmann et al., 2015; Lochau et al., 2014; Oster et al., 2011b; Lochau et al., 2012]. In addition, we are not aware of other industrial or even academic product-lines publicly available with their architecture models.

4.6 Related Work

Several approaches have been proposed to prioritize products based on different criteria. In Chapter 3, we proposed configuration-based prioritization to order products based on the similarity between them with respect to the feature selection. Similarly, we propose delta-oriented prioritization in this chapter to prioritize products by considering

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/thesis

the similarity between them, but in terms of deltas. In addition, we investigate a combined approach of the aforementioned approaches to differentiate between products. Furthermore, we investigate whether the distance functions, which are used to measure the similarity, and the way prioritization is computed (i.e., using the maximum or summation of distances) influence the results. Lity et al. [2017] propose to reduce the incremental product-line analysis efforts by optimizing product orders using adopted graph algorithms. In their work, they select the next product that is the less different to the previous ones to be analyzed to reduce the effort in regression analysis. However, their work is based on an assumption that the analysis process will be finished in the given time. In our work, we select a product that is least similar to the previously tested ones to be tested next in order to detect faults as soon as possible.

Sánchez et al. [2015] prioritize products by considering solution-space information as well as non-functional properties of features including following: the size, the number of changes, the cyclomatic complexity that represents the implementation of a feature, the number of test cases, the number of test assertions, number of developers and the number of reported installations. They use the Drupal framework as a case study and report that prioritizing products based on solution-space information, such as the size of features and its complexity code outperforms problem-space information (i.e., the feature selection) with respect to the early rate of fault detection. In a following work, Parejo et al. [2016] consider the aforementioned properties and the combination between them as inputs to their search-based algorithm by modeling product prioritization as a multi-objective optimization problem. Using the Drupal framework, they confirm the aforementioned observation reported by Sánchez et al. [2015] that considering solution-space information can increase the early rate of fault detection. In our work, we consider the solution-space information in terms of deltas to prioritize products.

Regarding the problem-space information in product prioritization, Henard et al. [2014b] propose search-based approach to sample products based on selected features. Similarly, Devroey et al. [2016] propose a search-based approach to select test cases for behavioral product-line models based on similarity with respect to the feature selection. Sánchez et al. [2014] propose to prioritize products using common feature model metrics. Henard et al. [2013c] apply the similarity-based technique to detect faults in feature models. They generate faults by mutating feature models. They report that the dissimilar test suites, in terms of feature selection, have a higher ability to detect faults than the similar ones. In our approach, we prioritize products based on their similarity in terms of deltas. With delta-oriented prioritization, we exploit the commonalities and differences between products in terms of solution-space artifacts (deltas) to prioritize products.

With respect to considering the domain knowledge in product prioritization, Devroey et al. [2014] select the products with high probability to be executed using statistical analysis of a usage model. Ensan et al. [2011] utilize the expectation of experts to select the most desirable features and give a higher priority to test products that include these features. Johansen et al. [2012b] prioritize the product based on a weight given to the T-wise interactions based on market knowledge. With delta-oriented prioritization, we

do not require expert knowledge to prioritize products. Instead, we follow a purely model-based gray-box approach solely requiring feature models and delta models as a basis.

Baller et al. [2014] introduce a framework to optimize products and test cases by considering certain objectives such as the costs and the profits of test cases and test requirements. Wang et al. [2013] automate the test case selection by mapping features to the test cases. In our approach, we use the similarity among products, in terms of deltas, as criteria to prioritize them. The test case prioritization is not considered in this thesis, which, however, can be extended in future work to enhance the product-line testing effectiveness.

Lachmann et al. [2015] propose to prioritize test cases using an integration testing approach for product lines based on delta-oriented test specifications. In their work, they prioritize test cases based on how many changed elements are covered by test cases. These changes are captured in the deltas. In following work, Lachmann et al. [2016] combine a prioritization based on structural and behavioral deltas with a dissimilarity approach to prioritize message sequence chart test cases. In our work, we use these delta models to prioritize products. Combining our approach with the aforementioned approach may improve effectiveness of product-line testing.

In single-system testing, Yoo and Harman [2012] survey efforts that have been made to select, minimize, and prioritize test cases in regression testing. For each technique, they introduce its goals and limitations. Hemmati and Briand [2010] propose to apply similarity-based testing on model-based test case selection. In our approach, we applied the same concepts, but to prioritize products to improve the effectiveness of product-line testing. Another work of Hemmati et al. [2011], where they investigate the properties of test suites in order to show under which conditions can similarity-based technique work as expected. Similar work can be done in the context of product-lines. However, instead of test cases, the generated products need to be considered. Henard et al. [2016] perform a comparison between black-box (problem-space information) and white-box (solution-space information) prioritization approaches with respect to the detection fault rate. In their results, they show a slight improvement of using white-box prioritization approaches to prioritize test cases over the black-box prioritization approaches. However, they recommend for further investigation in order to generalize their results. In addition, they do not investigate whether the combination of black-box and white-box prioritization approaches may enhance the software testing.

Rothermel et al. [2001] propose to prioritize test cases in regression testing using solution-space information. They consider code information, such as code coverage and the estimation on the ability of test cases to detect faults in test-case prioritization. Using the code information in product-line testing to prioritize products or the test cases might enhance product-line testing, as observed by Sánchez et al. [2015]. In our work, we show that considering more solution-space information in terms of deltas (i.e., white-box prioritization approach) in the context of product lines can potentially enhance the product-line testing. Moreover, we combine both approaches that consider

problem-space and solution-space information. Although the results of combining both approaches using our subject product-line do not show an improvement over considering only the solution-space information, combining both approaches will enable the testers to prioritize products regardless of the type of the given information.

4.7 Summary

The increasing interest in variable software systems in academia and industry requires different types of testing techniques to improve the quality of variable software systems. Several approaches have been proposed to improve product-line testing, such as reducing the number of products to test. Moreover, prioritization approaches have been proposed to increase the rate of early fault detection. In Chapter 3, we focus on problem-space information only. In this chapter, we propose delta-oriented prioritization to order products based on their similarity with respect to deltas. We compare the delta-oriented prioritization approach against random orders and the default order of the sampling algorithm *MoSo-PoLiTe*. The results show that prioritizing products based on delta modeling outperforms the random orders as well as the default order of the sampling algorithm *MoSo-PoLiTe*. Thus, we conclude that delta-oriented prioritization can enhance the effectiveness of product-line testing. In addition, we proposed a combined approach that considers problem-space (i.e., feature selection) and solution-space (i.e., delta modeling) information in product prioritization. The reported results indicate that adding more solution-space can improve the testing effectiveness.

Furthermore, we evaluated the impact of using Hamming and Jaccard distance in measuring the similarity of products. We reported that using Hamming distance outperforms Jaccard distance. Finally, we conclude that there is no significant impact of the way we calculate the distance among more than two products (i.e., maximum over the distance minimum, or the distance summation).

Based on our experiments in Chapter 3 and Chapter 4, we observed that sampling a set of configurations using the existing sampling algorithms takes considerable amount of time. Johansen et al. [2012a] also report that applying pairwise testing on a version of the Linux kernel with 6,888 features needs nine hours, which is a large amount of time considering the usual limited testing time. In addition, we observed that most of the existing sampling algorithms are non-deterministic. Thus, in the following chapter, we propose an incremental sampling algorithm that overcomes the aforementioned challenges.

5. Incremental Pairwise Sampling

This chapter shares material with the GPCE'16 paper "IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling" [Al-Hajjaji et al., 2016b]. The tool support of this work is presented also in the GPCE'16 demo paper "Tool Demo: Testing Configurable Systems with FeatureIDE" [Al-Hajjaji et al., 2016c].

As mentioned in the previous chapters, it is not possible to check whether each individual product meets its requirements due to the large number of possible products of a software product line. In a single system, an exhaustive test that includes all input parameters and their combinations is not feasible [Bernot, 1991]. Thus, several approaches have been proposed to minimize the testing space with respect to parameters, such as combinatorial interaction testing Cohen et al. [2003]. An instance of combinatorial interaction testing, pairwise testing, is a promising approach widely used to reduce the testing cost [Grindal et al., 2005; Nie and Leung, 2011; Cohen et al., 2003; Kuhn et al., 2008]. As we face a similar problem, even more complex, combinatorial interaction testing is proposed to be used in software product line testing [Oster et al., 2010; Perrouin et al., 2012, 2010; Henard et al., 2014b; Johansen et al., 2012a; Garvin et al., 2011; Oster et al., 2011c]. Considering features of the feature models as parameters, we generate a set of configurations that covers the feature combinations. Therefore, the number of products is reduced significantly. Covering T-wise combinations of features in the generated configurations results in a T-wise feature interaction coverage (cf. for more details about the T-wise testing, see Section 2.2). To achieve T-wise coverage, several sampling algorithms have been proposed, such as ICPL [Johansen et al., 2012a], CASA [Garvin et al., 2011], Chvatal [Johansen et al., 2011; Chvatal, 1979], MoSoPoLite [Oster et al., 2011c], and IPOG [Lei et al., 2007].

However, these algorithms do not scale well to larger product lines with respect to the CPU time and memory consumption that are required for sampling [Medeiros et al.,

2016; Henard et al., 2014b]. Specifically, most of the existing algorithms make the choice of giving results for the entire sample that achieves sufficient feature interaction coverage, instead of delivering intermediate results one by one. As a consequence they require a considerable amount of time to compute the result, even for smaller product lines.

Johansen et al. [2012a] report that generating products for a version of the Linux kernel (with 6,888 features) requires approximately nine hours for pairwise combinatorial interaction testing. As those samples are usually not available until a sampling algorithm is completely terminated, the sampling process may take a considerable part of the limited testing time. As a result, the Linux kernel developers use the built-in facility of the Linux kernel build system `randconfig` to generate random configurations, because none of the existing sampling algorithms scales to a more recent version of the feature model of the Linux kernel with over 15 thousand features [Melo et al., 2016]. An alternative to combinatorial interaction testing, several search-based approaches have been proposed to generate a set of products [Henard et al., 2014b; Ensan et al., 2012; Henard et al., 2013a]. However, the aforementioned approaches do not guarantee a certain degree of coverage. Although the number of generated products can be specified at the beginning of sampling, the process of generating these products is not incremental and non-deterministic (i.e., different configurations for each run). To tackle the aforementioned problems, we propose **Inc**remental samp**Ling** (IncLing) to generate products one at a time. Besides generating products incrementally, we aim also to enhance the sampling efficiency, in terms of the required time to generate a sample, as well as testing effectiveness, in terms of the interaction coverage rate (i.e., covering the combinations of features faster).

In particular, the IncLing algorithm samples products incrementally one by one, based on a greedy selection heuristics, to achieve pairwise coverage. With IncLing, while further products are added to the sample, IncLing selects the next product that covers as many of uncovered feature combinations as possible. In Chapter 3 and Chapter 4, we observed that ordering products based on their similarity may enhance the testing effectiveness. Thus, with IncLing, we propose a feature ranking heuristic to influence in which order these products are generated. In particular, we increase the diversity of the generated products by covering dissimilar pairwise feature combinations, each time a further product is generated. For this purpose, we reorder features, which serve as an input to our algorithms, based on how often they appear in the uncovered feature combinations. As a result, the covering rate of feature combinations is increased, which might lead to faster fault detection [Namin and Andrews, 2009; Frankl and Iakounenko, 1998; Cai and Lyu, 2005; Perry, 2006]. We dynamically continue generating further products until the testing time is exhausted or coverage is achieved.

The rest of this chapter is organized as follows. We explain IncLing using our running example in details in Section 5.1. In Section 5.2, we highlight the main characteristics that differentiate IncLing from state-of-the-art sampling algorithms. We present the implementation and the integration of IncLing to FeatureIDE in Section 5.3. We evaluate IncLing using a corpus of real-world and artificial feature models of different

Algorithm 5.1 Main algorithm of IncLing.

Require: $\mathcal{FM}$ ▷ a feature model of a product line
C_{old} ▷ the given configurations and/or the already generated ones
$Time$ ▷ the given testing time
$Coverage$ ▷ the required coverage
Return: P_{tested} ▷ list of generated and tested products

1: **function** IncLing($\mathcal{FM}$, C_{old}, $Coverage$, $Time$)
2: $F \leftarrow$ getFeatures($\mathcal{FM}$)
3: $G \leftarrow$ constructGraph($\mathcal{FM}$)
4: $freq \leftarrow$ emptyList
5: $signum \leftarrow$ emptyList
6: $lsCombs_{initial} \leftarrow$ generateCombinations($\mathcal{FM}$)
7: $lsCombs_{left} \leftarrow lsCombs_{initial} \setminus$ (generateCombinations(C_{old}) $\cup$ generateInvalidCombinations($\mathcal{FM}$, G))
8: forbidConfigurations($\mathcal{FM}$, C_{old})
9: **while** (coveredCombinations( ) $<$ $Coverage$ $\wedge$ passedTime( ) $<$ $Time$) **do**
10: updateFrequency($lsCombs_{left}$, $freq$, $signum$)
11: sort(F, $freq$)
12: $c_{new} \leftarrow \emptyset$
13: **for** $i \leftarrow 1$ **to** $|F|$ **step** 1 **do**
14: **for** $j \leftarrow 0$ **to** i **step** 1 **do**
15: $lsCombs_{test} \leftarrow$ getCombinations($lsCombs_{left}$, $signum$, $F[i]$, $F[j]$)
16: testCombinations($lsCombs_{test}$, c_{new})
17: **end for**
18: **end for**
19: autoComplete($\mathcal{FM}$, c_{new})
20: $C_{old} \leftarrow C_{old} \cup \{c_{new}\}$
21: $lsCombs_{left} \leftarrow lsCombs_{left} \setminus lsCombs_{covered}$
22: generateAndReturnProduct(c_{new})
23: $P_{tested} \leftarrow P_{tested} \cup \{c_{new}\}$
24: forbidConfigurations($\mathcal{FM}$, c_{new})
25: **end while**
26: **end function**

sizes and compare it against four sampling algorithms and random configurations. We present and discuss our results in Section 5.4. We point out related work in Section 5.5 and summarize the chapter in Section 5.6.

5.1 Incremental Pairwise Sampling with IncLing

We present the main algorithm for IncLing in pseudo-code in Algorithm 5.1. Furthermore, we show the main activities of IncLing in Figure 5.1. In the *initialization* phase,

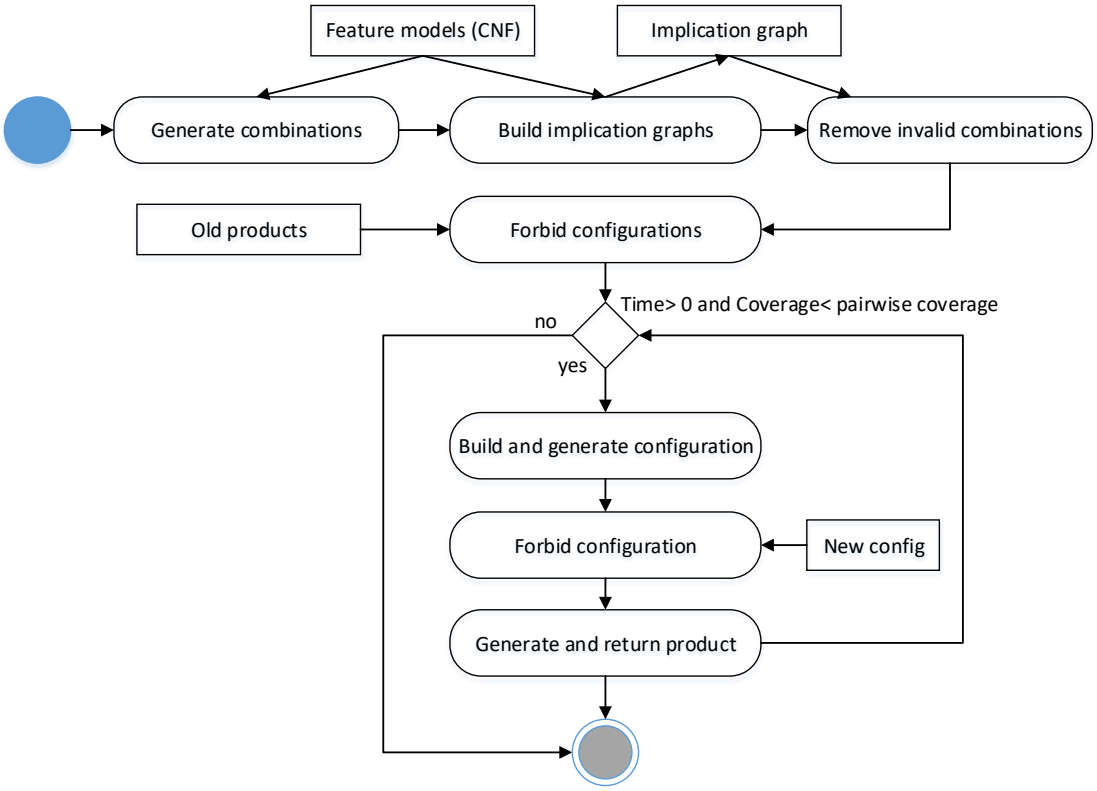


Figure 5.1: Activity diagram of IncLing.

we generate all pairwise combinations, remove the invalid ones, and consider the already generated products, if any. Then, we *generate products* one at a time. To generate a product, we *build a configuration* by consecutively *adding feature combinations* until the configuration is complete. We keep generating products until the pairwise coverage is achieved or the testing time is exhausted.

5.1.1 Initialization

The input of our algorithm consists of the feature model $\mathcal{FM}$, the set of products C_{old} that have been already tested, the desired pairwise *Coverage*, and the testing *Time* available. The output of the algorithm is a list of generated products to test. At the beginning, we initialize all required variables. First, we get all features from the feature model $\mathcal{FM}$ (Line 2). We transform feature models into propositional formulas. These formulas are written in Conjunctive Normal Forms (CNFs) (cf. Section 2.1.2). CNFs resemble a collection of constraints that must be fulfilled. These constraints, which are represented with clauses, are connected with the logical operator *conjunction*. Each clause consists of a disjunction of positive or negative variables (e.g., see Figure 2.4 on Page 11). In our case, the variables are the features. The variable is positive if the feature is part of a partial configuration, otherwise it is negative. These positive and

negative variables are defined as literals. Second, we construct the implication graph G for the feature model $\mathcal{FM}$ (Line 3). This graph is used to detect the invalid literal combinations. Third, we generate all pairwise combinations of literals $lsCombs_{initial}$ (Line 6). Fourth, we create a list of all uncovered literal combinations $lsCombs_{left}$ by removing the invalid literal combinations as well as the combinations of literals that are already covered in the set of configurations C_{old} (Line 7). Checking the satisfiability of adding a particular combination of literals is an NP-complete problem. Thus, we aim to reduce the number of SAT query by removing the invalid literal combinations using the constructed implication graph (see Section 5.2.2 for more detail about the implication graph). Finally, we add the given generated configuration C_{old}, to the feature model ($\mathcal{FM}$) as a blocking clause to avoid generating them (Line 8). This blocking clause contains a disjunction of positive or negative variables that are forbidden to be generated again (i.e., $\mathcal{FM} \bigwedge_{c_i \in C_{old}} (\neg c_1 \vee \neg c_2, \vee ..., \vee \neg c_n)$).

Example 5.1. *We demonstrate the single steps of our algorithm with the help of the* graph *product line example (cf. Figure 2.2 on Page 9). We have the following list of seven features F = (*GraphLibrary*,* Edges*,* Directed*,* Undirected*,* Algorithms*,* Number*,* Cycle*). The list of all possible pairwise literal combinations consists of 168 elements in total.*

*After the removal of all invalid literal combinations, such as (*Directed*,* Undirected*) (i.e., both features cannot be chosen together) and (*¬Algorithms*,* Number*) (i.e., contradiction of a parent-child relationship), the list contains* 111 *valid literal combinations that need to be covered (e.g., (*Directed*,* ¬Undirected*), (*¬Algorithms*,* ¬Cycle*), (*Cycle*,* Directed*), ...). To explain the following steps in a simple way, let us assume we do not have any already tested products (i.e., C_{old}=0).*

5.1.2 Generating Products

In the main loop of the algorithm (Lines9–25), we build configurations until either running out of testing time or there are no more literal combinations to cover. As illustrated in Figure 5.2, before we build a configuration, we calculate the current *signum* and *frequency* of each feature, which we use for our heuristic. Frequency denotes how often each feature occurs in the uncovered literal combinations. In Line 11, we rank features in descending order according to their frequencies. The signum of a feature is positive, if the feature is more often selected than deselected in the uncovered literal combinations and negative otherwise. We use the signum to increase the diversity among configurations with regard to feature selections (Line 15). Note that, the invalid literal combinations are taken into account for calculating the *frequency* and *signum*, as we consider them as covered (Line 7). Ignoring invalid literal combinations, which can never be covered normally, leads to a biased heuristic, as they are missing in the calculations. For instance, the ranking of the features would favor features that appear in many invalid literal combinations.

Next, we build a new configuration (Lines 12–19), which we describe in more detail later on. For each new configuration, we exclude all literal combinations that are covered by

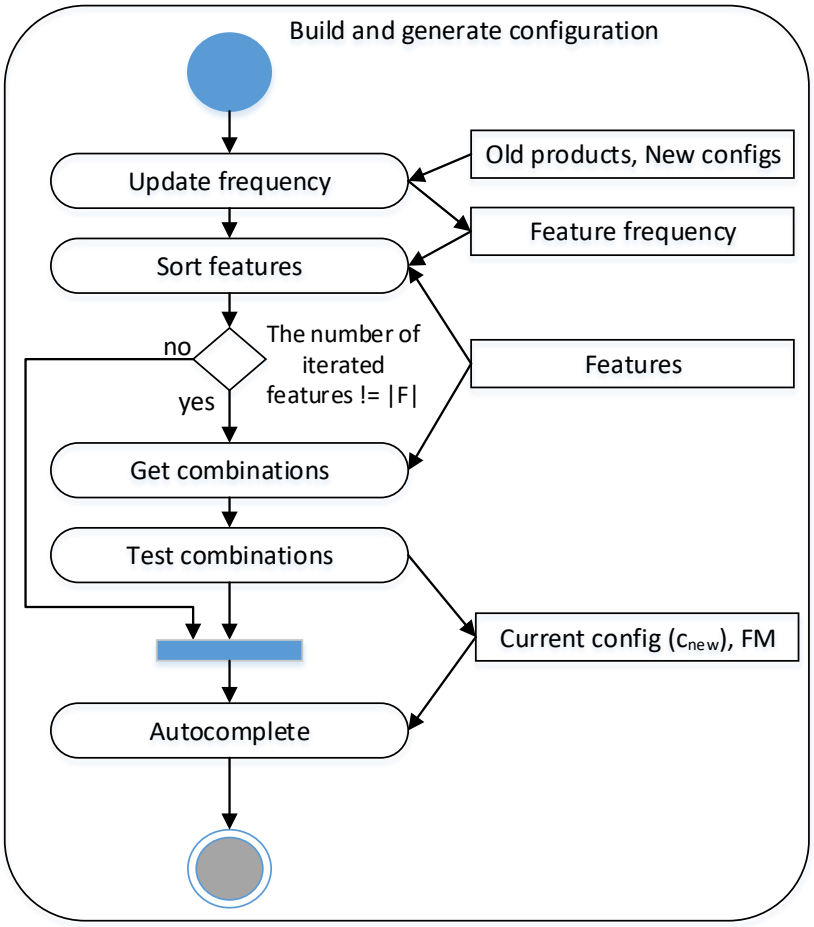


Figure 5.2: Activity diagram of generating products with IncLing.

the configuration from the list of uncovered literal combinations $combs_{left}$ (Line 21). Finally, we generate a product from this configuration (Line 22), which may then be tested in parallel while the algorithm goes to the next iteration. We avoid building the same configurations by excluding the already generated ones from the feature model ($\mathcal{FM}$) via adding a blocking clause (Line 24). This process continues until either we reach a certain degree of pairwise feature coverage or we run out of testing time (Line 9).

Example 5.2. *In our example, we list the initial computation of frequency and signum in Table 5.1. The numbers for frequency range from 0 and 168, which represent the minimum and maximum frequency for seven features, respectively. When we rank the features accordingly, we get the list* $F =$ (Number, Algorithms, Cycle, Directed, Undirected, GraphLibrary, Edges)*, where the first feature* (Number) *has the highest frequency. From Table 5.1, we notice that the signum value of feature* (Number) *is* -1*. That is, the feature* Number *should be part of the next configuration.*

Feature	G	E	D	U	A	N	C
Frequency	155	155	161	161	162	163	162
Signum	11	11	1	-1	2	-1	-2

Table 5.1: The frequency and the signum for each feature in the uncovered literal combinations list for product line *GraphLibrary*

5.1.3 Building a Configuration

As we show in Algorithm 5.1, Line 12, we first create an empty configuration c_{new}. Then, we execute two nested loops over the ranked feature list to generate the combinations for each pair of features (Lines 13–18). For each pair of features, we call the function `getCombinations` to generate a list of uncovered literal combinations and order them using the signum values (Line 15). We then test whether we can cover one of these literal combinations within the current configuration c_{new} by executing the function `testCombinations` (Line 16).

Example 5.3. *For the first product in our example, we generate the following literal combinations. The first feature pair according to the feature ranking is* Number *and* Algorithms*. In this case, the function* `getCombinations` *returns this list of literal combination* $lsCombs_{left}$ *= ((*¬Number, Algorithms*), (*Number, Algorithms*), (*¬Number, ¬Algorithms*)). The combination (*Number, ¬Algorithms*) is excluded in the initialization phase as it is an invalid one. These valid literal combinations are then tried to be added to the current configuration.*

5.1.4 Testing a Combination

In Algorithm 5.2, we present the function `testCombinations`. In addition, we show the main activities of this function in Figure 5.3. In this function, we iterate over the given list of literal combinations $lsCombo_{test}$ and test for each combination whether it is possible to include it in the current configuration c_{new}.

First, we check whether a feature from the tested combination of literals is already included in the current configuration (Line 5). In this case, we continue with the next literal combination in $lsCombo_{test}$ if at least one of the following conditions is true: c_{new} contains the complement of at least one literal in the combination (i.e., the complement of literal A is ¬A and vice versa), the combination is already contained in c_{new}, or the percentage of covered combinations is below a certain threshold. The reason for moving to the next literal combination if the previous condition is *true* is that we aim to cover as many of the uncovered combinations for each a new configuration. Thus, considering a combination where a one literal of it is already part of the current configuration will not help us to achieve that aim. Otherwise, we test using a satisfiability solver whether it is possible to add the combination to the current configuration c_{new}. If the current configuration is not valid, we move to the next combination.

Algorithm 5.2 Tests whether a combination from the list $lsCombs_{test}$ can be added to the current configuration c_{new}.

Require: c_{new} ▷ the current configuration
$lsCombs_{test}$ ▷ the combination of literals
Return: c_{new} ▷ the current configuration c_{new}

```
1:  function testCombinations(lsCombs_test, c_new)
2:      for each combo ∈ lsCombs_test do
3:          f_a ← combo[0]
4:          f_b ← combo[1]
5:          if ({f_a, f_b, ¬f_a, ¬f_b} ∩ c_new ≠ ∅) ∧
                ( ( ¬f_a ∈ c_new ∨ ¬f_b ∈ c_new) ∨ (f_a ∈ c_new ∧ f_b ∈ c_new) ∨
                  coveredCombinations( ) < threshold( ) )
            then
6:              continue
7:          end if
8:          if f_a ∉ c_new then
9:              if ¬ isSatisfiable(c_new ∪ {f_a}) then
10:                 c_new ← c_new ∪ {¬f_a}
11:                 return
12:             end if
13:         end if
14:         if isSatisfiable(c_new ∪ {f_b}) then
15:             c_new ← c_new ∪ {f_a, f_b}
16:             return
17:         else
18:             if f_a ∈ c_new then
19:                 c_new ← c_new ∪ {¬f_b}
20:             end if
21:         end if
22:     end for
23: end function
```

We use a threshold value as a mechanism to speed up the algorithm. Before reaching a certain amount of covered literal combinations, we exclude every combination with at least one literal of a feature in the current configuration (i.e., $(\{f_a, f_b, \neg f_a, \neg f_b\} \cap c_{new} \neq \emptyset)$), which considerably reduces the number of literal combinations that we have to check. After reaching the threshold, we consider all remaining literal combinations.

In the `testCombination` function, we do not test the entire combination at once, but try to add the literals consecutively. We start by testing the validity of adding the first feature from the combination f_a to c_{new} (Lines 8–13). If c_{new} is no longer satisfiable, we conclude that f_a is conditionally core or dead (cf. Section 2.1.2) and, thus, we add the complement of f_a to c_{new} and return from the function. Next, we test the validity of adding the second feature f_b to c_{new}. If c_{new} is still satisfiable, we are able to cover the

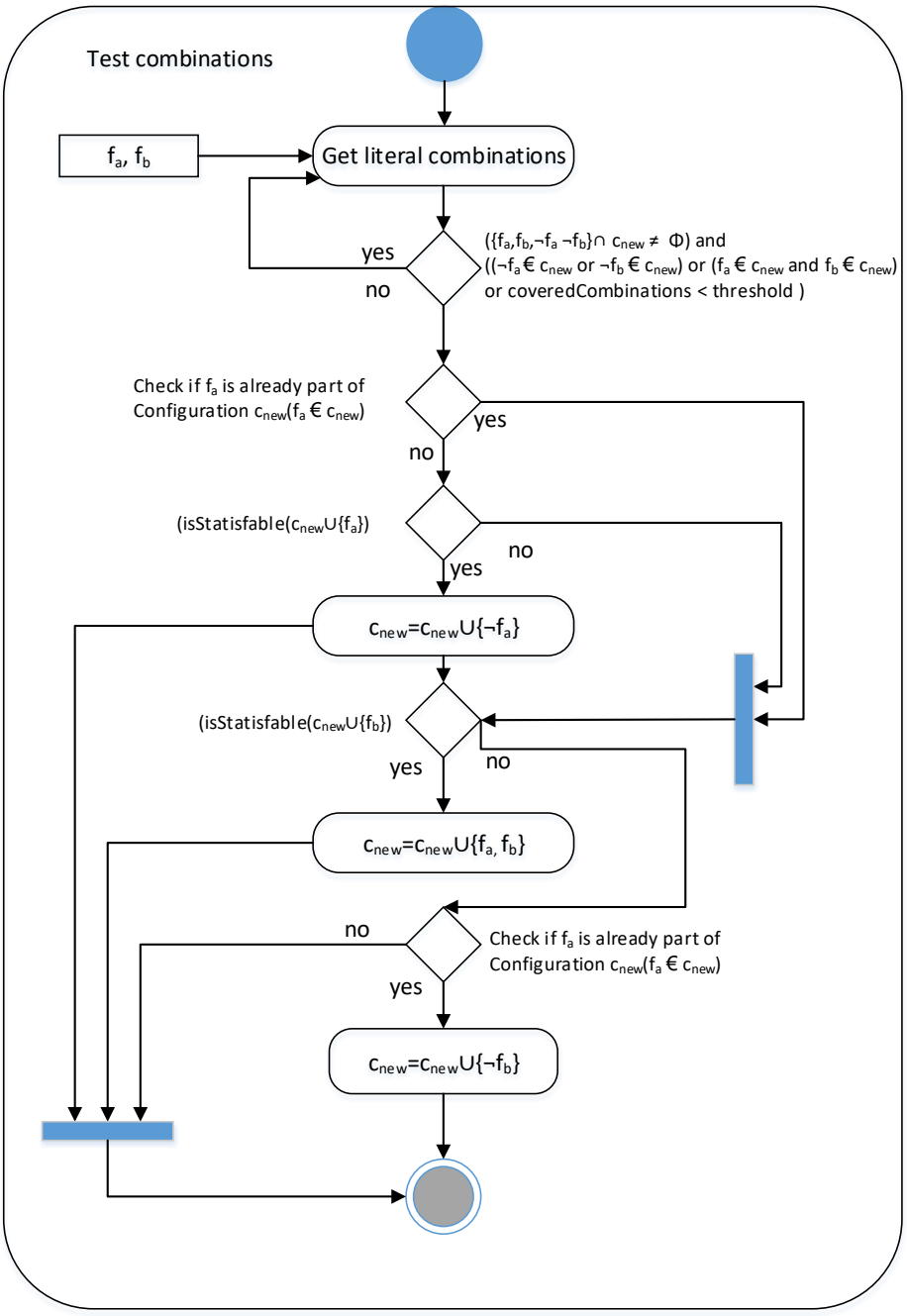


Figure 5.3: Activity diagram of the function *testCombinations()*.

given combination and return from the function. Otherwise, we have to check whether feature f_a is already part of the current configuration c_{new}. If so, the complement of f_b is added to the current configuration c_{new} (Lines 17–20).

Example 5.4. *Regarding our example, the first input of the function is the list of literal combinations* $lsCombs_{test}$ *=((*¬Number, Algorithms*), (*Number, Algorithms*), (*¬Number, ¬Algorithms*)) and an empty configuration. We start by testing the first configuration (*¬Number, Algorithms*). Since none of both features are included in the current configuration, we add* ¬Number *to* c_{new} *and find that it is still satisfiable. Next, we add literal* Algorithms *to* c_{new}*. As* c_{new} *is still satisfiable, we successfully added the literal combination (*¬Number, Algorithms*) to the current configuration and return from the function.*

Continuing the execution of our algorithm, the next input consists of the list of literal combinations $lsCombs_{test}$ *=((*Number, Cycle*), (*Number, ¬Cycle*), (*¬Number, Cycle*), (*¬Number, ¬Cycle*)) and the current configuration* c_{new} = {¬Number, Algorithms}*. The feature* Number *is already contained in the configuration. Thus all literal combinations containing* Number *or* ¬ Number *are ignored. Thus, the next relevant feature pair is* Cycle *and* Directed *with the literal list* $lsCombs_{test}$ *= ((*¬Cycle, Directed*), (*Cycle, Directed*), (*¬Cycle, ¬Directed*)). Again, we add the literal* ¬Cycle *to* c_{new} *and find that it is still satisfiable. However, when we add literal* Directed*, the configuration becomes unsatisfiable. Thus, we remove both features from* c_{new} *and test the next combination (i.e.,* (Cycle, Directed*)). Given the current configuration, it is possible to select both features and, thus, we add them to* c_{new} *and continue with the next feature pair. Further continuing the process results in the configuration* c_{new} *=*{¬Number, Algorithms, Cycle, Directed, ¬Undirected, GraphLibrary, Edges}*. To avoid generating the same configuration again, we add the current configuration* c_{new} *to the feature model* $\mathcal{FM}$ *as a blocking clause (i.e.,* $\mathcal{FM} \wedge \neg(C_{new})$*).*

5.2 Main Characteristics of Incremental Pairwise Sampling

We propose IncLing to efficiently generate configurations for sample-based product-line testing. The product-line testing process, as considered in the following, includes creating configurations (e.g., sampling), generating products, and finally testing them. Similar to ICPL [Johansen et al., 2012a], IncLing generates new products by sequentially selecting a combination of features that are not already covered by previously selected configurations. Although our algorithm is based on the general concept of ICPL, we propose four major modifications. In the following, we explain these modifications and their impact in more detail.

5.2.1 Incremental Approach

IncLing generates products incrementally, whereas in the current implementation of ICPL, the user has to wait until all feature combinations are covered. The incremental nature of our algorithm has the advantage that products can be generated and tested in parallel until the testing time is exhausted or the desired coverage is achieved. Our incremental approach enables us to utilize the testing time efficiently, because particular

products can be tested immediately after sampling returned a configuration. However, a potential drawback of the incremental approach is that the order of the products cannot be adapted before testing. However, the order in which products are generated using IncLing is influenced by considering these products that are already generated and tested. We explain more about influencing products order when we discuss the feature ranking heuristic. Note that with a modification to the current implementation of existing sampling algorithms, such as ICPL [Johansen et al., 2012a] and MoSo-PoLiTe [Oster et al., 2010], these algorithms can also generate products incrementally.

5.2.2 Detecting Invalid Combinations

Invalid combinations are all those literal combinations that are impossible to cover due to feature-model dependencies. As checking the validity of a product is an NP-complete problem, it is likely not be solved in polynomial time. Thus, reducing the SAT queries, which is used to check the validity of a combination, may enhance the performance of the sampling algorithm.

ICPL removes invalid combinations after it covered a certain number of combinations [Johansen et al., 2012a]. MoSo-PoLiTe [Oster et al., 2010] removes the invalid combinations on the fly. During the configuration building, whenever MoSo-PoLiTe finds a combination that cannot be part of a configuration, the combination is removed [Oster, 2012]. In contrast, IncLing removes invalid combinations at the beginning of the sampling process. The advantage of detecting invalid combinations at the beginning is that the algorithm has to consider only valid combinations and thus saves computation time. With IncLing, we detect the invalid combinations efficiently using implication graphs, which we describe in more detail later on, that are created based on the corresponding feature models [Krieter et al., 2018]. The potential disadvantage of detecting these invalid combinations at the beginning is that it needs additional time at the beginning, which is the cost of saving effort during the sampling process.

Detecting Invalid Literal Combinations Using Implication Graphs

Reducing the number of SAT queries will reduce the required time to derive a valid configuration, as each query is an NP-complete problem. Hence, we consider an approach that uses *implication graphs* to reduce the number of necessary queries to the satisfiability solver [Krieter et al., 2018].

With implication graphs, certain relationships between features in a feature model can be represented by implication graphs. Detecting the invalid literal combinations using implication graphs requires two steps. First, graph construction, where an implication graph for the feature model is computed. Computing the implication graphs is based on the observation that defining features (i.e., selected or deselected) affects the definition of a small set of other features in a feature model as a result of its relationships, such as *parent-child* relationships. In case the definition of other additional features is also affected, it usually results of involving this feature in *include* or *exclude* constraints. Constructing the graph is executed only once unless the feature model is modified.

Second, the created graph is utilized to minimize the number of satisfiability solver queries by identifying the unaffected features as well as inferring the selection state of the affected features. During the invalid combinations detection, the implication graph is traversed in order to find potentially implied literals. For instance, each literal l_i can be reached by literal l_j, the combination of both literals is valid. Thus, we infer from the previous that the combination of literal l_i and literal $\neg l_j$ (i.e., $l_i \wedge \neg l_j$) is an invalid literal combination without a need to query the SAT solver. Following the previous step yields a reduction in the number of SAT queries, which improves the IncLing performance in general. Similar to our step of constructing implication graphs, the sampling algorithm MoSo-PoLiTe [Oster, 2012] translates feature models into binary constraint graphs. Then, forward checking [Haralick and Elliott, 1980] is used to identify whether the literal combination cannot be part of a valid configuration (cf. Section 2.2.2).

5.2.3 Feature Ranking Heuristic

In Chapter 3 and Chapter 4, we prioritize products based on their similarity with respect to the problem-space information (i.e., feature selection) and solution-space information (i.e., delta modeling), respectively. The prioritized products can be generated using sampling algorithms, which, however, require a considerable amount of time. Hence, with IncLing, we propose to influence in which order these products are generated by reordering the input feature list of our algorithm. In Section 5.1.2, we demonstrated the way these features are ranked using our running example in more detail. As already mentioned, IncLing uses a heuristic that ranks the features based on the previously generated products. With each new product, this greedy strategy aims to cover the maximum number of pairwise feature combinations that have not already been covered by the previously generated products. Thus, our algorithm has the potential advantage of covering many feature combinations as fast as possible. Moreover, testing time may be saved as no need to generate the entire set of products at the beginning and then prioritize them. However, a potential drawback is that we cannot ensure the order of the generated products is optimal.

Compared to existing sampling algorithms, ICPl does not consider any particular heuristic approach that influence in which order products are generated. However, we observed in our evaluation of Chapter 3, there is an implicit order as a result of their goal of covering as many uncovered combinations as soon as possible. To the best of our knowledge, besides IncLing, only the sampling algorithm MoSo-PoLiTe [Oster, 2012] has a ranking heuristic, where a literal that appears frequently in the uncovered literal combinations has a higher probability to be part of the upcoming configuration.

5.2.4 Detecting Conditionally Dead or Core Features

IncLing uses a satisfiability solver to test whether it is possible to select or deselect a feature in the current product. ICPL also uses satisfiability solver to determine whether a combination of literals can be (de)selected simultaneously. In contrast to ICPL, IncLing does not test whether *the whole combination of literals* can be selected in the

current product simultaneously, but each individually (i.e., IncLing tests whether adding a literal of a combination to the partial configuration is satisfiable, instead of testing the whole combination). However, it can be beneficial to test features individually, as this detects features that are conditionally dead or core [Benavides et al., 2010]. A feature can be a conditionally *dead* or *core* if it is under certain circumstances. For instance, given the feature model and already fixed features of the current configuration, features in some cases must be selected or deselected as a result of feature dependencies (cf. Section 2.1.2). Consequently, combinations that include these features do not need to be considered, since they will be covered automatically. Hence, by reducing the number of SAT queries, the overall performance of the algorithm is improved.

5.3 The Integration of IncLing in FeatureIDE

As mentioned in Chapter 3, we have already integrated several sampling algorithms to FeatureIDE [Meinicke et al., 2017; Thüm et al., 2014b], such as CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979; Johansen et al., 2011], and ICPL [Johansen et al., 2012a]. To overcome the limitation of existing sampling algorithms, with our implementation of IncLing, we generate products in an incremental manner. That is, developers do not have to wait until the whole sampling process is over in order to start testing. Once, the configuration is created, the corresponding product is generated and then tested. In addition, we enable developers to specify the maximum number of products to be tested, which is not possible for several sampling algorithms, especially the greedy ones, such as ICPL and Chvatal. The current implementation of IncLing supports pairwise testing, which could be extended in the future to a higher T-wise interaction coverage. Our implementation is publicly available as a part of the current release of FeatureIDE.

Due to the scalability problem of the existing sampling algorithms, it is common in practice to generate random products to be tested, especially for large product lines (e.g., randconfig in Linux) [Melo et al., 2016]. For this purpose, we also implemented a random generator to create a fixed number of random configurations based on the satisfiability solver Sat4J [Le Berre and Parrain, 2010]. The method can efficiently generate a large number of distinct random configurations. Once the configuration is created, it is included immediately to the feature model as a blocked clause (i.e., $foreach..$) in order to avoid generating the same configuration again. This generator is used in our thesis for evaluation purposes. However, it could be also used to evaluate other approaches related to feature model analysis. For more details about the way of using these functionalities, we refer the reader to Section A.1 in the Appendix.

5.4 Evaluation of IncLing

We evaluate IncLing against existing sampling algorithms and random configurations with respect to three criteria: sampling efficiency, testing efficiency, and testing effectiveness. With sampling efficiency, we refer to the aggregated computation time of the sampling algorithms to achieve pairwise coverage. Testing efficiency counts the number

of products generated by the sampling algorithms to achieve pairwise coverage. For testing effectiveness, we consider the increase of pairwise interaction coverage achieved by the product order generated by our approach compared to the product order returned by the other approaches. With respect to these three criteria, we focus on answering the following research questions.

RQ1 Does IncLing increase the average sampling efficiency for achieving pairwise coverage compared to existing sampling algorithms?

RQ2 Does IncLing decrease product-line testing efficiency compared to existing sampling algorithms?

RQ3 Does IncLing increase product-line testing effectiveness compared to random configurations and existing sampling algorithms?

5.4.1 Experiment Settings

In the experiments, we consider real and artificial feature models of different sizes in terms of the number of features and different complexity in terms of the ratio of the number of distinct features in cross-tree constraints to the number of features (CTCR). The real feature models consist of up-to 6,888 features. The size of the artificial feature models, which we use in our experiment, ranges between 15 and 5,542 features. In Table 5.2, we summarize the properties of each feature model. We report for each feature model the number of features, the number of constraints, and CTCR.

To evaluate IncLing, we generate configurations that are required to achieve the pairwise coverage criterion. For the threshold mentioned in Section 5.1, we use 99%. We repeated the experiment for each setting five times and calculated the average in order to reduce the impact of outliers in our measurements. As an exception for random configurations, we conducted the experiment 100 times to additionally mitigate the effect of random impacts. For a fair comparison, we set the number of created configurations for random configurations to the same number of configurations as created with IncLing for the respective feature model. We performed the experiments using a PC with an Intel Core i5-4670 CPU @ 3.40 GHz, 16 GB RAM, and Windows 7. In the next section, we discuss the results of our experiments.

5.4.2 Results and Discussion

In this section, we present and discuss the results of the IncLing evaluation.

Sampling Efficiency (RQ1)

To answer RQ1, we compare our approach, with respect to the required time to achieve the pairwise coverage, against existing sampling algorithms, namely CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979], ICPL [Johansen et al., 2012a], and IPOG [Lei

Feature Model	Features	#Constraints	CTCR
Email	10	3	50%
Violet	88	27	66%
BerkeleyDB1	53	20	42%
BerkeleyDB2	99	68	82%
Dell	46	110	80%
EShopFIDE	192	21	10%
EShopSplot	287	21	12%
GPL	27	16	63%
SmartHome22	60	2	6%
BattleofTanks	144	0	0%
FM_Test	168	46	28%
BankingSoftware	176	4	2%
Electronic Shopping	290	21	11%
DMIS	366	192	93%
eCos 3.0 i386pc	1,245	2,478	99%
FreeBSD kernel 8.0.0	1,369	14,295	93%
Automotive1	2,513	2,833	28%
Linux_2_6_28_6	6,888	6,847	99%
10xAFM15†	15.0	2.6	19%
10xAFM50†	50.0	9.7	17%
10xAFM100†	100.0	20.0	17%
10xAFM200†	200.0	39.0	17%
10xAFM500†	500.0	100.0	17%
10xAFM1K†	1,000.0	100.0	14%
1xAFM5K	5,542	300	11%

CTCR: cross-tree constraints representative

†: The values next to the corresponding artificial feature models represent the average values over 10 feature models

Table 5.2: Feature models used in our evaluation

et al., 2007]. Unfortunately, we did not consider the sampling algorithm MoSo-PoLiTe [Oster et al., 2010] in our evaluation, because we could not have access to the tool. Nevertheless, Johansen et al. [2012a] compare ICPL against MoSo-PoLiTe and report that the sampling efficiency, with respect to the required time to achieve T-wise coverage, of ICPL outperforms the sampling efficiency of MoSo-PoLiTe. Thus, comparing IncLing to ICPL can give us a clue on how IncLing could perform compared to MoSo-PoLiTe.

In Table 5.3, we show the time that is required to achieve pairwise coverage using IncLing compared to existing sampling algorithms. We observed that some of these sampling algorithms do not scale well. We stopped computation that required more than 24 hours. We refer to these cases in Table 5.3 as (*).

Feature Model	CASA	Chvatal	ICPL	IPOG	IncLing
Email	0.13	**0.01**	0.03	0.46	**0.01**
Violet	59.75	1.38	0.31	4841.02	**0.12**
BerkeleyDB1	47.60	0.59	0.08	664.22	**0.03**
BerkeleyDB2	5240.35	1.61	0.21	(*)	**0.05**
Dell	54.95	0.18	0.10	2562.60	**0.04**
EShopFIDE	4789.30	23.16	1.20	(*)	**0.30**
EShopSplot	(*)	15.57	0.72	(*)	**0.22**
GPL	14.58	0.24	0.15	173.65	**0.06**
SmartHome22	14.40	0.32	0.05	169.56	**0.03**
BattleofTanks	16955.84	11.72	3.38	(*)	**2.75**
FM_Test	11945.02	4.05	0.62	16997.68	**0.20**
BankingSoftware	1423.98	4.13	0.48	19187.96	**0.15**
ElectronicShoping	12512.83	16.08	0.84	88680.47	**0.22**
DMIS	6969.84	26.19	1.66	(*)	**0.47**
eCos 3.0 i386pc	(*)	1334.02	78.92	(*)	**22.90**
FreeBSD kernel 8.0.0	(*)	2155.77	105.28	(*)	**32.68**
Automotive1	(*)	56344.51	7602.94	(*)	**510.91**
Linux_2_6_28_6	(*)	(*)	18805.14	(*)	**3296.41**
10xAFM15	0.77	0.02	0.02	3.20	**0.01**
10xAFM50	30.84	0.27	0.06	643.10	**0.02**
10xAFM100	3066.64	1.45	0.27	11991.00	**0.09**
10xAFM200	(*)	9.64	1.19	(*)	**0.43**
10xAFM500	(*)	189.4	14.97	(*)	**4.76**
10xAFM1K	(*)	1744.55	228.98	(*)	**35.66**
1xAFM5K	(*)	(*)	35172.81	(*)	**1870.72**

*: No result within 24 hours of computation

†: The values next to the corresponding artificial feature models represent the average values over 10 feature models

Table 5.3: The computation time of different sampling algorithms (in seconds) using feature models of different sizes.

Examining the values of the required time to achieve the pairwise coverage for each feature model, we observe that IncLing outperforms the other sampling algorithms for all feature models, except for feature model *Email* (with 10 features), where the computation sampling time of IncLing is equal to the computation sampling time of Chvatal. For instance, for feature model *GPL* (27 features), we notice that the required time to cover all pairwise combinations for IncLing is 0.06 second, while the sampling algorithms CASA, Chvatal, ICPL, and IPOG require 14.58, 0.24, 0.15, and 173.65 seconds. That is, with IncLing, we can save 99.6%, 75%, 60%, and 99.9% of CASA's, Chvatal's, ICPL's, and IPOG's sampling time, respectively. For feature model *Automotive1*, the required time for IncLing is 8.6 minutes. Sampling algorithms ICPL and Chvatal re-

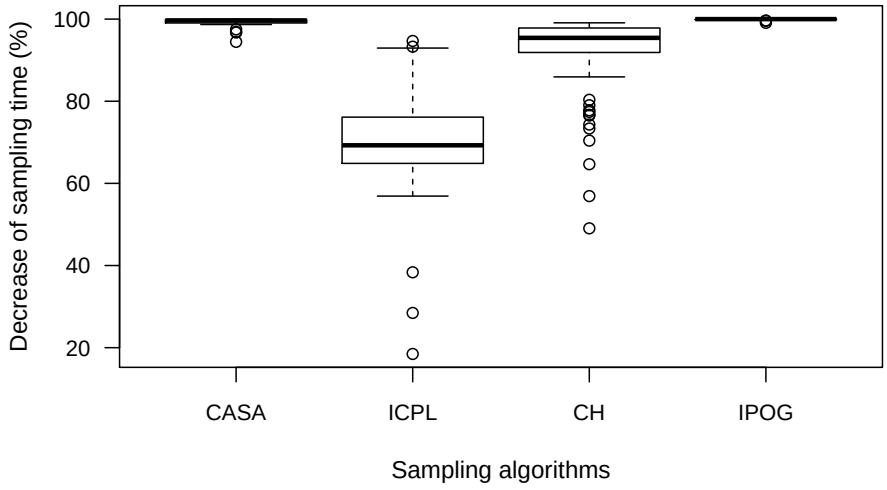


Figure 5.4: Distributions of decrease in computation time (percentage) of our approach compared to sampling algorithms over all feature models, (CH: Chvatal algorithm).

	CASA	CH	ICPL	IPOG
Avg. decrease time	99.1%	91.8%	70%	99.9%

Table 5.4: The average time decrease for our approach to sampling algorithms over all feature models, (CH: Chvatal algorithm).

quire 126.72 minutes and 939.08 minutes with 93% and 99.1% decrease of its sampling time, respectively. For the feature model of the *Linux kernel*, only IncLing and ICPL scale to this large feature model. The times required for IncLing and ICPL to achieve the pairwise coverage are 0.92 hours and 5.2 hours, respectively. In particular, 82.5% sampling time can be saved when IncLing is used instead of ICPL.

To compute the saved time percentage of using the sampling algorithm IncLing, we calculate the relative time decrease compared to existing algorithms with the following formula:

$$TimeDecrease = (1 - \frac{Time_{IncLing}}{Time_{Existing}}) \cdot 100\% \tag{5.1}$$

where $Time_{IncLing}$ is the sampling time of IncLing and $Time_{Existing}$ is the time of the existing sampling algorithm.

In Figure 5.4, we show the distribution of relative time decrease of existing sampling algorithms when using IncLing instead (cf. Equation 5.1). We observe an improvement

Feature Model	CASA	Chvatal	ICPL	IPOG	IncLing
Email	**6.0**	7.0	7.0	8.0	9.0
Violet	**22.0**	27.0	28.0	28.0	25.0
BerkeleyDB1	**19.6**	25.0	24.0	22.0	24.0
BerkeleyDB2	**19.0**	24.0	21.0	(*)	22.0
Dell	**33.0**	38.0	37.0	43.0	43.0
EShopFIDE	24.0	22.0	**21.0**	(*)	23.0
EShopSplot	(*)	23.0	**21.0**	(*)	23.0
GPL	**15.0**	20.0	19.0	18.0	22.0
SmartHome22	**14.8**	18.0	17.0	17.0	17.0
BattleofTanks	664.0	**448.0**	460.0	(*)	650.0
FM_Test	**40.0**	47.0	45.0	51.0	42.0
BankingSoftware	**37.0**	41.0	42.0	49.0	47.0
ElectronicShoping	31.4	23.0	24.0	26.0	**22.0**
DMIS	27.0	27.0	29.0	(*)	**26.0**
eCos 3.0 i386pc	(*)	66.0	**63.0**	(*)	64.0
FreeBSD kernel 8.0.0	(*)	77.0	77.0	(*)	**75.0**
Automotive1	(*)	**913.0**	**913.0**	(*)	946.0
Linux_2_6_28_6	(*)	(*)	479.0	(*)	**450.0**
10xAFM15	**11.0**	12.0	12.0	13.2	16.7
10xAFM50	**24.0**	26.9	26.6	28.0	28.0
10xAFM100	**49.4**	56.0	56.6	63.5	58.5
10xAFM200	(*)	**88.1**	89	(*)	91.7
10xAFM500	(*)	185.2	189.1	(*)	**186.4**
10xAFM1K	(*)	346.7	346.7	(*)	**340.2**
1xAFM5K	(*)	(*)	685.0	(*)	**607.0**

*: No result within 24 hours of computation

†: The values next to the corresponding artificial feature models represent the average values over 10 feature models

Table 5.5: The number of generated products of different sampling algorithms using feature models of different sizes.

of our approach compared to the other sampling algorithms. If IncLing is used, the median values of the decrease of time for existing sampling algorithms range between 65% and 99.9%. The average time decrease over all feature models is illustrated in Table 5.4, where we show that 99.1%, 70%, 91.8%, and 99.9% of CASA's, ICPL's, Chvatal's, and IPOG's sampling time can be saved using IncLing.

We conclude from the results in Table 5.3, Figure 5.4, and Table 5.4 that IncLing is more efficient than the existing sampling algorithms with respect to the required time to achieve the pairwise coverage.

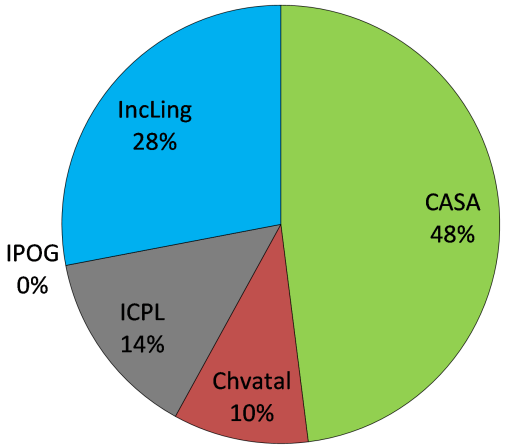


Figure 5.5: The percentage of feature models with the minimum number of generated products for each sampling algorithm.

Testing Efficiency (RQ2)

Regarding RQ2, we expect that we need more products to achieve pairwise coverage compared to existing sampling algorithms. We collected the number of generated products for each feature model to achieve pairwise coverage. In Table 5.5, we report the required number of products for each feature model to achieve pairwise coverage (the minimum ones are highlighted in bold font). As illustrated in Table 5.5 and Figure 5.5, CASA generates the least number of products for most feature models that it was able to sample (48% of feature models). In addition, we observe that IPOG does not generate the minimum number of products for any feature model. In the case of Chvatal, ICPL, and IncLing, they generate the minimum number of products for 10%, 14%, and 28% of feature models, respectively.

Moreover, we conducted the Mann-Whitney U Test to investigate whether the differences between IncLing and the existing sampling algorithms are significant regarding the number of generated products for each feature model (cf. Section 3.3.1). In our results, we observe that the difference is not significant between IncLing and CASA, Chvatal, ICPL, and IPOG with p-values 0.12, 0.70, 0.65, and 0.71, respectively. While it is not our primary goal to generate the minimum number of products, p-values above indicate that most of the sampling algorithms have similar testing efficiency, because the differences are not significant.

Testing Effectiveness (RQ3)

To answer RQ3, we measure the potential gain of testing effectiveness of IncLing with respect to the increase of interaction coverage achieved by the product order compared

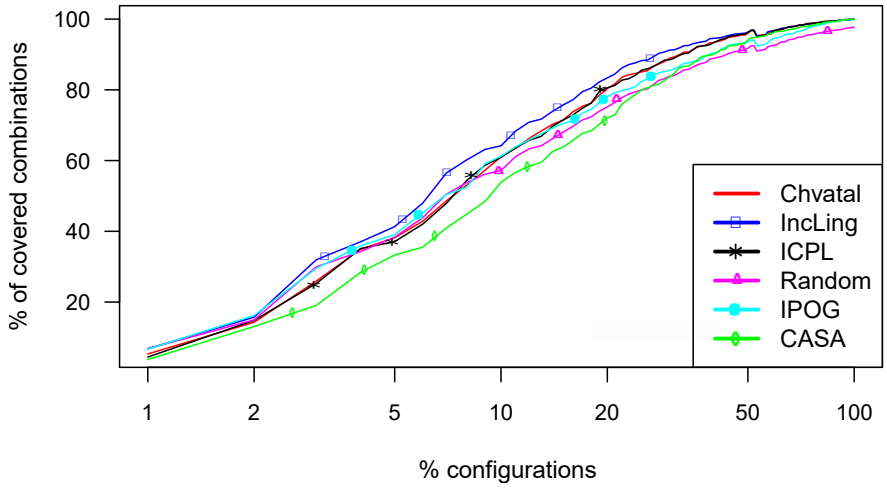


Figure 5.6: Average percentage of covered combinations over all feature models with size $< 200 features$ (i.e., all sampling algorithms scale to these feature models), in addition to random configurations.

to random configurations and the existing sampling algorithms. In Figure 5.6, we show the percentage of covered combinations to the percentage of configurations for the 38 feature models that could be sampled by all evaluated algorithms. From Figure 5.6, we observe that on average for the first 40% of the generated configurations, IncLing covers more combinations than the other sampling algorithms. IPOG covers more combinations than CASA, Chvatal, ICPL, and random configurations until approximately 8% of the generated configurations. However, with random, we do not achieve 100% coverage. Surprisingly, CASA performs worse than all other sampling algorithms and even than random configurations.

In Figure 5.7, we show the average percentage of covered feature combinations for the relative number of configurations generated by Chvatal, IncLing, ICPL, and random configurations considering only feature models between 200 and 3,000 features. We find that IncLing covers more feature combinations for the first 3% of configurations than Chvatal, ICPL, and random. For Chvatal and ICPL, we notice that they behave almost similar. In the case of random configurations, it does not compete with the other sampling algorithms. Comparing the results between Figure 5.6 and Figure 5.7, we observe a reduction of the covered combinations by IncLing for the first generated configurations. However, 3% of the configurations is still an acceptable rate. For instance, for product line *Automotive1*, 3% means that IncLing covers more feature combinations for the first 27 configurations (the total number is 913) than the other sampling algorithms.

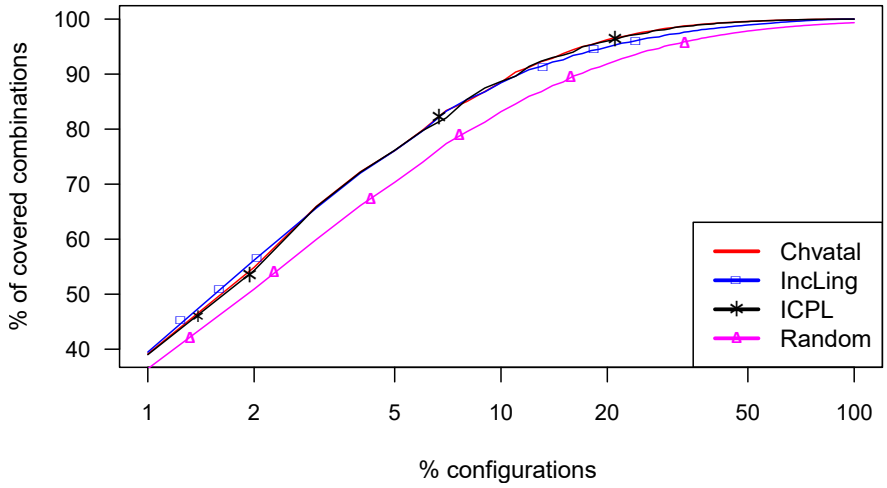


Figure 5.7: Average percentage of covered combinations for feature models between 200 and 3000 features using IncLing, ICPL, Chvatal, and random configurations.

Looking closer at the large feature models, we show in Figure 5.8 the aggregated number of covered combinations of *the Linux kernel* for IncLing, ICPL, and random configurations. Note that IncLing and ICPL are the only sampling algorithms scaling to this large feature model. The results show that IncLing covers more feature combinations in the first 26 and three configurations for *AFM5K* and *Linux*, respectively. We argue that increasing the diversity among configurations, which is intended in IncLing, yields to cover more feature combinations as early as possible. As a result, many faults may be detected more quickly, which could enhance the testing effectiveness. To answer RQ3, we show that there is a potential gain in testing effectiveness with respect to the interaction coverage of IncLing. In particular, we found that IncLing increases the testing effectiveness regarding the interaction coverage, because up-to the first 40% of the generated configurations by IncLing cover more feature combinations than all existing sampling algorithms.

5.4.3 Threats to Validity

In this section, we introduce validity threats of our experiments that may affect our results and discuss the steps that we considered to mitigate those threats.

Internal Validity

There is a potential threat that may affect the results with random configurations. To mitigate random effects, we conducted the experiments 100 times. Another internal threat is that we compared IncLing to the existing sampling algorithms. Some of these

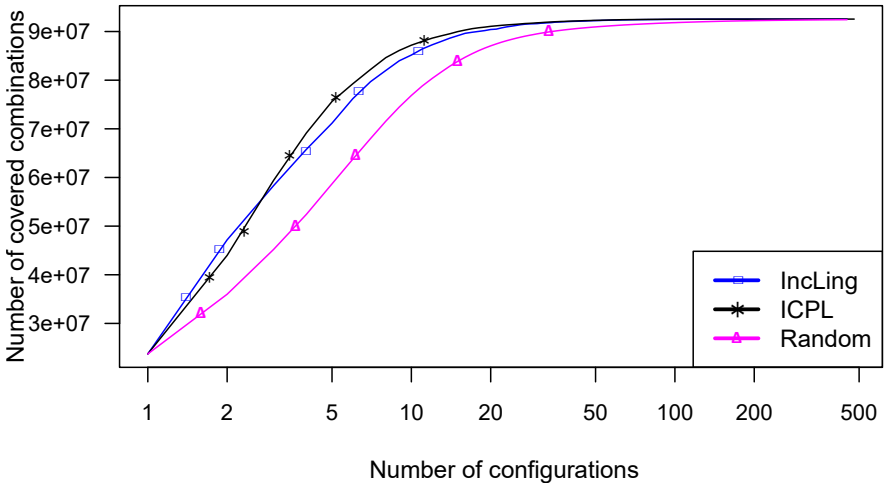


Figure 5.8: Number of covered combinations for the feature model of Linux kernel (6,888 features) using IncLing, ICPL, and random configurations.

sampling algorithms, such as CASA, are known to be non-deterministic. Moreover, in some cases we also observed a non-deterministic behavior for Chvatal and ICPL. To minimize the impact of this threat, we repeated all experiments five times. Furthermore, another potential threat is that the implementation of IncLing could contain errors which may affect the presented results. To limit this threat, we divided the implementation task into sub-tasks to have better control on the overall implementation. In addition, the results of comparing IncLing to the existing sampling algorithms give us a confidence in the IncLing implementation. Besides that, we made our implementation publicly available as IncLing is currently integrated to FeatureIDE.

External Validity

We cannot guarantee that the artificial feature models we used in our evaluation are able to simulate realistic real-world product lines. Furthermore, we cannot ensure that the compared algorithms will provide similar results on different sets of feature models. What mitigates this threat is that we evaluate IncLing using artificial feature models of different sizes and complexity, which already served as a benchmark to evaluate product-line testing [Johansen et al., 2011, 2012a; Henard et al., 2014b]. In addition, we used real-world feature models, which represent the variability of complex product lines, such as a version of the Linux kernel with 6,888 features. Another threat is that we considered the interaction coverage in this chapter as a testing effectiveness measure.

5.5 Related Work

In the literature, several approaches have been proposed to sample sets of products [Perrouin et al., 2012; Cohen et al., 2007; Lochau et al., 2012; Kim et al., 2011; Shi et al., 2012; Johansen et al., 2011, 2012a; Carmo Machado et al., 2014]. Combinatorial interaction testing is a promising approach that has been used to select a subset of products [Kuhn et al., 2004]. Perrouin et al. [2010] propose a non-deterministic approach that divides the T-wise generation problem into several sub problems that can be solved automatically. To solve these sub problems, a set of products is generated to cover part of the T-wise interactions using Alloy [Jackson, 2012]. The union of these sets of products covers the whole T-wise interactions. A potential drawback of this approach is that the number of the generated products to be tested will be large. Thus, their approach does not scale to large feature models. As we showed in our results, IncLing scales for large feature models when pairwise coverage is applied.

Johansen et al. [2011] adopt the Chvatal algorithm Chvatal [1979] to generate covering arrays from feature models. They propose ICPL based on Chvatal with several enhancements, such as parallelizing the process of generating covering arrays, which reduces the computation time significantly [Johansen et al., 2012a]. ICPL tries to cover as many uncovered feature combinations as possible with each configuration added to the covering array. The configurations are included to the covering arrays until all valid combinations of features are covered. Although the process of generating the covering arrays is incremental, with the current implementation of the algorithm, the users have to wait until all valid combinations are covered to generate and test products. With IncLing, we generate the configurations incrementally. In addition to considering the pairwise coverage, we can also generate configurations within a given testing time. Furthermore, with our approach, we are trying to increase the diversity among the created configurations. In addition, the results show that the performance of our approach potentially outperforms the performance of ICPL, especially for large feature models. Medeiros et al. [2016] compare ten sampling algorithms regarding the size of the samples and their capability of fault detection. They report that existing sampling algorithms do not scale well, as they require a considerable amount of time. With IncLing, we show that its efficiency outperforms all sampling algorithms for pairwise coverage.

Oster [2012] proposes a pairwise sampling algorithm, called MoSo-PoLiTe, to generate samples. To do so, MoSo-PoLiTe transforms feature models into a binary constraint satisfaction problem, which represents a high-level description of a problem based on constraints. With IncLing, we represent feature models as forms of propositional formulas and, therefore, we handle the feature models as propositional satisfiability problem. MoSo-PoLiTe detects invalid combinations on the fly during the sampling. Whenever it finds an invalid combination cannot be part of any valid configuration, this combination is removed. In contrast, IncLing detects these invalid combinations at the beginning of the sampling process. It exploits the *built implication graph* to detect them efficiently. To check the validity of adding a combination to a partial configuration, MoSo-PoLiTe uses forward checking [Haralick and Elliott, 1980], while IncLing

uses the SAT solver [Mendonça et al., 2009b]. Both sampling algorithms, IncLing and MoSo-PoLiTe, influence in which order these products are generated by exploiting ranking heuristics of feature combinations. It will be interesting to conduct an experiment in the future that compares between both algorithms to investigate their scalability as well as their strong and weak points.

Garvin et al. [2011] propose CASA to generate covering arrays using simulated annealing. The algorithm does not scale to large feature models and it takes a long time to generate the covering arrays. Compared to IncLing, the performance of IncLing outperforms CASA significantly. However, considering a search-based algorithm may help IncLing to avoid being trapped in local optima. Hence, combining IncLing with search-based techniques is subject to future work. Henard et al. [2014b] propose an alternative to combinatorial interaction testing by employing a search-based approach to sample products. They propose the similarity notion to increase the interaction coverage for the generated products. Ensan et al. [2012] propose an evolutionary sampling approach to generate a set of products based on genetic algorithm. The generated products are evaluated using two aspects: the variability and the integrity constraints. They quantify these aspects using the following metrics: variability coverage, which represents the number of optional features that are required to generate a particular product, and cyclomatic complexity, which represent for each product the number of integrity constraints that are involved in the product generating. Similar to the previous two approaches, with IncLing, we can test a fixed number of products or test as many products as possible in a fixed time. In addition, with our approach, we consider the pairwise combinatorial interaction testing to sample products. Moreover, our approach is deterministic, which is not the case with this search-based approach, where different products might be generated in each run. Lopez-Herrejon et al. [2013] propose a multi-objective pairwise approach that considers maximizing the coverage and minimizing the number of products. While our main aim with IncLing is to generate these products efficiently in terms of the computation time, we also consider the two mentioned objectives during the sampling process.

Kowal et al. [2013] propose to provide additional information to feature models about the actual source of feature interaction. They use this information to reduce the number of products that need to be tested. However, additional information about the actual interaction between features is typically not available. With IncLing, we are trying not only to reduce the number of products, but also to generate them incrementally and efficiently. Johansen et al. [2012b] use domain knowledge to generate a set of products by exploiting market information. Similarly, Ensan et al. [2011] propose to test a set of products that contain the most desirable features recognized by customers. Kim et al. [2011] propose to reduce the number of products under test by using static analysis techniques that identify *irrelevant* features for testing. Their approach is based on the observation that some of the combinations of these features are unnecessary to be considered in testing.

As an alternative to combinatorial interaction testing, several fault-based approaches exploit the concept of mutation testing to generate a set of products [Henard et al.,

2014a; Reuling et al., 2015; Arcaini et al., 2015]. For instance, Henard et al. [2014a] propose a search-based approach that explores the configuration space of a product line to generate a set of products. For this purpose, they mutate the propositional formulas and they select these products based on their ability to detect mutants of feature models. Similarly, Reuling et al. [2015] introduce fault-based sampling to generate a set of products by mutating the feature diagrams. In addition, they discuss the possibility of using their approach to enhance the effectiveness of the generated products by sampling algorithms. By mutating the feature model of a product line, Arcaini et al. [2015] propose to generate a set of products based on their ability to distinguish the original version feature models from its faulty. While we use combinatorial interaction testing to generate products, the aforementioned fault-based approaches can be used to evaluate the generated products of IncLing.

5.6 Summary

As testing the entire products of a product line is infeasible, several approaches have been proposed to sample a set of products as representatives for the entire products of a product line while achieving a certain degree of coverage. However, existing sampling algorithms require a considerable amount of time to sample products, which may not all be tested due to the limitation of testing time. Thus, we propose IncLing to sample products one at a time. With our approach, there is no need to wait for a long time to have the first samples. Besides generating a fixed number of products within a given time, we can achieve a pairwise coverage.

Using feature models of different sizes, we evaluated IncLing against existing sampling algorithm with respect to three criteria, sampling efficiency (i.e., the computation time of sampling to achieve pairwise coverage), testing efficiency (i.e., the required number of products to achieve pairwise coverage), and testing effectiveness (i.e., how fast the pairwise interactions are covered). Regarding the sampling efficiency, the results of applying IncLing show significant improvements over existing sampling algorithms. In particular, applying IncLing can save, on average, at least 70% of the sampling time of existing algorithms. With respect to the testing efficiency, in some cases, IncLing generates more products than existing sampling algorithms. However, we argue that it is an acceptable cost compared to the saved time. Regarding testing effectiveness, IncLing covers as many feature interactions as soon as possible by increasing the diversity among the created configurations. In particular, with the same number of products, IncLing covers more feature interactions than existing sampling algorithms, which is likely to enhance product-line testing effectiveness.

6. Conclusion and Future Work

In the following, we conclude our thesis and discuss potential future work on product lines with respect to product prioritization and efficient sampling.

6.1 Conclusion

The increasing interest in variable software systems in academia and industry requires different types of testing techniques to improve the quality of product lines. While testing a single system is already a difficult task due to the limited testing resources, testing a product line is even more challenging due to the vast number of products that can be generated. Thus, reducing the number of products is a necessary step in product-line testing, and, therefore, several approaches have been proposed to select a set of products to be tested, such as combinatorial interaction testing.

Combinatorial interaction testing is a well-recognized testing approach that has been employed to restrict the number of products by generating a subset of these products systematically while achieving a certain degree of coverage. In spite of the fact that combinatorial interaction testing already reduces the number of products significantly, this number can still be large, especially for large product lines. Thus, the order in which products are tested matters to improve the use of testing time, by finding faults faster. For this purpose, we propose similarity-driven product prioritization that considers problem-space and solution-space information to prioritize products.

The aim of similarity-driven product prioritization is to increase the interaction coverage rate by increasing the diversity among products under test. To achieve that aim, we consider the feature selections, as problem-space information, and deltas, as solution-space information, to differentiate between products. Moreover, we propose to combine feature selections and deltas in product prioritization, where the impact of each can be adjusted using a weight factor. If testers wish to focus on a subset of products under test, we propose to cluster them into groups. Clustering products can also be

helpful to reduce the testing efforts in case the diversity among products is extremely large. For instance, clustering products into groups may enable testers to avoid wasting testing efforts as a result of changing the testing environment each time a very dissimilar product to the previously tested one is tested next. The results show that considering feature selections and deltas in product prioritization can enhance the product-line testing effectiveness with respect to the early rate of fault detection. In addition, the results reveal that more information about these products to differentiate between them leads to better testing effectiveness in terms of fault detection rate.

One of the challenges we observed during our evaluation is the considerable time that is required to sample products lines. The existing sampling algorithms do not scale well for large product lines, and even for small ones, these sampling algorithms require a considerable amount of time. Furthermore, these sampling algorithms do not produce products until the sampling process is finished. In addition, some of these sampling algorithms are not deterministic. Thus, a different order, and even for some sampling algorithms, a different product number, can be generated for each run over the same product line. To overcome the aforementioned limitations of these sampling algorithms, we propose the IncLing algorithm that generates products incrementally while achieving a certain degree of coverage. IncLing is a deterministic algorithm that generates products efficiently with the goal of increasing the interaction coverage as soon as possible. During the sampling, we influence the order in which products are sampled by ranking the features. The results of applying IncLing can save, on average, between 70% and 99% of the computation time compared to existing sampling algorithms.

6.2 Future Work

We identified several points that offer the potential for future work regarding the two main contributions of our thesis.

Product Prioritization in Product-Line Testing

In our work, we prioritize products based on the similarity between them with respect to feature selections and deltas. In future, considering more solution-space information, such as the source code should be investigated as it may enhance the effectiveness of software product-line testing. The main challenge of considering the source code in product prioritization is the initial cost of measuring the similarity between products, which could be addressed in future. In this thesis, we have applied our prioritization approach using three subject product lines with source code, an automotive product line (body comfort system), and feature models of different product-lines sizes. While we tried to generalize our results by considering different product lines, the prioritization approaches in future should be applied to real-world systems that have been implemented by different product-line implementation techniques.

In our product prioritization, we aim to find faults faster by increasing the rate of feature interaction coverage as a result of exploiting the diversity among products under

test. In case the diversity among products is large per definition, testing these products may require a considerable time as a result of the execution of a higher number of redundant tests cases, especially in regression testing [Lity et al., 2016]. To reduce the efforts in integration-based testing, we propose to test next the most similar product to the previously tested one [Lity et al., 2017]. Therefore, the number of test cases that is required to be executed will be minimized as the difference between the consecutive tested products is small. However, a trade-off between the goal of finding faults faster and reducing the testing effort should be applied. That is, handling product prioritization as an optimization problem that considers multi-objectives, possibly conflicting ones, in product prioritization should be investigated.

Furthermore, in addition to prioritizing products, which we consider in this thesis, combining test-case prioritization with product prioritization should be considered in future as it may enhance the effectiveness of product-line testing.

Sampling in Product-Line Testing

IncLing guarantees that the generated products of a product line achieve pairwise interaction coverage. To cover a higher value of interaction coverage (i.e., $T > 2$), IncLing should be extended in the future. However, applying a higher degree of interaction coverage yields a large number of products. In addition, these higher degrees of feature interactions occur rarely in practice. Thus, approaches that detect these feature interactions in a product line should be proposed. Considering only these detected feature interactions can reduce the number of generated products as combinations of features that do not interact with each other might be avoided during sampling.

In general, fault-based testing techniques are used to measure the effectiveness of test cases [Jia and Harman, 2011]. Recently, researchers pay attention to the fault-based testing techniques in software product line engineering [Henard et al., 2014a; Reuling et al., 2015; Arcaini et al., 2015; Al-Hajjaji et al., 2016a, 2017a; Carvalho et al., 2018]. Some of these techniques have been used to generate products as well as evaluating them by checking their possibility of containing faults. However, most of the existing approaches either consider only faults in feature models [Henard et al., 2014a; Reuling et al., 2015; Arcaini et al., 2015], which are not enough as most of the faults are in the source code [Abal et al., 2014], or they are not mature enough to be applied immediately. Thus, in future, building on the existing approaches and investigating their ability to generate products that have a higher probability of having faults should be further investigated.

A. Appendix

The Appendix is organized as follows. In Section A.1, we give an overview on the functionalities of FeatureIDE that support testing product lines. In particular, we present the testing support of product-by-product testing as well as other functionalities that are integrated to avoid limitations of product-by-product testing, such as avoiding redundant tests. In Chapter 3, Section 3.4, we discuss the aggregated results of cluster-based product prioritization evaluation. In section, we present the detailed results of the proposed approach (cf. Table A.1). Finally, in Section A.3, we introduce the architecture definition of the core product for the Body Comfort System (BCS), which we used to evaluate delta-oriented prioritization (cf. Chapter 4).

A.1 Testing Software Product Line with FeatureIDE

This section shares material with GPCE'16 demo paper "Tool Demo: Testing Configurable Systems with FeatureIDE"[Al-Hajjaji et al., 2016c].

Testing a product line includes many activities, such as creating configurations, building the corresponding products, testing them by executing their test cases, and using the results for debugging [Perrouin et al., 2010]. In Chapter 3 and Chapter 5, we already introduced the implementation support for the corresponding approaches. In addition, we implemented and integrated other approaches that have been used in our evaluation, such as the sampling algorithms CASA, Chvatal, and ICPL. In this section, we summarize the aforementioned implementation support and discuss other testing functionalities that are integrated into FeatureIDE.

In practice, developers tend to test only one or a few products, as testing all valid products only scales to product lines with a small number of features [Medeiros et al., 2015]. Considering only a single configuration is not enough as some faults may only appear in some configurations [Jackson and Zave, 1998]. Ideally, all valid products

of a software system should be tested, especially for safety-critical systems. However, testing all valid configurations is often not possible due to the combinatorial explosion in the number of features and due to limited testing resources. Several strategies have been proposed to reduce the number of configurations that need to be tested, such as generating a reduced, yet sufficient subset of configurations [Kuhn et al., 2004; Johansen et al., 2012a; Garvin et al., 2011; Thüm et al., 2014a], random configurations, or user-defined configurations. These strategies require tedious manual effort, such as configuring and generating the software system as well as checking its validity. In addition, they may require a specialized testing framework. As a result, these strategies are often not applied in practice. Hence, automating the testing process is a necessary step to use the testing resources wisely.

To automate the testing process, numerous sampling tools have been introduced to create configurations while achieving a certain degree of coverage, such as MoSo-PoLiTe [Steffens et al., 2012], CASA [Garvin et al., 2011], Chvatal Johansen et al. [2011], ICPL Johansen et al. [2012a], and IncLing [Al-Hajjaji et al., 2016b]. Furthermore, Henard et al. introduce the PLEDGE tool to create and prioritize products based on their dissimilarity [Henard et al., 2013b]. Moreover, Bürdek et al. present a tool that systematically explores similarities among products to enhance the testing efficiency [Bürdek et al., 2015]. While the aforementioned tools show promising results, each of them focuses only on testing rather than supporting the entire product-line development process.

We extended our tool FeatureIDE to automate the process of creating configurations, building products, and testing them by integrating the corresponding strategies. Our extension to FeatureIDE includes the following:

- Derivation of configurations using different techniques namely, (a) deriving all valid configurations, (b) generating random configurations, (c) using user-defined configurations, and (d) T-wise sampling.
- Generation of program variants independent of the programming paradigm, such as preprocessors and feature-oriented programming.
- Customization of product generation, such as adjusting the maximum number of created configurations and the order in which they are tested.
- Testing of generated program variants using JUnit and derivation of configuration-aware test results.
- Support to avoid redundant test executions and to achieve family-based testing.

In this section, we discuss the testing work-flow that we automated in FeatureIDE. We show a screen-shot of the FeatureIDE perspective in Eclipse in Figure A.1. In the following, we explain each element and how it is integrated into FeatureIDE for testing purposes. The single elements are as follows: At ①, we show the source code of the program that we want to test, including two unit tests. At ②, we show the feature

model that defines the variability of the program [Batory, 2005]. The folder `configs` at ④ shows three configurations that are user-defined. The folder `products` at ⑤ shows generated sample programs that are used for testing. The result of the tests for the configurations is shown in the JUnit view at ③.

A.1.1 Developing Product Lines with FeatureIDE

Features in product line can have dependencies among each other (e.g., one feature might require or exclude another one). To specify the dependencies of features, FeatureIDE provides a feature model editor shown in Figure A.1.②. The feature model is the central part of projects in FeatureIDE as it defines the variability of the systems that is used for configuration [Benavides et al., 2010; Pereira et al., 2016] and analyses (e.g., to detect unused features or dead code) [Tartler et al., 2011].

FeatureIDE supports the feature-oriented implementation of product line and is designed as an extensible framework [Thüm et al., 2014b]. In particular, it supports a variety of implementation mechanisms, such as feature-oriented programming [Batory et al., 2004], aspect-oriented programming [Kiczales et al., 1997], preprocessors [Meinicke et al., 2016a], and runtime variability.

In our example, we show a product line using the integrated preprocessor Antenna (cf. Figure A.1 ①). To configure the system, the user can manually define configurations using the configuration editor of FeatureIDE [Pereira et al., 2016]. In the example of Figure A.1, there are three *user-defined* configurations in the folder `configs` at Figure A.1.④. Only one configuration can be active at a time, which is then used to preprocess and compile the source code of the `src` folder.

Derive Configurations

To automatically derive configurations as well as to generate and test the products, we provide a dialog in which the user can choose how to derive the configurations. The user can open the dialog of building products via *FeatureIDE* → *Product Generator* in the context menu of the Project-/Package-Explorer. We show the dialog in Figure A.2. In the following, we discuss the meaning of the options provided in the dialog. As mentioned in Chapter 3 and Chapter 5, we support in FeatureIDE several strategies to provide configurations for testing, namely using user-defined configurations, deriving all valid configurations, using T-wise sampling, and randomly generating configurations. The user can select any of these strategists from the drop-down box *Strategy* in the dialogue *Build Products* (cf. Figure A.2).

Using the integrated configuration editor [Pereira et al., 2016], *User-defined* configurations can be created manually. These configurations are contained in the folder `configs` (cf. Figure A.1.④). Using an algorithm that exploits the tree structure of the feature model, *all valid* configurations can be generated. As this algorithm scales only for product lines with a few number of features, we provide alternatively a strategy to generate a fixed number of *random* configurations (cf. Chapter 5, Section 3.2). Furthermore, in addition to our sampling algorithm *IncLing*, we integrated several T-wise

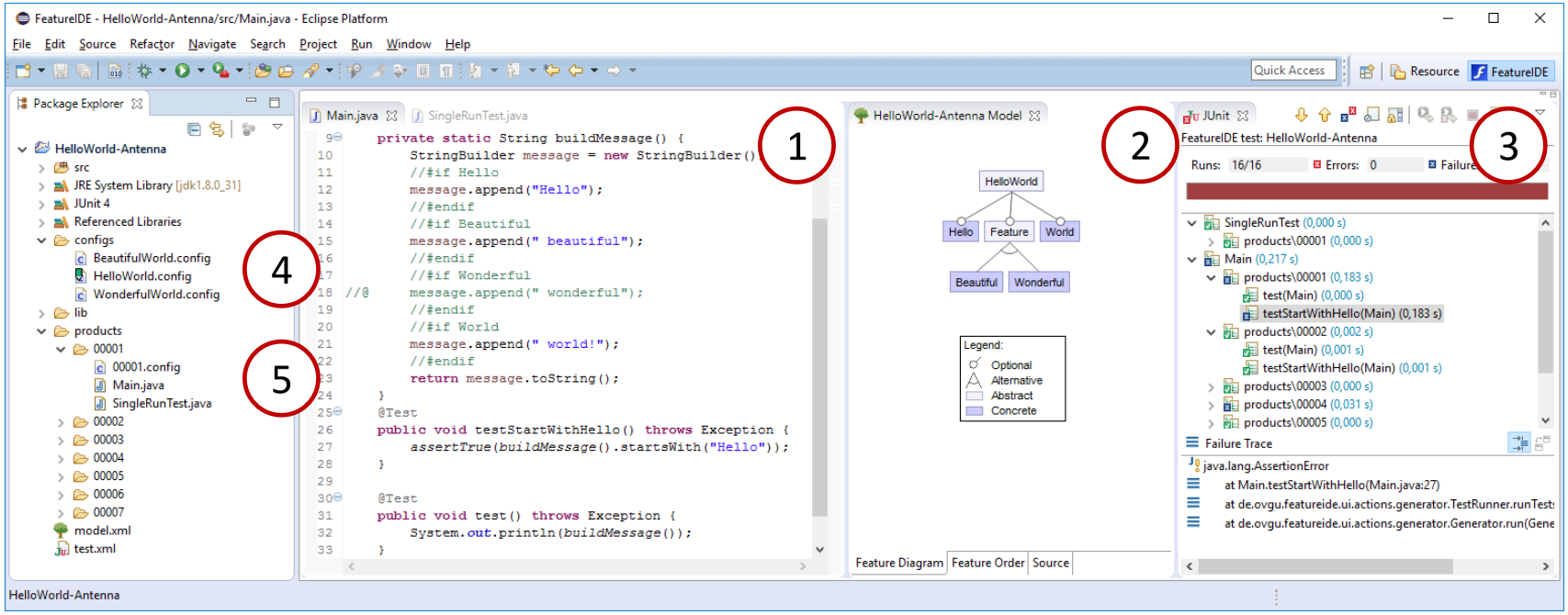


Figure A.1: Support for testing with FeatureIDE: ① source code of a program including two unit tests, ② feature model defining valid combinations, ③ JUnit view, ④ user-defined configurations, and ⑤ a set of generated sample products.

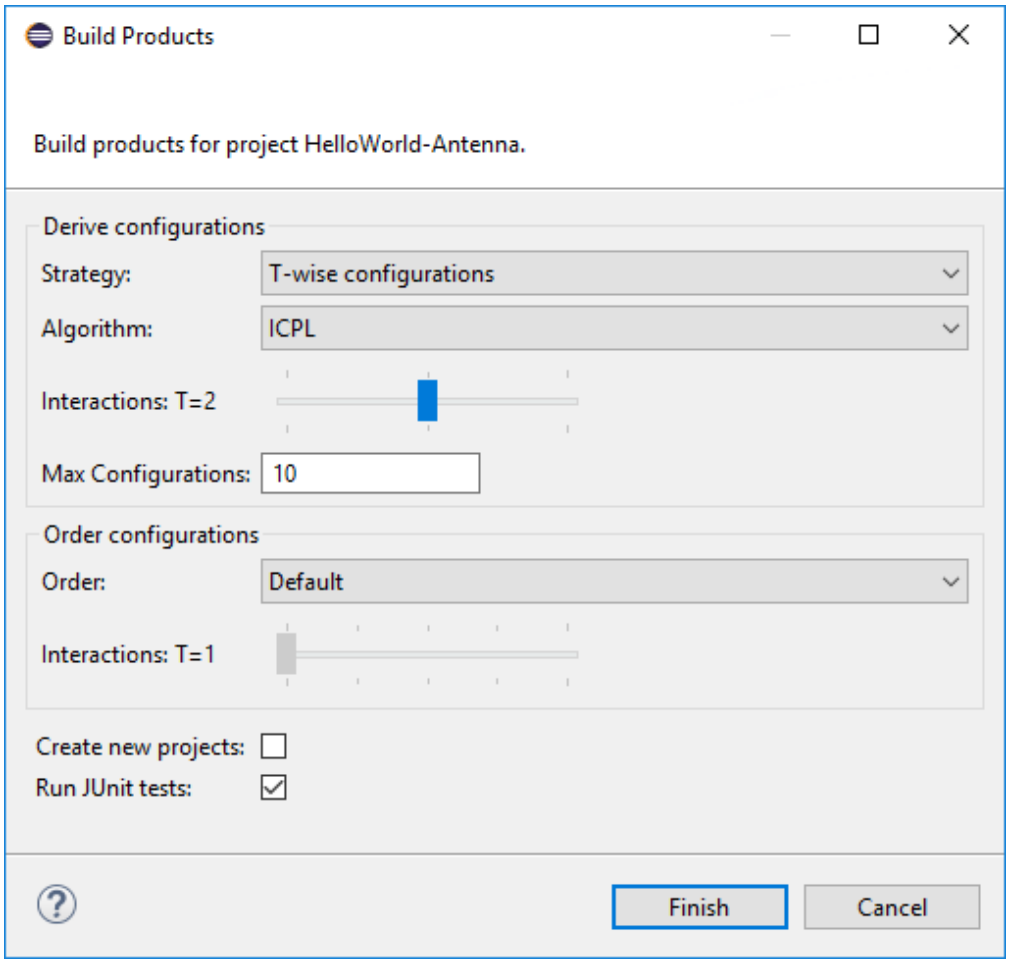


Figure A.2: Dialog to automatically derive and test products.

sampling algorithms that aim to generate a minimal set of products covering all interactions among T features, such as CASA [Garvin et al., 2011], Chvatal [Chvatal, 1979; Johansen et al., 2011], ICPL [Johansen et al., 2012a], and IncLing (cf. Chapter 3, Section 3.2). The user can specify a particular sampling algorithm using the drop-down box *Algorithm* in the dialogue *Build Products* (cf. Figure A.2). The value of T, which represent the number of involved features in a combination, can be specified using what so-called an *interaction*-bar.

Maximal Number Of Configurations The user can specify a threshold n for the maximum number of configurations that should be tested. This option is available for all generation strategies. When generating all or a random set of configurations at most n configurations are calculated. For the strategy to use user-defined configurations, only the first n configurations will be tested. For all T-Wise sampling algorithms, we can limit the number of generated configuration to n as well. Thus, a 100% T-Wise interaction coverage might not be reached. As T-Wise algorithms typically cover most

interactions in the first configurations, it is still reasonable to give a threshold for T-Wise sampling.

The generated configurations can be built into the products folder as shown in Figure A.1.② or into a distinct Eclipse project using the option *create new project* in Figure A.2.

Ordering of Generated Configurations

Optimizing the order of test cases is a good strategy to detect faults early. Therefore, the generated configurations can be *ordered* using one of three different techniques (cf.Figure A.2). Each generation strategy outputs a list of configurations in a certain order.

To improve the order in which configurations are tested, we currently provide two greedy algorithms. The first one is to order configurations by *dissimilarity* Chapter 3. The second technique aims to optimize feature *interaction coverage*. For this, the configuration that covers most feature interactions that are not already covered by previous configurations is selected. This process is continued until all configurations are tested, or all interactions are covered (i.e., the number of configurations to test may be smaller). The user can specify in which order these products are generated using the drop-down box of *Order* in the dialogue *Build Products* (cf. Figure A.2). Regarding the *interaction-based* ordering, the degree of the interaction coverage to order can be specified using the *interactions*-bar, whereas higher T require more effort for ordering.

Test Configurations

The last step for testing the system is to execute the test cases. So far, we integrated JUnit to execute tests for Java. When selecting the check box called *Run JUnit tests* in the dialogue *Build Products* (cf. Figure A.2), test cases are executed after each product is generated. As shown in Figure A.1.①, it is only necessary to annotate the test cases as known from JUnit.

To comprehend the results of testing multiple configurations (i.e., to associate the faults with configurations and to reproduce the fault), we provide a structured tree in the JUnit view as shown in Figure A.1.③. The root elements are the classes that are tested with the configurations as direct children. The leaf elements are the actual test cases. As known from the *JUnit view*, the stack trace of the failing test is shown when selecting the element. Also, the location of the fault will be opened when selecting the entry in the stack trace.

Faults are associated with configurations that cause the fault. However, aggregated results that show, which tests fail under which condition (i.e., a minimal feature selection) would improve the comprehension of the faults [Xin et al., 2008; Hoffman et al., 2009; Meinicke et al., 2016b]. For example, the test case *testStartsWithHello* in Figure A.1 fails in all configurations where the feature *Hello* is not selected. The integration of aggregated results is usually nontrivial, especially when only a subset of all configurations is tested. Thus, this improvement is subject to future work.

A.1.2 Beyond Product-By-Product Testing

Product-by-product testing allows the execution of test cases on a set of configurations. In this section, we show how FeatureIDE provides further strategies to improve testing configurable systems by avoiding redundant test executions and with support for family-based testing.

Avoid Redundant Tests

Multiple testing approaches aim to reduce the number of tests to execute [Kim et al., 2013]. However, these approaches usually require a specialized infrastructure or domain knowledge. We propose a lightweight approach to improve the time to test multiple configurations.

Unit test cases are usually designed to test a small part of the program, such as single methods or classes. When running the test case on multiple products it is unlikely to get different results, especially if the test case is not affected by variability. Thus, for product-by-product testing the test case is executed redundantly multiple times causing unnecessary overhead. Instead, the test case should rather be executed only once.

To avoid redundant test executions, we provide a Java annotation for test classes called `@NonInteracting`. While testing multiple configurations as discussed in the previous section, FeatureIDE will execute tests of an annotated class only once. However, the user needs to decide manually whether the test cases do not interact. In the example of Figure A.1, the test cases of the class `SingleRunTest` are only executed once as the JUnit view illustrates (cf. Figure A.1.③).

Family-Based Testing

Product-by-product analyses are most common in practice as standard analysis techniques can be used for the analysis of configurable systems. However, this strategy is either unsound (i.e., it may miss faults that could be found by testing other configurations) or does not scale to the high amount of configurations to test. To analyze all configurations, family-based mechanisms (also known as *variability-aware mechanisms*) have been proposed [Thüm et al., 2014a; Thüm et al., 2014; Havelund and Pressburger, 2000; von Rhein et al., 2011; Beckert et al., 2007; Meinicke et al., 2016b]. Family-based analyses exploit the fact that the analysis of two similar configurations is also similar. Thus, these redundant calculations when analyzing multiple configurations can be avoided. Family-based aim to execute these redundant parts only once to reduce the overall effort to execute all configurations.

Family-based testing requires a product simulator (a.k.a. metaproduct) that represents all configurations [Thüm et al., 2014; Apel et al., 2013c]. This simulator is a transformation of the system into a program with runtime variability, which can simulate all configurations. Currently, FeatureIDE only supports family-based analysis for feature-oriented programming with FeatureHouse [Apel et al., 2013b; Thüm et al., 2014; Apel

et al., 2013c]. The metaproduct can be generated via the project's contextmenu (i.e., *FeatureIDE* → *FeatureHouse* → *Build Metaproduct*). When building the project the metaproduct will be generated instead of a single configuration.

Different tools for family-based analysis require special types of feature model classes (i.e., a standard Java class that defines all features and valid combinations thereof). To support these different tools, FeatureIDE enables the user to choose between different model files. In particular, we support family-based testing with VarexJ [Meinicke et al., 2016b], model checking with JavaPathfinder [Havelund and Pressburger, 2000] and JPF-BDD [von Rhein et al., 2011], and theorem proving with KeY [Beckert et al., 2007]. The type of the model class can be selected via the *project's properties* (i.e., *FeatureIDE* → *Feature Project* → *Metaproduct Generation*). In the future, we aim to provide further support for other implementation techniques, especially for runtime variability, to ease the application of family-based testing and analyses.

A.2 Cluster-based prioritization

In this section, we present some raw and detailed results that support the conclusion of Section 3.4. Using feature models of different sizes, we show in Table A.1 the APFD values of the cluster-based prioritization approach with different numbers of clusters (K=5, 10, and 15) as well as the APFD values of the configuration-based prioritization and random orders. In addition, we present the p-values that show whether the differences between the random orders as well as the configuration-based prioritization approach and the cluster-based prioritization approach with K= 5 (Table A.2), K=10 (Table A.3), and K=15 (Table A.4). We show the p-values of considering different number of clusters with intra-cluster prioritization (Table A.5) and without intra-cluster prioritization (Table A.6).

FM	APFD							
	Clustering							
	K=5	K=5 (W/O ICP)	K=10	K=10 (W/O ICP)	K=15	K=15 (W/O ICP)	Random	Conf.Prio.
BattleofTanks	0.700	0.697	0.699	0.699	0.703	0.698	0.689	0.708
FM_Test	0.740	0.718	0.741	0.741	0.740	0.723	0.660	0.738
Printers	0.760	0.751	0.760	0.760	0.760	0.754	0.745	0.749
BankingSoftware	0.585	0.551	0.587	0.587	0.561	0.554	0.536	0.608
Electronic Shopping	0.705	0.702	0.697	0.697	0.686	0.682	0.668	0.702
DMIS	0.716	0.676	0.699	0.673	0.670	0.630	0.639	0.733
eCos 3.0 i386pc	0.739	0.688	0.737	0.737	0.733	0.672	0.658	0.767
FreeBSD kernel 8.0.0	0.673	0.647	0.662	0.662	0.649	0.629	0.549	0.679
Linux_2_6_28_6	0.841	0.829	0.820	0.835	0.835	0.818	0.722	0.850
AFM5K	0.348	0.342	0.347	0.351	0.348	0.344	0.312	0.354
Average	**0.681**	**0.660**	**0.677**	**0.663**	**0.669**	**0.650**	**0.618**	**0.689**

K: number of clusters; W/O ICP: without considering intra-cluster prioritization

Table A.1: Average APFD for cluster-based prioritization, random orders, and configuration-based prioritization.

FM	P-values	
	$K = 5$	
	Random	Conf. Prio.
BattleofTanks	0.296	0.717
FM_Test	0.000	0.760
Printers	0.343	0.840
BankingSoftware	0.000	0.0462
Electronic Shopping	0.000	0.434
DMIS	0.000	0.000
eCos 3.0 i386pc	0.000	0.000
FreeBSD kernel 8.0.0	0.000	0.000
Linux_2_6_28_6	0.000	0.000
AFM5K	0.000	0.000
K: number of clusters		

Table A.2: P-values of the Mann-Whitney U test between APFD values of cluster-based prioritization and random orders as well as the configuration-based prioritization.

FM	P-values	
	$K = 10$	
	Random	Conf. Prio.
BattleofTanks	0.638	0.390
FM_Test	0.000	0.646
Printers	0.313	0.782
BankingSoftware	0.000	0.0638
Electronic Shopping	0.000	0.0769
DMIS	0.000	0.000
eCos 3.0 i386pc	0.000	0.000
FreeBSD kernel 8.0.0	0.000	0.000
Linux_2_6_28_6	0.000	0.000
AFM5K	0.000	0.000
K: number of clusters		

Table A.3: P-values of the Mann-Whitney U test between APFD values of cluster-based prioritization and random orders as well as the configuration-based prioritization.

FM	P-values	
	$K = 15$	
	Random	Conf. Prio.
BattleofTanks	0.200	0.707
FM_Test	0.000	0.692
Printers	0.335	0.825
BankingSoftware	0.002	0.000
Electronic Shopping	0.039	0.000
DMIS	0.000	0.000
eCos 3.0 i386pc	0.000	0.000
FreeBSD kernel 8.0.0	0.000	0.000
Linux_2_6_28_6	0.000	0.000
AFM5K	0.000	0.000

K: number of clusters

Table A.4: P-values of the Mann-Whitney U test between APFD values of cluster-based prioritization and random orders as well as the configuration-based prioritization.

FM	P-values		
	(K=5&K=10)	(K=5&K=15)	(K=10&K=15)
BattleofTanks	0.635	0.917	0.535
FM_Test	0.908	0.874	0.958
Printers	0.981	0.970	0.991
BankingSoftware	0.848	0.035	0.018
Electronic Shopping	0.297	0.009	0.094
DMIS	0.002	0.000	0.000
eCos 3.0 i386pc	0.667	0.162	0.345
FreeBSD kernel 8.0.0	0.003	0.000	0.000
Linux_2_6_28_6	0.000	0.000	0.813
AFM5K	0.000	0.844	0.000

K: number of clusters

Table A.5: P-values of the Mann-Whitney U test between APFD values of cluster-based prioritization using different numbers of clusters with considering the intra-cluster prioritization.

FM	P-values		
	(K=5&K=10)	(K=5&K=15)	(K=10&K=15)
BattleofTanks	0.922	0.925	0.841
FM_Test	0.339	0.604	0.695
Printers	0.805	0.788	0.962
BankingSoftware	0.004	0.744	0.011
Electronic Shopping	0.116	0.001	0.104
DMIS	0.676	0.000	0.000
eCos 3.0 i386pc	0.645	0.000	0.000
FreeBSD kernel 8.0.0	0.342	0.000	0.000
Linux_2_6_28_6	0.000	0.000	0.171
AFM5K	0.000	0.012	0.000

K: number of clusters

Table A.6: P-values of the Mann-Whitney U test between APFD values of cluster-based prioritization using different numbers of clusters without considering the intra-cluster prioritization.

A.3 Architecture Definition of the Core Product of BCS

In this section, we list the architecture definition of the core product of BCS Listing A.1 as well as the deltas that are required to be applied on the core product in order to generate new products Listing A.2.

```
1
2  architecture p0 for featuremodel '/Users/.../BCS.featuremodel'{
3  signals {
4      pw_but_mv_dn boolean
5      pw_but_mv_up boolean
6      em_but_mv_left boolean
7      em_but_mv_right boolean
8      em_but_mv_up boolean
9      em_but_mv_dn boolean
10
11     pw_but_up boolean
12     pw_but_dn boolean
13     em_but_right boolean
14     em_but_left boolean
15     em_but_up boolean
16     em_but_down boolean
17
18     em_pos_left boolean
19     em_pos_right boolean
20     em_pos_top boolean
21     em_pos_bottom boolean
22     em_mv_left boolean
23     em_mv_right boolean
24     em_mv_up boolean
25     em_mv_down boolean
26
27     finger_detected boolean
28     fp_on boolean
29     fp_off boolean
30
31     pw_pos_up boolean
32     pw_pos_dn boolean
33     pw_mv_up boolean
34     pw_mv_dn boolean
35 }
36
37 components {
38     HMI {
39         ports {
40             in p_pw_but_mv_dn pw_but_mv_dn
41             in p_pw_but_mv_up pw_but_mv_up
42             in p_em_but_mv_left em_but_mv_left
43             in p_em_but_mv_right em_but_mv_right
44             in p_em_but_mv_up em_but_mv_up
45             in p_em_but_mv_bottom em_but_mv_up
46
47             out p_pw_but_up pw_but_up
48             out p_pw_but_dn pw_but_dn
49             out p_em_but_right em_but_right
50             out p_em_but_left em_but_left
51             out p_em_but_up em_but_up
52             out p_em_but_down em_but_down
```

```
53          }
54      }
55      ManPW {
56          ports {
57              in p_pw_but_up pw_but_up
58              in p_pw_but_dn pw_but_dn
59              in p_pw_pos_up pw_pos_up
60              in p_pw_pos_dn pw_pos_dn
61              in p_fp_on fp_on
62              in p_fp_off fp_off
63
64              out p_pw_mv_dn pw_mv_dn
65              out p_pw_mv_up pw_mv_up
66          }
67      }
68      FP {
69          ports {
70              in p_finger_detected finger_detected
71              in p_pw_but_dn pw_but_dn
72
73              out p_fp_on fp_on
74              out p_fp_off fp_off
75          }
76      }
77
78      EM {
79          ports {
80              in p_em_but_right em_but_right
81              in p_em_but_left em_but_left
82              in p_em_but_up em_but_up
83              in p_em_but_down em_but_down
84              in p_em_pos_left em_pos_left
85              in p_em_pos_right em_pos_right
86              in p_em_pos_top em_pos_top
87              in p_em_pos_bottom em_pos_bottom
88
89              out p_em_mv_left em_mv_left
90              out p_em_mv_right em_mv_right
91              out p_em_mv_up em_mv_up
92              out p_em_mv_dn em_mv_down
93          }
94      }
95  }
96  connectors {
97      hmi1(HMI,em_but_right,em_but_right,EM)
98      hmi2(HMI,em_but_left,em_but_left,EM)
99      hmi3(HMI,em_but_up,em_but_up,EM)
100     hmi4(HMI,em_but_down,em_but_down,EM)
101     hmi5(HMI,pw_but_up,pw_but_up,ManPW)
102     hmi6(HMI,pw_but_dn,pw_but_dn,ManPW)
103     hmi7(HMI,pw_but_dn,pw_but_dn,FP)
104
105     env1(ENV,pw_but_mv_dn,pw_but_mv_dn,HMI)
```

```
106     env2(ENV,pw_but_mv_up,pw_but_mv_up,HMI)
107     env3(ENV,em_but_mv_left,em_but_mv_left,HMI)
108     env4(ENV,em_but_mv_right,em_but_mv_right,HMI)
109     env5(ENV,em_but_mv_up,em_but_mv_up,HMI)
110     env6(ENV,em_but_mv_dn,em_but_mv_dn,HMI)
111     env7(ENV,em_pos_left,em_pos_left,EM)
112     env8(ENV,em_pos_right,em_pos_right,EM)
113     env9(ENV,em_pos_top,em_pos_top,EM)
114     env10(ENV,em_pos_bottom,em_pos_bottom,EM)
115     env11(ENV,finger_detected,finger_detected,FP)
116     env13(ENV,pw_pos_up,pw_pos_up,ManPW)
117     env14(ENV,pw_pos_dn,pw_pos_dn,ManPW)
118
119     fp1(FP,fp_on,fp_on,ManPW)
120     fp2(FP,fp_off,fp_off,ManPW)
121
122     em1(EM,em_mv_left,em_mv_left,ENV)
123     em2(EM,em_mv_right,em_mv_right,ENV)
124     em3(EM,em_mv_up,em_mv_up,ENV)
125     em4(EM,em_mv_down,em_mv_down,ENV)
126
127     pw1(ManPW,pw_mv_dn,pw_mv_dn,ENV)
128     pw2(ManPW,pw_mv_up,pw_mv_up,ENV)
129
130     }
131 }
```

Listing A.1: Architecture definition of the core product [Lity et al., 2013]

```
1  deltas {
2      //AutoPW
3      DAutomaticPW when 'Automatic Power Window' {
4          removeconnector{
5              fp1(FP,fp_on,fp_on,ManPW)
6              fp2(FP,fp_off,fp_off,ManPW)
7              pw1(ManPW,pw_mv_dn,pw_mv_dn,ENV)
8              pw2(ManPW,pw_mv_up,pw_mv_up,ENV)
9              hmi5(HMI,pw_but_up,pw_but_up,ManPW)
10             hmi6(HMI,pw_but_dn,pw_but_dn,ManPW)
11             env13(ENV,pw_pos_up,pw_pos_up,ManPW)
12             env14(ENV,pw_pos_dn,pw_pos_dn,ManPW)
13         }
14         removecomponent {
15             ManPW
16         }
17         removesignal {
18             pw_mv_dn
19             pw_mv_up
20         }
21
22         addsignal {
23             pw_auto_mv_up boolean
24             pw_auto_mv_dn boolean
```

```
25              pw_auto_mv_stop boolean
26          }
27
28
29          addcomponent{
30              AutoPW{
31
32              }
33          }
34
35          addconnector{
36              fpautopw1(FP,fp_on,fp_on,AutoPW)
37              fpautopw2(FP,fp_off,fp_off,AutoPW)
38              hmiautopw1(HMI,pw_but_up,pw_but_up,AutoPW)
39              hmiautopw2(HMI,pw_but_dn,pw_but_dn,AutoPW)
40              autopwenv1(AutoPW,pw_auto_mv_up,pw_auto_mv_up,ENV)
41              autopwenv2(AutoPW,pw_auto_mv_dn,pw_auto_mv_dn,ENV)
42              autopwenv3(AutoPW,pw_auto_mv_stop,pw_auto_mv_stop,ENV)
43              envautopw1(ENV,pw_pos_up,pw_pos_up,AutoPW)
44              envautopw2(ENV,pw_pos_dn,pw_pos_dn,AutoPW)
45          }
46
47
48      }
49      //EMH
50      DHeatable when 'Heatable' {
51      removeconnector {
52
53          em1(EM,em_mv_left,em_mv_left,ENV)
54          em2(EM,em_mv_right,em_mv_right,ENV)
55          em3(EM,em_mv_up,em_mv_up,ENV)
56          em4(EM,em_mv_down,em_mv_down,ENV)
57          hmi1(HMI,em_but_right,em_but_right,EM)
58          hmi2(HMI,em_but_left,em_but_left,EM)
59          hmi3(HMI,em_but_up,em_but_up,EM)
60          hmi4(HMI,em_but_down,em_but_down,EM)
61          env7(ENV,em_pos_left,em_pos_left,EM)
62          env8(ENV,em_pos_right,em_pos_right,EM)
63          env9(ENV,em_pos_top,em_pos_top,EM)
64          env10(ENV,em_pos_bottom,em_pos_bottom,EM)
65      }
66
67      removecomponent {
68          EM
69      }
70      addsignal {
71          heating_on boolean
72          heating_off boolean
73          time_heating_elapsed boolean
74          em_too_cold boolean
75      }
76
77      addcomponent {
```

```
78          EMH {
79
80          }
81      }
82      addconnector {
83          emhenv1(EMH,em_mv_left,em_mv_left,ENV)
84          emhenv2(EMH,em_mv_right,em_mv_right,ENV)
85          emhenv3(EMH,em_mv_up,em_mv_up,ENV)
86          emhenv4(EMH,em_mv_down,em_mv_down,ENV)
87          emhenv5(EMH,heating_on,heating_on,ENV)
88          emhenv6(EMH,heating_off,heating_off,ENV)
89          envemh1(ENV,em_pos_right,em_pos_right,EMH)
90          envemh2(ENV,em_pos_top,em_pos_top,EMH)
91          envemh3(ENV,em_pos_bottom,em_pos_bottom,EMH)
92          envemh4(ENV,em_too_cold,em_too_cold,EMH)
93          envemh5(ENV,time_heating_elapsed,time_heating_elapsed,EMH)
94          envemh6(ENV,em_pos_left,em_pos_left,EMH)
95
96          hmiemh1(HMI,em_but_right,em_but_right,EMH)
97          hmiemh2(HMI,em_but_left,em_but_left,EMH)
98          hmiemh3(HMI,em_but_up,em_but_up,EMH)
99          hmiemh4(HMI,em_but_down,em_but_down,EMH)
100     }
101     }
102
103     //AS
104     DAS when 'Alarm System' {
105         addsignal {
106             as_activated boolean
107             as_deactivated boolean
108             as_alarm_detected boolean
109             time_alarm_elapsed boolean
110             key_pos_lock boolean
111             key_pos_unlock boolean
112             as_active_on boolean
113             as_active_off boolean
114             as_alarm_on boolean
115             as_alarm_off boolean
116             activate_as boolean
117             deactivate_as boolean
118             as_alarm_was_detected boolean
119         }
120
121         addcomponent {
122             AS {
123
124             }
125         }
126
127         addconnector {
128             envhm1(ENV,activate_as,activate_as,HMI)
129             envhm2(ENV,deactivate_as,deactivate_as,HMI)
130             hmias1(HMI,as_activated,as_activated,AS)
```

```
131             hmias1(HMI,as_deactivated,as_deactivated,AS)
132
133             envas1(ENV,as_alarm_detected,as_alarm_detected,AS)
134             envas2(ENV,time_alarm_elapsed,time_alarm_elapsed,AS)
135             envas3(ENV,key_pos_lock,key_pos_lock,AS)
136             envas4(ENV,key_pos_unlock,key_pos_unlock,AS)
137
138             asenv1(AS,as_active_on,as_active_on,ENV)
139             asenv2(AS,as_alarm_on,as_alarm_on,ENV)
140             asenv3(AS,as_active_off,as_active_off,ENV)
141             asenv4(AS,as_alarm_off,as_alarm_off,ENV)
142             asenv5(AS,as_alarm_was_detected,as_alarm_was_detected,ENV)
143         }
144
145     }
146     //IM for AS
147     DASIM after DAS when 'Interior Monitoring' {
148         addsignal {
149             im_alarm_detected boolean
150             as_im_alarm_on boolean
151             as_im_alarm_off boolean
152         }
153
154         addconnector {
155             envasim1(ENV,im_alarm_detected,im_alarm_detected,AS)
156             asimenv1(AS,as_im_alarm_on,as_im_alarm_on,ENV)
157             asimenv2(AS,as_im_alarm_off,as_im_alarm_off,ENV)
158         }
159     }
160     //CLS for Manual Power Window
161     DCLSM when 'Central Locking System AND Manual Power Window' {
162         addsignal {
163             key_pos_lock boolean
164             key_pos_unlock boolean
165             cls_lock boolean
166             cls_unlock boolean
167         }
168         addcomponent {
169             CLS {
170
171             }
172         }
173
174         addconnector {
175             envcls1(ENV,key_pos_lock,key_pos_lock,CLS)
176             envcls2(ENV,key_pos_unlock,key_pos_unlock,CLS)
177             clsenv1(CLS,cls_lock,cls_lock,ENV)
178             clsenv2(CLS,cls_unlock,cls_unlock,ENV)
179             clsmanpw1(CLS,cls_lock,cls_lock,ManPW)
180             clsmanpw2(CLS,cls_unlock,cls_unlock,ManPW)
181         }
182         }
183         //CLS for Automatic Power Window
```

```
184     DCLSA when 'Central Locking System AND Automatic Power Window' {
185         addsignal {
186             key_pos_lock boolean
187             key_pos_unlock boolean
188             cls_unlock boolean
189             cls_lock boolean
190         }
191         addcomponent {
192             CLS {
193
194             }
195         }
196
197         addconnector {
198             envcls1(ENV,key_pos_lock,key_pos_lock,CLS)
199             envcls2(ENV,key_pos_unlock,key_pos_unlock,CLS)
200             clsenv1(CLS,cls_lock,cls_lock,ENV)
201             clsenv2(CLS,cls_unlock,cls_unlock,ENV)
202             clsautopw1(CLS,cls_lock,cls_lock,AutoPW)
203             clsautopw2(CLS,cls_unlock,cls_unlock,AutoPW)
204
205         }
206     }
207
208     //RCK_Ctrl for CLS with Automatic Power Window
209     DRCKA after DCLSA when 'Remote Control Key AND Automatic Power
            Window' {
210         addsignal {
211             rck_but_lock boolean
212             rck_but_unlock boolean
213             rck_lock boolean
214             rck_unlock boolean
215         }
216
217         addcomponent {
218             RCK_Ctrl {
219
220             }
221         }
222         addconnector {
223             envrck1(ENV,rck_but_lock,rck_but_lock,RCK_Ctrl)
224             envrck2(ENV,rck_but_unlock,rck_but_unlock,RCK_Ctrl)
225             rckcls1(RCK_Ctrl,rck_lock,rck_lock,CLS)
226             rckcls2(RCK_Ctrl,rck_unlock,rck_unlock,CLS)
227         }
228     }
229
230     //RCK_Ctrl for CLS with Manual Power Window
231     DRCKM after DCLSM when 'Remote Control Key AND Manual Power Window'
            {
232         addsignal {
233             rck_but_lock boolean
234             rck_but_unlock boolean
```

```
235             rck_lock boolean
236             rck_unlock boolean
237         }
238
239         addcomponent {
240             RCK_Ctrl {
241
242             }
243         }
244         addconnector {
245             envrck1(ENV,rck_but_lock,rck_but_lock,RCK_Ctrl)
246             envrck2(ENV,rck_but_unlock,rck_but_unlock,RCK_Ctrl)
247             rckcls1(RCK_Ctrl,rck_lock,rck_lock,CLS)
248             rckcls2(RCK_Ctrl,rck_unlock,rck_unlock,CLS)
249         }
250     }
251
252     //RCK SF
253     DRCKSFA after DRCKA when 'Safety Function AND Automatic Power Window
            ' {
254         addsignal {
255             door_open boolean
256             time_rck_sf_elapsed boolean
257         }
258         addconnector {
259             envrcksf1(ENV,door_open,door_open,RCK_Ctrl)
260             envrcksf2(ENV,time_rck_sf_elapsed,time_rck_sf_elapsed,
                    RCK_Ctrl)
261         }
262     }
263     DRCKSFM after DRCKM when 'Safety Function AND Manual Power Window' {
264         addsignal {
265             door_open boolean
266             time_rck_sf_elapsed boolean
267         }
268         addconnector {
269             envrcksf1(ENV,door_open,door_open,RCK_Ctrl)
270             envrcksf2(ENV,time_rck_sf_elapsed,time_rck_sf_elapsed,
                    RCK_Ctrl)
271         }
272     }
273
274     //CAP
275     DRCKCAP after DRCKA when 'Control Automatic Power Window AND
            Automatic Power Window' {
276         addsignal {
277             pw_rm_up boolean
278             pw_rm_dn boolean
279         }
280         addconnector {
281             envrckcap1(ENV,pw_rm_up,pw_rm_up,RCK_Ctrl)
282             envrckcap2(ENV,pw_rm_dn,pw_rm_dn,RCK_Ctrl)
283             rckcapautopw1(RCK_Ctrl,pw_but_up,pw_but_up, AutoPW)
```

```
284             rckcapautopw2(RCK_Ctrl,pw_but_dn,pw_but_dn, AutoPW)
285             rckcapfp1(RCK_Ctrl,pw_but_dn,pw_but_dn,FP)
286         }
287     }
288     //CAS
289     DCASM after DAS DRCKM when 'Control Alarm System AND Manual Power
            Window' {
290         addsignal{
291             rck_lock boolean
292             rck_unlock boolean
293         }
294         addconnector {
295             rckcasas1(RCK_Ctrl,rck_lock,rck_lock,AS)
296             rckcasas2(RCK_Ctrl,rck_unlock,rck_unlock,AS)
297         }
298     }
299     //CAS
300     DCASA after DAS DRCKA when 'Control Alarm System AND Automatic Power
            Window' {
301         addsignal{
302             rck_lock boolean
303             rck_unlock boolean
304         }
305         addconnector {
306             rckcasas1(RCK_Ctrl,rck_lock,rck_lock,AS)
307             rckcasas2(RCK_Ctrl,rck_unlock,rck_unlock,AS)
308         }
309     }
310
311     //AL
312     DALA after DCLSA when 'Automatic Locking AND Automatic Power Window'
            {
313         addsignal {
314             door_open boolean
315             car_drives boolean
316             car_lock boolean
317             car_unlock boolean
318         }
319         addconnector {
320             envclsal1(ENV,car_drives,car_drives,CLS)
321             envclsal2(ENV,door_open,door_open,CLS)
322             clsalsenv1(CLS,car_lock,car_lock,ENV)
323             clsalsenv2(CLS,car_unlock,car_unlock,ENV)
324         }
325     }
326     //AL
327     DALM after DCLSM when 'Automatic Locking AND Manual Power Window' {
328         addsignal {
329             door_open boolean
330             car_drives boolean
331             car_lock boolean
332             car_unlock boolean
333         }
```

```
334         addconnector {
335             envclsal1(ENV,car_drives,car_drives,CLS)
336             envclsal2(ENV,door_open,door_open,CLS)
337             clsalsenv1(CLS,car_lock,car_lock,ENV)
338             clsalsenv2(CLS,car_unlock,car_unlock,ENV)
339         }
340     }
341
342     //LED_EM
343     DLEDEM when 'LED Exterior Mirror AND NOT Heatable' {
344         addsignal {
345             em_pos_vert_bottom boolean
346             em_pos_vert_pend boolean
347             em_pos_vert_top boolean
348
349             em_pos_hor_right boolean
350             em_pos_hor_pend boolean
351             em_pos_hor_left boolean
352
353             led_em_bottom_on boolean
354             led_em_top_on boolean
355             led_em_right_on boolean
356             led_em_left_on boolean
357
358             led_em_bottom_off boolean
359             led_em_top_off boolean
360             led_em_right_off boolean
361             led_em_left_off boolean
362         }
363
364         addcomponent {
365             LED_EMB {
366
367             }
368             LED_EMT {
369
370             }
371             LED_EMR {
372
373             }
374             LED_EML {
375
376             }
377         }
378
379         addconnector {
380             emledemb1(EM,em_pos_vert_bottom,em_pos_vert_bottom,LED_EMB)
381             emledemb2(EM,em_pos_vert_pend,em_pos_vert_pend,LED_EMB)
382             ledembenv1(LED_EMB,led_em_bottom_on,led_em_bottom_on,ENV)
383             ledembenv2(LED_EMB,led_em_bottom_off,led_em_bottom_off,ENV)
384
385             emledemt1(EM,em_pos_vert_pend,em_pos_vert_pend,LED_EMT)
386             emledemt2(EM,em_pos_vert_top,em_pos_vert_top,LED_EMT)
```

```
387             ledemtenv1(LED_EMT,led_em_top_on,led_em_top_on,ENV)
388             ledemtenv2(LED_EMT,led_em_top_off,led_em_top_off,ENV)
389
390             emledemr1(EM,em_pos_hor_right,em_pos_hor_right,LED_EMR)
391             emledemr2(EM,em_pos_hor_pend,em_pos_hor_pend,LED_EMR)
392             ledemrenv1(LED_EMR,led_em_right_on,led_em_right_on,ENV)
393             ledemrenv2(LED_EMR,led_em_right_off,led_em_right_off,ENV)
394
395             emledeml1(EM,em_pos_hor_left,em_pos_hor_left,LED_EML)
396             emledeml2(EM,em_pos_hor_pend,em_pos_hor_pend,LED_EML)
397             ledemlenv1(LED_EML,led_em_left_on,led_em_left_on,ENV)
398             ledemlenv2(LED_EML,led_em_left_off,led_em_left_off,ENV)
399         }
400     }
401
402     //LED_EM
403     DLEDEMWITHH after DHeatable when 'LED Exterior Mirror AND Heatable'
            {
404         addsignal {
405             em_pos_vert_bottom boolean
406             em_pos_vert_pend boolean
407             em_pos_vert_top boolean
408
409             em_pos_hor_right boolean
410             em_pos_hor_pend boolean
411             em_pos_hor_left boolean
412
413             led_em_bottom_on boolean
414             led_em_bottom_off boolean
415             led_em_top_on boolean
416             led_em_top_off boolean
417             led_em_right_on boolean
418             led_em_right_off boolean
419             led_em_left_on boolean
420             led_em_left_off boolean
421         }
422
423         addcomponent {
424             LED_EMB {
425
426             }
427             LED_EMT {
428
429             }
430             LED_EMR {
431
432             }
433             LED_EML {
434
435             }
436         }
437
438         addconnector {
```

```
439             emhledemb1(EMH,em_pos_vert_bottom,em_pos_vert_bottom,LED_EMB
                    )
440             emhledemb2(EMH,em_pos_vert_pend,em_pos_vert_pend,LED_EMB)
441             ledembenv1(LED_EMB,led_em_bottom_on,led_em_bottom_on,ENV)
442             ledembenv2(LED_EMB,led_em_bottom_off,led_em_bottom_off,ENV)
443
444             emhledemt1(EMH,em_pos_vert_pend,em_pos_vert_pend,LED_EMT)
445             emhledemt2(EMH,em_pos_vert_top,em_pos_vert_top,LED_EMT)
446             ledemtenv1(LED_EMT,led_em_top_on,led_em_top_on,ENV)
447             ledemtenv2(LED_EMT,led_em_top_off,led_em_top_off,ENV)
448
449             emhledemr1(EMH,em_pos_hor_right,em_pos_hor_right,LED_EMR)
450             emhledemr2(EMH,em_pos_hor_pend,em_pos_hor_pend,LED_EMR)
451             ledemrenv1(LED_EMR,led_em_right_on,led_em_right_on,ENV)
452             ledemrenv2(LED_EMR,led_em_right_off,led_em_right_off,ENV)
453
454             emhledeml1(EMH,em_pos_hor_left,em_pos_hor_left,LED_EML)
455             emhledeml2(EMH,em_pos_hor_pend,em_pos_hor_pend,LED_EML)
456             ledemlenv1(LED_EML,led_em_left_on,led_em_left_on,ENV)
457             ledemlenv2(LED_EML,led_em_left_off,led_em_left_off,ENV)
458         }
459     }
460
461     //LED_EMH
462     DLEDHeatable after DHeatable when 'LED Heatable' {
463         addsignal {
464             led_em_heating_on boolean
465             led_em_heating_off boolean
466         }
467
468         addcomponent {
469             LED_EMH {
470
471             }
472         }
473         addconnector {
474             emhled1(EMH,heating_on,heating_on,LED_EMH)
475             emhled2(EMH,heating_off,heating_off,LED_EMH)
476             ledemhenv1(LED_EMH,led_em_heating_on,led_em_heating_on,ENV)
477             ledemhenv2(LED_EMH,led_em_heating_off,led_em_heating_off,ENV
                    )
478         }
479     }
480     // LED_FP
481     DLEDFingerProtection when 'LED Finger Protection' {
482         addsignal{
483             led_fp_on boolean
484             led_fp_off boolean
485         }
486
487         addcomponent {
488             LED_FP {
489
```

```
490             }
491         }
492
493         addconnector {
494             fp3(FP,fp_on,fp_on,LED_FP)
495             fp4(FP,fp_off,fp_off,LED_FP)
496             ledfp1(LED_FP, led_fp_on,led_fp_on,ENV)
497             ledfp2(LED_FP, led_fp_off,led_fp_off,ENV)
498         }
499     }
500     //LED_CLS
501     DLEDCLSM after DCLSM when 'LED Central Locking System AND Manual
            Power Window' {
502         addsignal{
503             cls_lock boolean
504             cls_unlock boolean
505             led_cls_on boolean
506             led_cls_off boolean
507         }
508         addcomponent {
509             LED_CLS {
510
511             }
512         }
513         addconnector {
514             clsledcls1(CLS,cls_lock,cls_lock,LED_CLS)
515             clsledcls2(CLS,cls_unlock,cls_unlock,LED_CLS)
516             ledclsenv1(LED_CLS,led_cls_on,led_cls_on,ENV)
517             ledclsenv2(LED_CLS,led_cls_off,led_cls_off,ENV)
518         }
519     }
520
521     DLEDCLSA after DCLSA when 'LED Central Locking System AND Automatic
            Power Window' {
522         addsignal{
523             cls_lock boolean
524             cls_unlock boolean
525             led_cls_on boolean
526             led_cls_off boolean
527         }
528         addcomponent {
529             LED_CLS {
530
531             }
532         }
533         addconnector {
534             clsledcls1(CLS,cls_lock,cls_lock,LED_CLS)
535             clsledcls2(CLS,cls_unlock,cls_unlock,LED_CLS)
536             ledclsenv1(LED_CLS,led_cls_on,led_cls_on,ENV)
537             ledclsenv2(LED_CLS,led_cls_off,led_cls_off,ENV)
538         }
539     }
540
```

```
541     //LED AutoPW without CLS
542     DLEDAPW after DAutomaticPW when 'LED Power Window AND Automatic
            Power Window AND NOT Central Locking System' {
543         addsignal {
544             led_pw_up_on boolean
545             led_pw_up_off boolean
546             led_pw_dn_on boolean
547             led_pw_dn_off boolean
548             pw_auto_mv_up boolean
549             pw_auto_mv_dn boolean
550             pw_auto_mv_stop boolean
551         }
552         addcomponent {
553             LED_AutoPW {
554
555             }
556         }
557         addconnector {
558             autopwledapw1(AutoPW,pw_auto_mv_dn,pw_auto_mv_dn,LED_AutoPW)
559             autopwledapw2(AutoPW,pw_auto_mv_up,pw_auto_mv_up,LED_AutoPW)
560             autopwledapw3(AutoPW,pw_auto_mv_stop,pw_auto_mv_stop,
                    LED_AutoPW)
561
562             ledapwenv1(LED_AutoPW,led_pw_up_on,led_pw_up_on,ENV)
563             ledapwenv2(LED_AutoPW,led_pw_up_off,led_pw_up_off,ENV)
564             ledapwenv3(LED_AutoPW,led_pw_dn_on,led_pw_dn_on,ENV)
565             ledapwenv4(LED_AutoPW,led_pw_dn_off,led_pw_dn_off,ENV)
566         }
567
568     }
569     //LED_AutoPW with CLS
570     DLEDAPWCLS after DAutomaticPW when 'LED Power Window AND Automatic
            Power Window AND Central Locking System' {
571         addsignal {
572             led_pw_up_on boolean
573             led_pw_up_off boolean
574             led_pw_dn_on boolean
575             led_pw_dn_off boolean
576
577             led_pw_cls_up_on boolean
578             led_pw_cls_up_off boolean
579
580         }
581         addcomponent {
582             LED_AutoPW {
583
584             }
585         }
586         addconnector {
587             autopwledapw1(AutoPW,pw_auto_mv_dn,pw_auto_mv_dn,LED_AutoPW)
588             autopwledapw2(AutoPW,pw_auto_mv_up,pw_auto_mv_up,LED_AutoPW)
589             autopwledapw3(AutoPW,pw_auto_mv_stop,pw_auto_mv_stop,
                    LED_AutoPW)
```

```
590
591             clsledapw1(CLS,cls_lock,cls_lock,LED_AutoPW)
592             clsledapw2(CLS,cls_unlock,cls_unlock,LED_AutoPW)
593
594             ledapwenv1(LED_AutoPW,led_pw_up_on,led_pw_up_on,ENV)
595             ledapwenv2(LED_AutoPW,led_pw_dn_on,led_pw_dn_on,ENV)
596             ledapwenv3(LED_AutoPW,led_pw_cls_up_on,led_pw_cls_up_on,ENV)
597             ledapwenv4(LED_AutoPW,led_pw_up_off,led_pw_up_off,ENV)
598             ledapwenv5(LED_AutoPW,led_pw_cls_up_off,led_pw_cls_up_off,
                    ENV)
599             ledapwenv6(LED_AutoPW,led_pw_dn_off,led_pw_dn_off,ENV)
600         }
601
602     }
603     //LED_MANPW
604     DLEDManPW when 'Manual Power Window AND LED Power Window' {
605         addsignal {
606             release_pw_but boolean
607             led_pw_up_on boolean
608             led_pw_up_off boolean
609             led_pw_dn_on boolean
610             led_pw_dn_off boolean
611             release_pw_but_dn boolean
612             release_pw_but_up boolean
613         }
614         addcomponent {
615             LED_ManPW {
616
617             }
618         }
619         addconnector {
620             manpwledapw1(ManPW,pw_mv_dn,pw_mv_dn,LED_ManPW)
621             manpwledapw2(ManPW,pw_mv_up,pw_mv_up,LED_ManPW)
622             hmiledapw3(HMI,release_pw_but,release_pw_but,LED_ManPW)
623
624             envhmi1(ENV,release_pw_but_up,release_pw_but_up,HMI)
625             envhmi2(ENV,release_pw_but_dn,release_pw_but_dn,HMI)
626
627             ledmanpwenv1(LED_ManPW,led_pw_up_on,led_pw_up_on,ENV)
628             ledmanpwenv2(LED_ManPW,led_pw_dn_on,led_pw_dn_on,ENV)
629             ledmanpwenv3(LED_ManPW,led_pw_up_off,led_pw_up_off,ENV)
630             ledmanpwenv4(LED_ManPW,led_pw_dn_off,led_pw_dn_off,ENV)
631         }
632     }
633
634     //LED_AS
635     DLEDAS after DAS when 'LED Alarm System' {
636         removeconnector {
637             asenv5(AS,as_alarm_was_detected,as_alarm_was_detected,ENV)
638         }
639
640         addsignal {
641             led_as_active_on boolean
```

```
642             led_as_active_off boolean
643             led_as_alarm_on boolean
644             led_as_alarm_off boolean
645             led_as_alarm_detected_on boolean
646             led_as_alarm_detected_off boolean
647             as_alarm_was_confirmed boolean
648             confirm_alarm boolean
649         }
650
651         addcomponent {
652             LED_ASAD {
653
654             }
655             LED_ASAC {
656
657             }
658             LED_ASAL {
659
660             }
661         }
662
663         addconnector {
664         //ASAD
665             hmiledasad1(HMI,as_alarm_was_confirmed,
                    as_alarm_was_confirmed,LED_ASAD)
666             asledasad1(AS,as_alarm_was_detected,as_alarm_was_detected,
                    LED_ASAD)
667             ledasadenv1(LED_ASAD,led_as_alarm_detected_on,
                    led_as_alarm_detected_on,ENV)
668             ledasadenv2(LED_ASAD,led_as_alarm_detected_off,
                    led_as_alarm_detected_off,ENV)
669             //ASAL
670             asledasal1(AS,as_alarm_on,as_alarm_on,LED_ASAL)
671             asledasal2(AS,as_alarm_off,as_alarm_off,LED_ASAL)
672             ledasalenv1(LED_ASAL,led_as_alarm_on,led_as_alarm_on,ENV)
673             ledasalenv2(LED_ASAL,led_as_alarm_off,led_as_alarm_off,ENV)
674             //ASAC
675             asledasac1(AS,as_active_on,as_active_on,LED_ASAC)
676             asledasac2(AS,as_active_off,as_active_off,LED_ASAC)
677             ledasacenv1(LED_ASAC,led_as_active_on,led_as_active_on,ENV)
678             ledasacenv2(LED_ASAC,led_as_active_off,led_as_active_off,ENV
                    )
679             //HMI
680             envhmi3(ENV,confirm_alarm,confirm_alarm,HMI)
681         }
682     }
683     //LED_ASIM
684     DLEDASIM after DAS DASIM when 'LED Alarm System AND Interior
            Monitoring' {
685         addsignal {
686             led_as_im_alarm_on boolean
687             led_as_im_alarm_off boolean
688         }
```

```
689
690         addcomponent {
691             LED_ASIM {
692
693             }
694         }
695         addconnector {
696             asledasim1(AS,as_im_alarm_on,as_im_alarm_on,LED_ASIM)
697             asledasim2(AS,as_im_alarm_off,as_im_alarm_off,LED_ASIM)
698             ledasimenv1(LED_ASIM,led_as_im_alarm_on,led_as_im_alarm_on,
                    ENV)
699             ledasimenv2(LED_ASIM,led_as_im_alarm_off,led_as_im_alarm_off
                    ,ENV)
700         }
701         }
702     }
703 }
```

Listing A.2: Deltas ofr BCS [Lity et al., 2013]

Bibliography

Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 421–432, New York, NY, USA, 2014. ACM. (cited on Page xvii, 21, 23, 38, 39, 56, 57, 67, 76, and 115)

Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 197–206, New York, NY, USA, 2014. ACM. (cited on Page 19)

Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. Mutation Operators for Preprocessor-Based Variability. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 81–88, New York, NY, USA, 2016a. ACM. (cited on Page 115)

Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 144–155, New York, NY, USA, 2016b. ACM. (cited on Page 19, 87, and 118)

Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 173–177, New York, NY, USA, 2016c. ACM. (cited on Page 27, 87, and 117)

Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software and System Modeling*, pages 1–23, 2016d. (cited on Page 5 and 19)

Mustafa Al-Hajjaji, Jacob Krüger, Fabian Benduhn, Thomas Leich, and Gunter Saake. Efficient Mutation Testing in Configurable Systems. In *Proc. of Int'l Workshop on n Variability and Complexity in Software Design (VACE)*, VACE '17, pages 2–8, Piscataway, NJ, USA, 2017a. IEEE Press. (cited on Page 115)

Mustafa Al-Hajjaji, Jacob Krüger, Sandro Schulze, Thomas Leich, and Gunter Saake. Efficient Product-line Testing Using Cluster-based Product Prioritization. In *Proc. of Int'l Workshop on Automation of Software Testing (AST)*, pages 16–22, Piscataway, NJ, USA, 2017b. IEEE Press. (cited on Page 19)

Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proc. of Int'l Workshop on n Variability and Complexity in Software Design (VACE)*, pages 34–40, Piscataway, NJ, USA, 2017c. IEEE Press. (cited on Page 63)

Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 532–535, New York, NY, USA, 2014. ACM. (cited on Page 1)

Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231, Washington, DC, USA, May 2009. IEEE. (cited on Page 12 and 32)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013a. (cited on Page xv, 1, 5, 6, 7, 11, 12, and 14)

Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79, January 2013b. (cited on Page 123)

Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491, Piscataway, NJ, USA, May 2013c. IEEE. (cited on Page 14, 29, 32, 55, and 123)

P. Arcaini, A. Gargantini, and P. Vavassori. Generating Tests for Detecting Faults in Feature Models. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015. (cited on Page 111 and 115)

Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1–10, New York, NY, USA, 2011. ACM. (cited on Page 31)

Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. Grammar-based Test Generation for Software Product Line Feature Models. In *Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 87–101, Riverton, NJ, USA, 2012. IBM Corp. (cited on Page 38)

Hauke Baller and Malte Lochau. Towards Incremental Test Suite Optimization for Software Product Lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 30–36, New York, NY, USA, 2014. ACM. (cited on Page 58)

Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 303–312, Washington, DC, USA, 2014. IEEE. (cited on Page 2, 18, 19, 58, 59, 63, and 84)

Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg, 2005. Springer. (cited on Page 9 and 119)

Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371, 2004. (cited on Page 1 and 119)

Bernhard Beckert, Reiner Hähnle, and Peter Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Heidelberg, 2007. (cited on Page 123 and 124)

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6): 615–708, 2010. (cited on Page 11, 99, and 119)

Gilles Bernot. Testing Against Formal Specifications: A Theoretical View. In *Proc. of the Int'l Joint Conf. Theory and Practice of Software Development on Advances in Distributed Computing (TAPSOFT)*, pages 99–119, New York, NY, USA, 1991. Springer. (cited on Page 87)

Antonia Bertolino, Paola Inverardi, Henry Muccini, and Andrea Rosetti. An approach to integration testing based on architectural descriptions. In *Proc. Int'l Conf. Engineering of Complex Computer Systems (ICECCS)*, pages 77–84. IEEE, Sep 1997. (cited on Page 68)

Antonia Bertolino, Donia Daoudagh, Said El Kateb, Christopher Henard, Yves Le Traon, Francesca Lonetti, Eda Marchetti, Tejeddine Mouelhi, and Mike Papadakis. Similarity Testing for Access Control. *J. Information and Software Technology (IST)*, 58:355–372, 2015. (cited on Page 59)

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Comm. ACM*, 53(2):66–75, 2010. (cited on Page 23)

Renée C. Bryce and Charles J. Colbourn. One-test-at-a-time Heuristic Search for Interaction Test Suites. In *Proc. Int'l Conf. on Genetic and Evolutionary Computation (GECCO)*, pages 1082–1089, New York, NY, USA, 2007. ACM. (cited on Page 60)

Renée C. Bryce and Atif M. Memon. Test Suite Prioritization by Interaction Coverage. In *Workshop on Domain Specific Approaches to Software Test Automation: Conjunction with the 6th ESEC/FSE Joint Meeting*, DOSTA '07, pages 1–7, New York, NY, USA, 2007. ACM. (cited on Page 18, 28, and 33)

Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 84–99. Springer, Berlin, Heidelberg, 2015. (cited on Page 13 and 118)

Benjamin Busjaeger and Tao Xie. Learning for Test Prioritization: An Industrial Case Study. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 975–980, New York, NY, USA, 2016. ACM. (cited on Page 49)

Xia Cai and Michael R. Lyu. The Effect of Code Coverage on Fault Detection Under Different Testing Profiles. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005. (cited on Page 88)

Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)*, 56 (10):1183–1199, 2014. (cited on Page 14, 19, and 109)

Emanuela G. Cartaxo, Patricia D. L. Machado, and Francisco G. Oliveira Neto. On the Use of a Similarity Function for Test Case Selection in the Context of Model-based Testing. *Software Testing, Verification and Reliability (STVR)*, 21(2):75–100, 2011. (cited on Page 20)

Luiz Carvalho, Marcio Augusto Guimarães, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, and Thomas Thüm. Equivalent Mutants in Configurable Systems: An Empirical Study. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 11–18, New York, NY, USA, 2018. ACM. (cited on Page 115)

Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T.H. Tse. Adaptive Random Testing: The ART of Test Case Diversity. *J. Systems and Software (JSS)*, 83(1):60 – 66, 2010. (cited on Page 20)

Vasek Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979. (cited on Page 2, 15, 27, 37, 60, 87, 99, 100, 109, and 121)

Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modelling. *Mathematical Structures in Computer Science*, 25(3):482—527, 2015. (cited on Page 63, 64, 65, 67, and 69)

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 1 and 5)

D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The Automatic Efficient Test Generator (AETG) system. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 303–309, Washington, Nov 1994. IEEE. (cited on Page 16)

M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn. Constructing Test Suites for Interaction Testing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 38–48. IEEE, May 2003. (cited on Page 87)

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 129–139. ACM, 2007. (cited on Page 2, 19, and 109)

Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proc. of the annual ACM symposium on Theory of computing (STOC)*, pages 151–158, New York, NY, USA, 1971. ACM. (cited on Page 15)

Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA, 2000. (cited on Page 1, 5, 6, and 7)

Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 10:1–10:7, New York, NY, USA, January 2014. ACM. (cited on Page 2, 18, 19, 58, 63, and 83)

Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based Similarity-driven Behavioural SPL Testing. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 89–96, New York, NY, USA, 2016. ACM. (cited on Page 2, 20, 22, 38, 52, 58, 69, and 83)

Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*, pages 15–20, NY, USA, 2012. ACM. (cited on Page 23 and 67)

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1976. (cited on Page 6 and 12)

Ibrahim K. El-Far and James A. Whittaker. *Model-Based Software Testing*, volume 1, pages 825–837. John Wiley & Sons, Inc., 2002. (cited on Page 13)

Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing Test Cases for Regression Testing. *SIGSOFT Software Engineering Notes*, 25(5):102–112, August 2000. (cited on Page 29, 52, and 78)

Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Software Engineering (TSE)*, 28(2):159–182, 2002. (cited on Page 29 and 55)

Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal (SQJ)*, 12(3):185–210, 2004. (cited on Page 2)

Lars Engebretsen. The Nonapproximability of Non-Boolean Predicates. *SIAM Journal on Discrete Mathematics*, 18(1):114–129, 2005. (cited on Page 2 and 15)

Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proc. Int'l Conf. on Information Technology:New Generations (ITNG)*, pages 291–298. IEEE, 2011. (cited on Page 18, 58, 83, and 110)

Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, volume 7328, pages 613–628. Springer, 2012. (cited on Page 38, 52, 88, and 110)

Stefan Ferber, Jürgen Haag, and Juha Savolainen. *Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line*, pages 235–256. Springer, Berlin, Heidelberg, 2002. (cited on Page 14)

Phyllis G. Frankl and Oleg Iakounenko. Further Empirical Studies of Test Effectiveness. *SIGSOFT Software Engineering Notes*, 23(6):153–162, 1998. (cited on Page 88)

Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)*, 16(1):61–102, 2011. (cited on Page 2, 11, 15, 19, 27, 37, 60, 87, 99, 100, 110, 118, and 121)

Mats Grindal, Jeff Offutt, and Sten F Andler. Combination Testing Strategies: A Survey. *Software Testing, Verification and Reliability (STVR)*, 15(3):167–199, 2005. (cited on Page 87)

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009. (cited on Page 52)

Robert J. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, January 2005. (cited on Page 32)

Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950. (cited on Page 22)

Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980. (cited on Page 17, 98, and 109)

Mary Jean Harrold. Testing: A Roadmap. In *Proc. of the Conf. on The Future of Software Engineering (FSE)*, pages 61–72, New York, NY, USA, 2000. ACM. (cited on Page 1 and 2)

Klaus Havelund and Thomas Pressburger. Model Checking Java Programs Using Java PathFinder. *Int'l J. Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000. (cited on Page 123 and 124)

Armijn Hemel and Rainer Koschke. Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 357–366, Washington, DC, USA, 2012. IEEE. (cited on Page 1)

Hadi Hemmati and Lionel Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 141–150, Washington, DC, USA, November 2010. IEEE. (cited on Page 20 and 84)

Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 327–336. IEEE, March 2011. (cited on Page 20 and 84)

Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving Scalable Model-based Testing Through Test Case Diversity. *Trans. Software Engineering and Methodology (TOSEM)*, 22(1):6:1–6:42, 2013. (cited on Page 18, 25, 65, and 69)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 62–71, New York, NY, USA, 2013a. ACM. (cited on Page 2, 18, 19, and 88)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. PLEDGE: A Product Line Editor and Test Generation Tool. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 126–129, New York, NY, USA, 2013b. ACM. (cited on Page 57 and 118)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An

Application to Similarity Testing. In *Proc. of Int'l Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 188–197. IEEE, March 2013c. (cited on Page 18, 58, 66, and 83)

Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2014a. (cited on Page 110, 111, and 115)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Engineering (TSE)*, 40(7):650–670, July 2014b. (cited on Page 2, 15, 18, 20, 25, 28, 36, 37, 57, 58, 63, 66, 69, 71, 83, 87, 88, 108, and 110)

Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing White-box and Black-box Test Prioritization. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 523–534, New York, NY, USA, 2016. ACM. (cited on Page 18, 29, 37, 55, 59, and 84)

Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware Trace Analysis. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 453–464, New York, NY, USA, 2009. ACM. (cited on Page 122)

IEEE. The Institute of Electrical and Eletronics Engineer. IEEE standard glossary of software engineering terminology. *IEEE Std.*, 610121990:3, 1990. (cited on Page 13)

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012. (cited on Page 109)

Michael Jackson and Pamela Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Trans. Software Engineering (TSE)*, 24(10):831–847, Oct 1998. (cited on Page 14 and 117)

Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Engineering (TSE)*, 37(5):649–678, September 2011. ISSN 0098-5589. (cited on Page 115)

Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 638–652. Springer, Berlin, Heidelberg, 2011. (cited on Page 2, 11, 15, 27, 87, 99, 108, 109, 118, and 121)

Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 46–55, NY, USA, 2012a. ACM. (cited on

Page 2, 11, 14, 15, 16, 19, 27, 36, 37, 38, 60, 85, 87, 88, 96, 97, 99, 100, 101, 108, 109, 118, and 121)

Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 269–284. Springer, Berlin, Heidelberg, 2012b. (cited on Page 2, 18, 19, 58, 63, 83, and 110)

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 1, 5, 6, and 8)

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, May 2008. ACM. (cited on Page 12)

Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 181–190, Pittsburgh, PA, USA, 2009. Software Engineering Institute. (cited on Page 2)

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242, Berlin, Heidelberg, 1997. Springer. (cited on Page 12 and 119)

Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57–68, New York, NY, USA, 2011. ACM. (cited on Page 109 and 110)

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 257–267, New York, NY, USA, 2013. ACM. (cited on Page 123)

Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proce. Int'l workshop on Variability & composition (VariComp)*, pages 1–6, New York, NY, USA, 2013. ACM. (cited on Page 110)

J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: An Integrated Approach to Distributed Computer Control Systems. *IEE Proc. Computers and Digital Techniques*, 130(1):1–10, January 1983. (cited on Page 32)

Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. Int'l Conf. Software Engineering (ICSE)*, New York, NY, USA, 2018. ACM. To appear. (cited on Page 97)

Michael J. Kuby. Programming Models for Facility Dispersion: The p-Dispersion and Maxisum Dispersion Problems. *Geographical Analysis*, 19(4):315–329, 1987. (cited on Page 25)

D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Software Engineering (TSE)*, 30(6):418–421, 2004. (cited on Page 38, 109, and 118)

D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, London, UK, 1st edition, 2013. (cited on Page 29 and 55)

Rick Kuhn, Yu Lei, and Raghu Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, May 2008. (cited on Page 87)

Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. Delta-Oriented Test Case Prioritization for Integration Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 81–90, New York, NY, USA, 2015. ACM. (cited on Page 18, 59, 65, 82, and 84)

Remo Lachmann, Sascha Lity, Franz E. Fürchtegott, Mustafa Al-Hajjaji, and Ina Schaefer. Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 1–10, New York, NY, USA, 2016. ACM. (cited on Page 18, 59, 65, and 84)

Miguel A. Laguna and Yania Crespo. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)*, 78(8):1010–1034, 2013. (cited on Page 1)

Daniel Le Berre and Anne Parrain. The SAT4J Library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7:59–64, 2010. (cited on Page 99)

Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 31–40, New York, NY, USA, 2012. ACM. (cited on Page 1)

Yu Lei and Kuo-Chung Tai. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *Proc. Int'l Symposium High-Assurance Systems Engineering (HASE)*, pages 254–261, Washington, 1998. IEEE Computer Society. (cited on Page 16)

Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proc. Int'l Conf. Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE, 2007. (cited on Page 2, 11, 15, 16, 87, and 100)

Zheng Li, Mark Harman, and Robert Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Trans. Software Engineering (TSE)*, 33(4):225–237, 2007. (cited on Page 29 and 55)

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, Washington, DC, USA, May 2010. IEEE. (cited on Page 12)

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, New York, NY, USA, 2013. ACM. (cited on Page 17)

Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. Delta-Oriented Software Product Line Test Models-The Body Comfort System Case Study. Technical report, Technical report, TU Braunschweig, 2013. (cited on Page xxi, 73, 75, 76, 77, 132, and 146)

Sascha Lity, Thomas Morbach, Thomas Thüm, and Ina Schaefer. Applying Incremental Model Slicing to Product-Line Regression Testing. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 3–19, Berlin, Heidelberg, 2016. Springer. (cited on Page 20, 65, and 115)

Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-Line Analysis. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 60–67, New York, NY, USA, 2017. ACM. (cited on Page 3, 20, 49, 83, and 115)

Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-oriented Software Product Lines. In *Proc. of Int'l Conf. on Tests and Proofs (TAP)*, pages 67–82, Berlin, Heidelberg, 2012. Springer. (cited on Page 65, 82, and 109)

Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-Oriented Model-Based Integration Testing of Large-Scale Systems. *J. Systems and Software (JSS)*, 91:63–84, 2014. (cited on Page 65, 73, and 82)

Roberto Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 404–407. IEEE, 2013. (cited on Page 19 and 110)

Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008. (cited on Page 76)

Malcolm Douglas McIlroy. Mass Produced Software Components. In *Proc. NATO Conf. Software Engineering*, pages 138–155. Springer, 1968. (cited on Page 6)

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with The C Preprocessor: An Interview Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 495–518, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (cited on Page 117)

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 643–654, New York, NY, USA, 2016. ACM. (cited on Page 2, 15, 17, 21, 23, 67, 87, and 109)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. FeatureIDE: Taming the Preprocessor Wilderness. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2016a. (cited on Page 119)

Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 483–494, New York, NY, USA, 2016b. ACM. (cited on Page 14, 29, 32, 55, 122, 123, and 124)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, Berlin, Heidelberg, 2017. (cited on Page 27 and 99)

Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wasowski. A Quantitative Analysis of Variability Warnings in Linux. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 3–8, New York, NY, USA, 2016. ACM. (cited on Page 17, 88, and 99)

Marcílio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762, New York, NY, USA, 2009a. ACM. (cited on Page 35 and 36)

Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 231–240, Pittsburgh, PA, USA, 2009b. Software Engineering Institute. (cited on Page 16, 39, and 110)

Akbar Siami Namin and James H. Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 57–68, New York, NY, USA, 2009. ACM. (cited on Page 88)

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, New York, NY, USA, 2014. ACM. (cited on Page 14)

Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):11:1–11:29, February 2011. (cited on Page 19 and 87)

Linda Northrop and P. Clements. A Framework for Software Product Line Practice, Version 5.0. *SEI*, 2007. (cited on Page 6)

Sebastian Oster. *Feature Model-Based Software Product Line Testing*. PhD thesis, TU Darmstadt, Darmstadt, January 2012. (cited on Page 16, 97, 98, and 109)

Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 196–210, Berlin, Heidelberg, 2010. Springer. (cited on Page 2, 11, 14, 15, 16, 19, 87, 97, and 101)

Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. A Survey of Model-Based Software Product Lines Testing. In *Model-Based Testing for Embedded System*, pages 339–381. CRC Press, Boca Raton, FL, USA, 2011a. (cited on Page 75)

Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 6:1–6:8, New York, NY, USA, 2011b. ACM. (cited on Page 75 and 82)

Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 79–82, New York, NY, USA, 2011c. ACM. (cited on Page xvi, xvii, 2, 73, 74, 78, 79, and 87)

José A. Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. Multi-Objective Test Case Prioritization in Highly Configurable Systems: A Case Study. *J. Systems and Software (JSS)*, 122:287 – 310, 2016. (cited on Page 59 and 83)

David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, December 1972. (cited on Page 12)

David L. Parnas. On the Design and Development of Program Families. *IEEE Trans. Software Engineering (TSE)*, SE-2(1):1–9, 1976. (cited on Page 6)

Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 397–401, Cham, 2016. Springer. (cited on Page 27 and 119)

Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 459–468, Washington, DC, USA, 2010. IEEE. (cited on Page 2, 11, 13, 14, 19, 87, 109, and 117)

Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal (SQJ)*, 20(3-4):605–643, 2012. (cited on Page 2, 11, 87, and 109)

William Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2006. (cited on Page 88)

Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84, September 2001. (cited on Page 32)

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, September 2005. (cited on Page 1, 5, 6, and 7)

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443, Berlin, Heidelberg, 1997. Springer. (cited on Page 12 and 32)

Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 255–264, Oct 2007. (cited on Page 29 and 55)

Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 131–140, New York, NY, USA, 2015. ACM. (cited on Page 111 and 115)

John Rice. *Mathematical Statistics and Data Analysis*. Nelson Education, 2006. (cited on Page 31)

G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing Test Cases for Regression Testing. *IEEE Trans. Software Engineering (TSE)*, 27(10):929–948, Oct 2001. (cited on Page 17, 18, 29, 55, 60, and 84)

Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 41–50, Washington, DC, USA, March 2014. IEEE. (cited on Page 29, 35, 37, 38, 52, 55, 57, and 83)

Ana B. Sánchez, Sergio Segura, JoséA. Parejo, and Antonio Ruiz-Cortés. Variability Testing in the Wild: The Drupal Case Study. *Software and System Modeling*, pages 1–22, 2015. (cited on Page 2, 18, 20, 37, 59, 63, 65, 83, and 84)

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 77–91, Berlin, Heidelberg, 2010. Springer. (cited on Page 12, 64, and 65)

Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14(5):477–495, 2012. (cited on Page 64)

Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 270–284, Berlin, Heidelberg, March 2012. Springer. (cited on Page 19 and 109)

Michaela Steffens, Sebastian Oster, Malte Lochau, and Thomas Fogdal. Industrial Evaluation of Pairwise SPL Testing with MoSo-PoLiTe. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 55–62, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1058-1. (cited on Page 118)

Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA, USA, 2002. (cited on Page 65)

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. of European Conf. on Computer Systems (EuroSys)*, pages 47–60, New York, NY, USA, 2011. ACM. (cited on Page 119)

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014a. (cited on Page 1, 13, 118, and 123)

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, January 2014b. (cited on Page 27, 99, and 119)

Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 177–186, New York, NY, USA, 2014. ACM. (cited on Page 123)

Mark Utting. *The Role of Model-Based Testing*, pages 510–517. Springer, Berlin, Heidelberg, 2008. (cited on Page 13)

Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (cited on Page 13)

Frank van der Linden. Software Product Families in Europe: The Esaps & Caféé Projects. *IEEE Software*, 19(4):41–49, July 2002. (cited on Page 6)

Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin, Heidelberg, 2007. (cited on Page 1)

Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model Checking Programs. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 3–12, Berlin, Heidelberg, 2000. Springer. (cited on Page 34)

Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java code. In *JavaPathfinder Workshop*, 2011. (cited on Page 123 and 124)

Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time-Aware Test Suite Prioritization. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 1–12, New York, NY, USA, 2006. ACM. (cited on Page 18, 29, 55, and 60)

Shuai Wang, Arnaud Gotlieb, Shaukat Ali, and Marius Liaaen. *Automated Test Case Selection Using Feature Model: An Industrial Case Study*, pages 237–253. Springer, Berlin, Heidelberg, 2013. (cited on Page 84)

David M. Weiss. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 395, Washington, DC, USA, 2008. IEEE. (cited on Page 1 and 6)

David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley, Boston, MA, USA, 1999. (cited on Page 6)

Colin Willcock. The ITEA D-MINT Project: Overview, Results, and Lessons Learnt. In *Nokia Siemens Networks*, Espoo, Finland,, 2011. Available online: "http://www.model-based-testing.de/mbtuc11/presentations/Keynote_Willcock_NSN_MBTUC2011.pdf/" (accessed on 06 July 2017). (cited on Page 1)

Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient Program Execution Indexing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 238–248, New York, NY, USA, 2008. ACM. (cited on Page 122)

Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability (STVR)*, 22(2): 67–120, 2012. (cited on Page 17, 18, 28, 29, 55, 60, and 84)

Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, ISSTA '09, pages 201–212, New York, NY, USA, 2009. ACM. (cited on Page 60)

Huihui Zhang, Shuai Wang, Tao Yue, Shaukat Ali, and Chao Liu. Search and Similarity Based Selection of Use Case Scenarios: An Empirical Study. *Empirical Software Engineering (EMSE)*, pages 1–78, 2017. (cited on Page 59)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 15.08.2017