TU Braunschweig

Master's Thesis

Re-Engineering Feature Models from Product Configurators

Author:

Nico Thiele

November 15, 2018

Advisors:

Prof. Ina Schaefer
Department of Software Engineering and Automotive Informatics

Dr. Thomas Thüm
Department of Software Engineering and Automotive Informatics

https://doi.org/10.24355/dbbs.084-201812111501-0

Thiele, Nico:
Re-Engineering Feature Models from Product Configurators
Master's Thesis, TU Braunschweig, 2018.

https://doi.org/10.24355/dbbs.084-201812111501-0

Abstract

Creating products according to own needs is omnipresent on the Internet. We choose the properties we want to have in a product from a product configurator, such as a car or printer configurator. The underlying models from the configurators are hidden. The migration of legacy configurators into feature models is meaningful to use the advantages of software configurations for product configurators. We introduce an abstraction level and assumptions for product configurators to readout information needed for a feature model. Furthermore, we develop an algorithm called FeatureDiagramSynthesizer. The algorithm consists of two parts which are finding features and adding constraints. For the algorithm, different strategies are used to create the feature model from a product configurator. Moreover, we implement the extensible algorithm FeatureDiagramSynthesizer and a prototype for the product configurator from Ricoh. Based on our developed algorithm, it is only basically possible to readout a feature models from product configurators which fulfill our assumption.

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

Zusammenfassung

Produkte nach den eigenen Bedürfnissen zu kreieren is heutzutage allgegenwärtig im Internet. Wir wählen in einem Produktkonfigurator die Eigenschaften, die wir haben wollen, für ein Produkt aus. Beispiele dafür sind Auto- oder Druckerkonfiguratoren. Die zugrunde liegenden Modelle der Konfiguratoren sind unbekannt. Die Migration von solchen Konfiguratoren in Feature Modelle is sinnvoll, um die Vorteile der Softwarekonfiguration auch für Produktkonfiguratoren zu nutzen. Dazu stellen wir Anforderungen an einen Produktkonfigurator, damit es möglich ist, Informationen für ein Feature Modell auszulesen. Außerdem entwickeln wir einen Algorithmus namens FeatureDiagramSynthesizer. Dieser Algorithmus besteht aus den zwei Teilen: Finden von Features und Hinzufügen von Constraints. Der Algorithmus nutzt verschiedene Strategien, um ein Feature Modell aus einem Produktkonfigurator zu erzeugen. Wir implementieren FeatureDiagramSynthesizer als erweiterbaren Algorithmus und einen Prototypen für den Produktkonfigurator von Ricoh. Basierend auf unserem Algorithmus ist es jedoch nur grundlegend möglich ein Feature Model aus einem Produktkonfigurator, der die Anforderungen erfüllt, auszulesen.

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

Contents

List of Figures x

List of Tables xi

List of Code Listings xiii

1 Introduction 1

2 Background 3
2.1 Product Configurators . . . 3
2.1.1 Utilization . . . 4
2.1.2 Configuration Knowledge . . . 4
2.1.3 Techniques of Product Configurators . . . 5
2.2 Software Product Lines . . . 7
2.2.1 Development of Software Product Lines . . . 8
2.2.2 Feature Model . . . 8

3 An Algorithm for Reverse Engineering Product Configurators 11
3.1 Assumptions . . . 11
3.2 Formalization of the Configuration Process . . . 14
3.3 An Algorithm for Feature Diagram Synthesis . . . 18
3.3.1 Creating the Feature Diagram . . . 18
3.3.2 Adding Constraints . . . 19
3.3.3 Strategy for Selecting Features. . . . 22
3.3.4 Algorithm for FeatureDiagramSynthesizer . . . 23
3.4 Discussion . . . 24
3.5 Summary . . . 25

4 Tool Support 27
4.1 Used Tools . . . 27
4.1.1 Selenium . . . 28
4.1.2 Geckodriver . . . 28
4.1.3 FeatureIDE . . . 28
4.1.4 Sat4j . . . 29
4.1.5 Log4j . . . 29
4.2 Implementation of a Prototype for FeatureDiagramSynthesizer . . . 29
4.2.1 Structure of the Prototype . . . 30
4.2.2 Prototype for Ricoh Configurator . . . 36

https://doi.org/10.24355/dbbs.084-201812111501-0

4.3 Discussion 39
4.4 Summary 40

5 Evaluation **41**
5.1 Subject Systems 41
5.1.1 Ricoh Configurator 41
5.1.2 Created Feature Models from the Printers 42
5.2 Experiment Setup 42
5.2.1 Research Questions 44
5.2.2 Procedure for Evaluating FeatureDiagramSynthesizer 46
5.2.3 Execution Environment 46
5.3 Results 47
5.3.1 Exactness of the Re-Engineered Feature Model 47
5.3.2 Effort for Re-Engineering a Feature Model 52
5.3.3 Anomalies in a Re-Engineered Feature Model 57
5.4 Threats to Validity 63
5.5 Summary 64

6 Related Work **67**

7 Conclusion and Future Work **69**
7.1 Thesis Conclusion 69
7.2 Future Work 70

A Appendix **73**

Bibliography **79**

https://doi.org/10.24355/dbbs.084-201812111501-0

List of Figures

2.1 Bicycle configurator steps[1] 6
2.2 Domain and Application Engineering [Cza98] 9
2.3 Example feature diagram for a human resource system 10

3.1 Example from Ricoh configurator steps[2] 14
3.2 Simple selection inside Ricoh configurator[3] 15
3.3 Example automatic selection inside Ricoh configurator[4] 16
3.4 Simple example hierarchy inside Ricoh configurator[5] 17

4.1 Attributes of ConfiguratorFeature 31
4.2 Simplified overview Class Algorithm 31
4.3 Relation between classes of the prototype 33
4.4 Process of Implementation of FeatureDiagramSynthesizer 34
4.5 Simplified Overview Structure of Adapter 36

5.1 Extract from feature diagram for Ricoh product MP_C501SP 43
5.2 MP_C501SP: False Positive vs False Negatives 48
5.3 MP_C4504ex_C6004ex: False Positive vs False Negatives 49
5.4 SP_C360SFNw_C361SFNw: False Positive vs False Negatives 49
5.5 MP_305SPF: False Positive vs False Negatives 50
5.6 SP_8400DN: False Positive vs False Negatives 51
5.7 MP_C501SP: Effort factor 53
5.8 MP_C4504ex_C6004ex: Effort factor 54
5.9 SP_C360SFNw_C361SFNw: Effort factor 55
5.10 MP_305SPF: Effort factor 55
5.11 SP_8400DN: Effort factor 56

https://doi.org/10.24355/dbbs.084-201812111501-0

5.12 MP_C501Sp: Ratio of Dead Features 57
5.13 MP_C4504ex_C6004ex: Ratio of Dead Features 58
5.14 SP_8400DN: Ratio of Dead Features 59
5.15 MP_C501Sp: Anomaly constraints 60
5.16 MP_C4504ex_C6004ex: Anomaly constraints 61
5.17 SP_C360SFNw_C361SFNw: Anomaly constraints 61
5.18 MP_305SPF: Anomaly constraints 62
5.19 SP_8400DN: Anomaly constraints 62

https://doi.org/10.24355/dbbs.084-201812111501-0

List of Tables

3.1 Overview Representation Format 12

4.1 Overview about different configurators against assumptions. 37

5.1 Exactness Levels 45

A.1 Valid configurations during tests for printer MP_C501SP 73
A.2 Valid configurations during tests for printer MP_C4504ex_C6004ex . . 73
A.3 Valid configurations during tests for printer SP_C360SFNw_C361SFNw 74
A.4 Valid configurations during tests for printer MP_305SPF 74
A.5 Valid configurations during tests for printer SP_8400DN 74
A.6 Effort for printer MP_C501Sp 74
A.7 Effort for printer MP_C4504ex_C6004ex 75
A.8 Effort for printer SP_C360SFNw_C361SFNw 75
A.9 Effort for printer MP_305SPF 75
A.10 Effort for printer SP_8400DN 75
A.11 Dead features for printer MP_C501Sp 76
A.12 Dead features for printer MP_C4504ex_C6004ex 76
A.13 Dead features for printer SP_8400DN 76
A.14 Overview Anomalies in constraints for MP_C501Sp 76
A.15 Overview Anomalies in constraints for MP_C4504ex_C6004ex 77
A.16 Overview Anomalies in constraints for SP_C360SFNw_C361SFNw . . 77
A.17 Overview Anomalies in constraints for MP_305SPF 77
A.18 Overview Anomalies in constraints for SP_8400DN 77

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

List of Code Listings

4.1 Source code for *runExtractFM* 31
4.2 Pseudo code for *simplifyModel()* 32

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

1. Introduction

Nowadays, product configurators are omnipresent on websites. The configuration of cars, laptops, or furniture are only some examples for a typical product configurator [BAKF04]. Product configurator applications are part of the research topic artificial intelligence [FZ00]. One of the aims is to enable the possibility to the customers to configure and build their own product in the way they want it. To fulfill this task, there are numerous possibilities for combinations from which the customer can choose. The underlying process is interactive. A typical example for product configurators is a car configurator. The selection of motorization, kinds of seat cover, or tires are some examples for that. Every single and valid combination of features ends up in a product inside the product configurator. The product variants of the product configurator are the individual products.
A product has to follow given rules and models to be a valid product from a product configurator [MSS01]. There can be different models and rules which result in many variable variants of the products. The rules could restrict the possible combinations. The underlying models and rules are hidden and unknown for an external viewer or customer. Furthermore, a problem is the inflexibility in the behavior of some configurators. A reason for missing flexibility are the guidelines a customer has to follow. Sometimes it also happens that users configure their product and errors occur without an error message (e.g., no feature is selectable anymore but the product is not completely configured). The reason is not always comprehensible to the customer. Product configurators can describe their own rules and models in different ways. One possibility is a model description with a real underlying model and rules behind the configurator. An alternative is a model directly coded in the source code. This results in high maintenance costs if changes are necessary. If a feature model is used, changes can be done in the model and are available for following processes. Feature models have the potential to resolve this weakness.
These properties have similarity to feature models. Feature models are one possibility to describe software product lines [KCH+90] and can be represented in different ways. Feature models are used to describe an abstraction level of a domain and the dependencies between the individual features. Furthermore, the feature models

https://doi.org/10.24355/dbbs.084-201812111501-0

describe a set of all valid feature combinations. This is called configuration which have similarities to individual products.

Goal

The goal of this thesis is to create concepts and strategies which are able to automatically reverse engineer options and rules from the configurator websites and build a feature model based on this information. In addition, the re-engineering of the feature model is another aim regardless of the underlying technique of the product configurator. The worst case is that rules and configurations are not separated inside the configurator. Even if the rules are separated, it is also possible that the product configurators are difficult to understand. Therefore the advantages of the feature models are used for analysis. The product configurator will be considered as a black box.

Furthermore the reverse engineering could be a new source to create feature models. Basic models with a large database are missing in the research. If it is possible to re-engineer a feature model from product configurators, the results can support the creation of models with a large database from a product configurator. The attributes from products can also be included in these models. A possible use case is discussed in a prior bachelor thesis *Computing Attribute Ranges for Partial Configurations with JavaSMT* [SS18] in which the models have been created manually. This master thesis should support to evaluate theories (such as in the bachelor thesis) with the help of the tool and the automatic creation of large feature models from product configurators. We want to support the overall vision to unify the product configurators and software configuration to use advantages of both sides.

Structure of the Thesis

This master thesis is structured as follows: In Chapter 2, we present the needed background knowledge which is necessary to understand this thesis. We describe how product configurators basically operate and which kinds of product configurators exist. We show techniques a product configurator uses, too. Moreover, we give an overview about software product lines with focus on feature models. In Chapter 3, we introduce an algorithm for reverse engineering product configurators. We make assumptions and formalizations which are needed for the algorithm. The algorithm is defined based on the formalizations. In Chapter 4, the implementation of a prototype is described. We present used tools for the implementation and the structure of the program. Moreover, we implement a prototype for the web configurator of Ricoh printers. We evaluate our algorithm in Chapter 5. The evaluation is done with the prototype for the Ricoh configurator. We introduce three research questions for the evaluation. Moreover, we use our test results to answer the research questions and interpret them. Threats to validity are discussed at the end of Chapter 5, too. In Chapter 6, we show related work concerning our thesis. Chapter 7 is the last chapter of this master thesis. We summarize the thesis and give an overview of future work for topics which could not discusses in this master thesis.

https://doi.org/10.24355/dbbs.084-201812111501-0

2. Background

Our main topic is to create an intersection of product configurators and software configuration. We want to unify the two parts to use advantages of both sides. Therefore, the background chapter provides the foundation which is needed to understand the following chapters. We define product configurators and what techniques exist to give an overview about them (cf. Section 2.1). Moreover, we give background knowledge for software configuration with focus on feature models (cf. Section 2.2).

2.1 Product Configurators

Product configurators and the configuration problem are part of the research topic artificial intelligence [FZ00]. The configuration problem for products can be defined as: "A product with specific properties results from combinations of a set of objects (features) and rules which must be valid" [Bri99, Sto07]. In contrast to product configurators also selectors exists. A selector does not have the possibility to create own configured products. Therefore, the customer can only choose between complete configured products [Hen04]. That suggests a configuration possibility to the customer even though it is only a selection of predefined products. In the literature, selectors are not always differentiated from product configurators. That depends on the author and use case. Sometimes the selectors are considered as product configurators and sometimes as a separate technique (e.g., [Hen04]). This does not affect our goal of that work. We include selectors into product configurators and make no difference between them because we are interested in the model behind the configurator. Product configurators enable the mass customization paradigm and moreover build an interface between customers and supplier [BAKF04]. A product configurator is a software-based expert system [HHM12]. The configuration itself is a process which is executed by the customer with the help of a product configurator. The customer acts as a co-designer or co-producer of products [BKM+10]. Mass customization means the production of products for a large market which meet individual needs of customers without loss of mass production efficiency [PR02, PIVB93]. There exist many different kinds of product configurators with different underlying techniques.

https://doi.org/10.24355/dbbs.084-201812111501-0

One configurator can use one, two, or more techniques during the configuration process. Product configurators can be divided into different groups between which an intersection is possible. We present the classification by utilization in Section 2.1.1 and by configuration knowledge in Section 2.1.2 on the basis of [PR02, BAKF04]. In Section 2.1.3 we show different concrete techniques for product configurators.

2.1.1 Utilization

We describe the classification by utilization for product configurators in this section. Therefore, we differentiate three classes: procedural systems, decision rule-based systems, and knowledge based systems. Utilization means how the configuration process is done.
Procedural systems define a starting and ending point of the configuration. There exist steps between these points which the customer has exactly to follow to create a product. The customer has to do every single step and no step can be skipped. After the last step is done a valid configuration is created [PR02].
Decision rule-based systems allow to start the configuration process at every step of the configurator a customer wants to. There exists no order which has to be followed by customers during the configuration. In contrast to procedural systems, the customer has more possibilities during the configuration process. This kind of system often supports the customer with automatic selected proposals or extensions to create a valid product [PR02].
Knowledge-based systems do not give choices to the customer. Instead, the customer gives own requirements concerning to the product (e.g., purposes, prices...) to the configurator. Afterwards the configurator generates potential solutions and products based on the requirements. The configurator uses interference techniques to find suitable products [PR02].

There exist hybrid forms of the three utilization techniques which depend on the kind of configurator. It is also possible to use only parts of the techniques.

2.1.2 Configuration Knowledge

We describe the classification by configuration knowledge for product configurators in this section. The configuration knowledge is based on the domain knowledge. Objects of that domain and their relations are described in a special way for which different classifications exist. We differentiate four classes, rule-based systems, model-based systems, decision-based systems, and case-based systems [BAKF04, PR02].
Rule-based systems. All dependencies between the features are created with *if condition then consequence* in a predefined system of rules. During the configuration, a rule-based system interactively restricts configuration options based on the system of rules. The system of rules contains dependencies between product components or attributes such as costs. For rule-based systems, it is not allowed to leave unresolved dependencies open because of the predefined system of rules. All dependencies have to be described before the configuration process starts. Therefore, a rule-based system is often realized with a procedural system (cf. Section 2.1.1) [BAKF04]. A system of rules has to be updated with changes to the product components or attributes. For large product configurators with many changes, the data management

https://doi.org/10.24355/dbbs.084-201812111501-0

of the system of rules is expensive because of the amount of possible rules. However, small product configurators or configurators with only a few changes for the configuration options can use rule-based systems because of the simple implementation. The rules can be coded directly to the configurator and no database or something similar is necessary.
Model-based systems base on a model consisting of classes of products, components and their relations. Every element for the configuration is defined by attributes and constraints. The model combined with the constraints builds the system of rules for the configuration. The customer chooses components from the model. After every selection, the selectable components are updated based on the model and constraints. In contrast to rule-based systems, constraints are not only limited to if-then rules. The data management for system of rules can be easily done in databases. The system of rules can be updated by changing the corresponding classes. Classes and components of the model can be easily added, removed, or replaced. Therefore, the maintenance of the system of rules is very flexible. However, the needed technical know-how and effort is larger than for rule-based systems [BAKF04, PR02].
Decision-based systems are object oriented and their system of rules is based upon tables (inside of databases). The tables contain information about product components and possible combinations. A configuration is based on decision matrices of the combination of tables. It is possible to start a configuration at every point of the configuration process. However, it is necessary to build database queries for every configuration step. That causes many queries for large systems which can lead to a decrease in performance. Decision-based systems are often used with decision rule-based systems [PR02].
Case-based systems try to adapt previous solutions to similar problems. These kind of system is often used if there exist more possible solutions to the requirements of a customer and the costumer does not know every single detail to find a special product. The system proposes the solution to the customer. Such a system needs a knowledge base and a control system to resolve the requests. However, the creation of a knowledge base has high costs. Moreover, the underlying decision system is very difficult to build. A potential decision system is based on artificial intelligence. Case-based systems correspond to knowledge-based systems. That kind of system opens possibilities for sale consulting because the system proposes products based on the requirements of the customer [BAKF04].

2.1.3 Techniques of Product Configurators

In Section 2.1.1 and Section 2.1.2 we defined different classifications for product configurators. The following subsection describes the most used techniques for web or online product configurators. Moreover, we connect the techniques with the classifications we described before.
Process-oriented Configurator is also called step-based configurator. This configurator type uses an interactive process to create a product. This process could consist of a single step, basic multi-step, or hierarchical multi-step. A basic multi-step process contains options in different steps or graphical containers inside one step. A hierarchical multi-step could contain inner steps but does not have to. All steps can be available at the start of the configuration or after the previous step is

https://doi.org/10.24355/dbbs.084-201812111501-0

Figure 2.1: Bicycle configurator steps[1]

completed [AHA+13, FFJ99]. This technique corresponds to procedural systems in combination with rule-based systems.

Default Configuration. The configuration process starts with a standard configuration. That means some features are preselected by the configurator. It is also possible that the customer has to choose between different default configurations at the start of the configuration process [BKM+10, SKSL+03]. This process helps the user to configure valid configurations. The default selection can be a complete valid configuration or not. The default configuration builds a basis for every configuration in the specific configurator and can reduce the mistakes a customer can do. This is due the preselection of needed features for the configuration which could be missed by a user. An extract from the default selection from the bicycle configurator is shown in Figure 2.1. For example, the feature *Integrated Lighting* is preselected. The configuration knowledge can be rule-based, model-based, or decision-based. Case-based is not meaningful because the purpose of a default configuration is providing a basis on which the customer creates his product. A case-based system creates complete products.

Validation of Configuration. The most product configurators validate the actual configuration after every activity a customer does, such as selecting or deselecting a feature. A configurator has to use configuration rules for the validation. The product configurator evaluates the configuration rules at runtime [BKM+10, FFJ99]. There exist configurators which check the rules at the end of the configuration process [AHA+13]. They allow to work with invalid states. Therefore, it is possible that

[1] cf. https://www.missionbicycle.com

https://doi.org/10.24355/dbbs.084-201812111501-0

the configuration is invalid at the end of the process. That technique corresponds to systems which have a set of system of rules, such as rule-based, model-based, or decision-based systems.
Interactive Resolution. This configurator technique is also called decision propagation [AHA+13]. This technique will react if a conflict state is reached once in the configuration process or if after a selection of a feature more selections are necessary. The configurator tries to resolve the conflict with an automatic resolution or an interactive process with the user, e.g., by giving proposals to resolve the issue [Sto07, TKS18, SKSL+03]. Knowledge-based systems could support the automatic resolution. The interactive resolution can be done based on procedural systems.
Hiding Invalid Combinations. Features are hidden if a selection of a feature before would cause conflicts with the other features. That prevents a user for configuring invalid products [Sto07, TKS18]. Rule-based, model-based, and decision-based systems are considered for that technique because they use a system of rules from which the invalid combinations can be found out.
Automatic Reconfiguration. During the configuration process the configurator can automatically reconfigure the actual selection after a feature has been selected or deselected. It is also possible that the customer wants the configurator to finish the rest of the configuration. An automatic reconfiguration can happen due to underlying constraints inside of the configurator [TKS18, AHA+13]. Moreover, it is possible that the configurator uses components of a knowledge-based system. Therefore, this technique could base on case-based systems. However, all other configuration knowledge systems can be used.
Alternatives of Compound Options. Nearly every product configurator uses alternative options. It can happen that there exist dependencies between different options. For that reason, groups are built. The groups combine the dependencies and options into one alternative category [TKS18].
Automatic Deselection. The automatic deselection is often used in groups where alternative options are present. If a feature is selected, alternative features to the selected are automatically deselected. Simple conflicts which may occur in alternative groups are automatically resolved [TKS18]. The selection and deselection can happen without considering the system of rules. The simplest implementation is if a feature is selected of an alternative group the other features of that group have to be deselected.

2.2 Software Product Lines

In this section, we define software product lines (SPL) and how software product lines are developed. Furthermore, we show what feature models are and how they can be represented.

The software development of specialized software or software for embedded systems is more and more important. For modern software, there are many requirements which have to be fulfilled to be the solution for one special customer. The customer wants to have the best possible software variant. Therefore, mass customization

https://doi.org/10.24355/dbbs.084-201812111501-0

do not only effect product configuration but also software products. Many software variants lead to many new developments, high costs, and development times. Software product lines can solve the mass customization problem for software development. SPL are part of variability modeling [KCH+90, ABKS13].
Software product line is a set of program variants with a common set of features. A feature is a characteristic of a software system and is used in software product lines to specify commonalities and differences of the developed products [ABKS13]. The program variants are tailored to a special market segment. That market segment is called domain. The goal of a software product line is to reuse common software artifacts to build program variants for a domain [ABKS13, Lü12]. The goal of software product lines is more efficiency, faster development, less costs, and individual adjustments for special requirements.

2.2.1 Development of Software Product Lines

The development of a software product line consists of two main parts: domain engineering and application engineering. The development does not follow classical development models such as waterfall model or spiral model [BH00]. Domain engineering composes of domain analysis, domain design, and domain implementation and is executed by domain experts. In the domain analysis, the set of reusable requirements are defined. That means a scoping to the domain is done and relevant features should be found. The domain design develops the architecture for the system. The reusable artifacts are implemented in the domain implementation. The result is not a finished distributable software. However, the result composes of commonalities between the software products and variability of the product line. A feature model (cf. Section 2.2.2) is one possible result of domain engineering. Application engineering uses the information of domain engineering to develop a software specifically for a use case [Cza98, PM08]. Figure 2.2 shows the context of domain engineering and application engineering for developing a software product line.

2.2.2 Feature Model

Feature modeling is a process at which features are connected to a domain and a feature model is the result [AK09, KCH+90]. With the help of feature models, products of a specific domain can be created. The products result from selections and combinations of features. The combination of features is called configuration which corresponds to products of a software product line. Features are normally referred to their name. Moreover, features can contain more information, such as attributes or descriptions. A feature model describes elemental abstractions from a domain (e.g., features) and their relations [ABKS13]. A combination of different features is called configuration. A configuration is valid if and only if the configuration fulfills all rules and relations of a feature model. Otherwise the configuration is invalid or incomplete [ABKS13]. A feature model can also be defined by the set of all possible configurations [CHE05], but this is only theoretically possible. With only 33 optional and independent features it is possible to create an individual configuration for every human on earth. Therefore it is impossible or impractical to describe all possible configurations because there exist so many configurations [HNA+17]. The space of

https://doi.org/10.24355/dbbs.084-201812111501-0

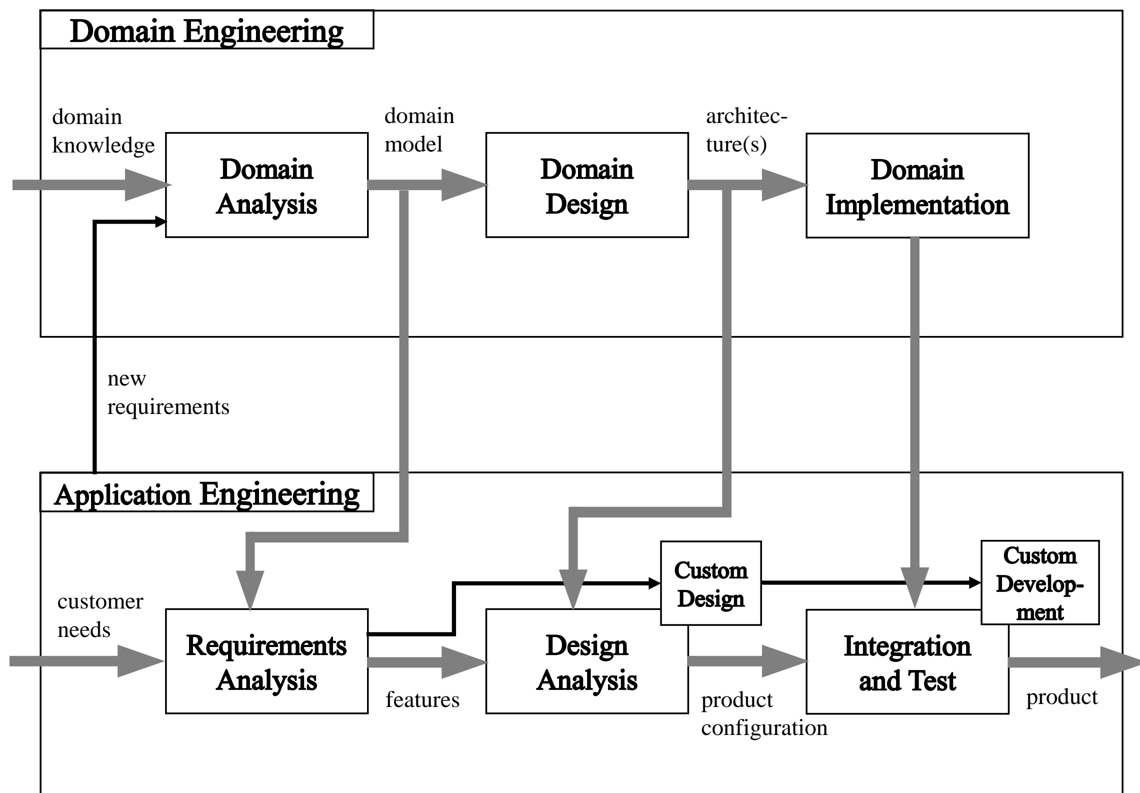

Figure 2.2: Domain and Application Engineering [Cza98]

variants is too large to use all configurations in practice. A feature could be abstract, mandatory, or optional. Abstract features are used to group up features and have no influence on the configuration [ABKS13]. Mandatory features have to be selected for all valid configurations. They are needed for every product. Optional features can be selected but it is not necessary for a valid configuration. Possible relations are *alternative-groups* from which only one feature can be selected for a configuration and *or-groups* from which more than one feature can be selected [KCH+90]. The relations are affected by optional or mandatory features inside the groups. It can happen that a feature is not selectable in any valid configuration of a feature model. This kind of feature is called dead feature.
Feature models can be represented in different ways. Possible representations are textual notations such as *Feature Description Language* [VDK01] or feature diagrams. A feature diagram visualizes features and their relations. Therefore, it uses a graphical presentation and reflects hierarchical structures of features as a tree [KCH+90] as shown in the example Figure 2.3. A feature is a node inside that tree. These structures are represented in a child-parent relation which means the selection of a child requires the parent selection. Child nodes represent the different relations such as alternative-group or or-group [Bat05]. There could exist constraints between features which are in different nodes and branches. These dependencies can be added to the diagram as arrows or text. They are called cross-tree constraints because they may span large parts of the feature diagrams [ABKS13]. An example for a cross-tree constraint is $SuccessionPlanning \implies CurriculumVitae$ in Figure 2.3. A feature model could contain redundant constraints. A constraint is redundant if removing the constraint from the feature model will not change the

https://doi.org/10.24355/dbbs.084-201812111501-0

validity of the configuration [KAT16] It is possible that a constraint is a tautology, e.g., $A \implies \neg A$. Tautologies are a subset of redundant features and can also be removed from a model.

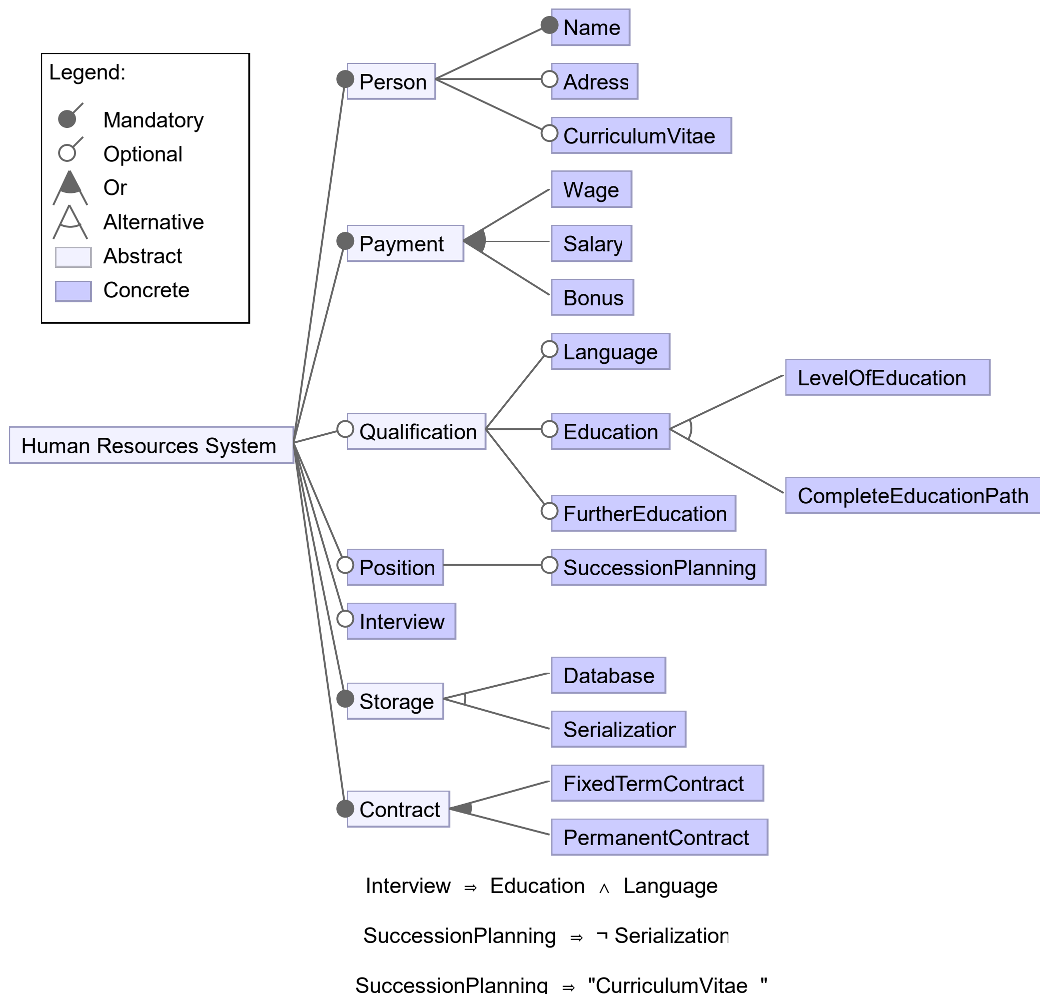

Figure 2.3: Example feature diagram for a human resource system

All components of a feature diagram can be transformed into propositional formula. Batory [Bat05] describes a possibility how this is done. The propositional formula of the feature diagram connected with all cross-tree constraints allows the representation of a feature model as one large propositional formula. This formula can be analyzed by tools such as SAT solvers. SAT is an abbreviation for satisfiability or boolean satisfiability problem. The SAT problem has the complexity class NP. A SAT solver gives an answer whether a formula is satisfiable or not. A formula in conjunctive normal form is given as input to a SAT solver [SS18]. It is possible to check the validity of the complete model or of certain configurations [ABKS13]. If the propositional formula of the feature model is unsatisfiable, there exist no valid configuration. Moreover, it is possible to find out whether a feature is mandatory or dead with the help of SAT queries. It is also possible to check partial configurations with SAT solvers.

https://doi.org/10.24355/dbbs.084-201812111501-0

3. An Algorithm for Reverse Engineering Product Configurators

This chapter describes strategies to create a feature model from a product configurator with focus on web configurators. We consider the product configurator as a blackbox. The overall vision is to unify product configuration and software configuration (e.g., feature models) and use properties and advantages from both sides. In this chapter, we define assumptions with respect to product configurators and a representation format we want to achieve (cf. Section 3.1). On that basis, we formalize the configuration process and define models, sets, and states (cf. Section 3.2). We develop an algorithm to synthesize a feature diagram from the predefined models, sets, and states (cf. Section 3.3). The assumptions will also show limitations and applicability of the developed algorithm. The algorithm is divided in a static phase and a dynamic phase. Afterwards there is a discussion about the algorithm and its functions and limitations (cf. Section 3.4).

3.1 Assumptions

We have to define assumptions for the algorithm because there exist numerous kinds of product configurators with different underlying techniques. For example, an alternative group is represented with radio buttons in one configurator and in another with pictures. In the end, it is the same type of feature dependency which has to be recognized. Therefore, we have to simplify the complexity and not every configurator can be used. The developed algorithm can be used for the configurators which fulfill our assumptions. We also define the representation format which is the result of the algorithm. Moreover, our assumptions define allowed and not allowed behavior or functionalities of a product configurator.

Representation Format

A feature model can be represented in different ways. There are some requirements for the representation format. It is necessary that the format is machine readable,

https://doi.org/10.24355/dbbs.084-201812111501-0

has a good scaling, and can be converted into other formats. Machine readability is needed because our algorithm should readout configurators automatically. Scaling means that the format also works good and is usable for large product spaces and resulting models. Without good scaling we cannot use our algorithm for large configurators and lose the possibility to re-engineer large models. Our result can be used by different fields of application if we can convert our output format into different other formats. The reverse engineering has the aim to use feature diagrams as representation format. This representation format has advantages over other formats such as natural language, enumerations, or propositional logic. Natural language can not be evaluated by machines because of its ambiguity. A representation with enumerations is only meaningful with a small number of variants. However, product configurators have a huge product space with numerous configurations. Propositional logic in comparison to feature diagrams cannot be clearly converted into all other formats. A feature diagram can be easily converted into a propositional formula but not the other way around. It is very difficult to convert propositional formula into a feature diagram because a formula is not unique in every case inside a feature diagram. The feature diagram is more flexible and better readable for users than other representation formats. Table 3.1 gives an overview about properties of different representation formats.

	Natural Language	Enumerations	Feature Diagram	Propositional Logic
Uniqueness	no	yes	yes	yes
Scalability	very bad	bad	good	good
Convertability	very bad	bad	very good	good

Table 3.1: Overview Representation Format

Assumption 1

All features can be recognized at the beginning of the process. That means there are no hidden features during the configuration process and no other features are added.

Assumption 2

It is possible to find out the selection status of a feature regardless of which kind of feature we have, such as pictures (icons) or checkboxes. That also means the information about the selection status must be machine readable.

Assumption 3

A radio button and drop-down menu are handled in the same way. One choice out of at least one item has to be made for both. If a drop-down menu occurs inside a configurator it must be a one selection menu. A multi selection from a drop-down menu is not allowed.

https://doi.org/10.24355/dbbs.084-201812111501-0

Assumption 4

After selecting a feature, it can happen that a dialog opens. Additional selections can be made or have to be made in this dialog. That depends on the kind of dialog. There exist numerous different dialogs which are very hard to unify. The dialog could give the possibility to go back or forth, make requirements, or may open another dialog. Therefore, we cannot completely support a dialog but give a basic support and allow dialogs. Hence, we just use the information from the configurator which opens the dialog and do not follow the dialog context. A dialog could have several steps and choices between them. It cannot be differentiated if one of the steps has dependencies to previous steps where different choices were made.

Assumption 5

During the configuration it may happen that the underlying model changes. We cannot identify the moment when this happens. Therefore, we suppose that the model will not change while the algorithm is working. The configuration process is a deterministic process. The same selections and clicks always result in the same configuration if we start at the same starting point.

Assumption 6

It has to be possible to implement a function which is able to check the validity of the configuration. We call that function *checkValidity*. The function can always be used by the algorithm.

Assumption 7

A choice between a minimum or maximum selection number can be transferred into a feature model. A minimum or maximum selection means that a certain number x ($x \in \mathbb{N}$) of selections has to be made from a group of features with number of features y ($y \in \mathbb{N}$). For minimum selections for x is valid ($x \geq 1$) and for maximum selection for x is valid $x < y$. It is not meaningful for a maximum selection to select all features from a group ($x = y$) because there is no need to define a maximum. There are many constraints necessary to represent these selection choices . The bigger the number of features from which the minimum or maximum selection has to be made, the more constraints are necessary to recreate it inside a feature model. The amount of constraints grows exponentially. Therefore, we cannot consider minimum and maximum number of selection for large configurators but only for small configurators.

Assumption 8

The basic algorithm will only work for configurators which have only one step or all possible features inside this one step, such as in Figure 3.1 .That means all possible configurations can be configured inside this one step. The algorithm has to be extended with more functionalities to work with more steps.

https://doi.org/10.24355/dbbs.084-201812111501-0

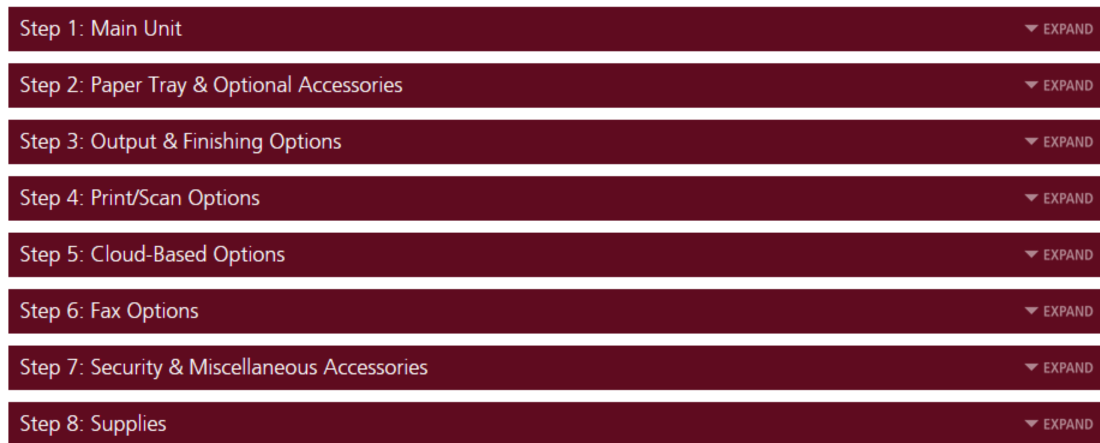

Figure 3.1: Example from Ricoh configurator steps[1]

Assumption 9

Manual deselected features do not occur during the configuration. For some feature models this option is possible. For the re-engineering process from product configurators it is not used. Nearly all product configurators do not use manual deselection.

Assumption 10

It has to be allowed to access the website with a bot. One goal of the thesis is to implement a prototype. That prototype acts like a bot on a website. The right to access websites with bots is granted by the ”robots.txt” file. This file is contained in the root directory of a website. The ”robots.txt” have to allow to access all needed directories of the website which are used by the product configurator. If the ”robots.txt” is missing, we assume that an access with bots is always possible. However, a website with an existing ”robots.txt” should always be preferred.

There are hundreds of different product configurators which use many different representations for options while configuring a product. Therefore, we made assumptions for the product configurators and the algorithm. These will build our basis for the algorithm. Our developed algorithm will work for configurators which satisfy the preconditions.

3.2 Formalization of the Configuration Process

In this section, we describe the formal needs for the algorithm for reverse engineering the feature model from product configurators. We start with defining different sets. Afterwards we define states which the algorithm will use. The algorithm will work on an abstraction level. For that, we introduce two mandatory and four optional sets. The sets follow from our assumptions. These sets build the first abstraction level and refer to the features. Furthermore, we abstract different components from a web product configurator.

[1]cf. https://ricohconfigurator.com/configure.php

https://doi.org/10.24355/dbbs.084-201812111501-0

- We consider radio buttons and drop-down menus as the same functionality as *assumption 3* assumes. We abstract them to the same information for our algorithm.
- Multiple steps inside one step, such as in Figure 3.1 can be combined to one step. That means we open all steps if necessary and consider them as only one step.

We define the following sets:

- Set of all features F. This mandatory set contains all features from the configurator regardless which features are selected. $F = \{x \mid x \text{ is a feature of a specific configurator}\}$. This set follows from *assumption 1*.
- Set of selected features S. This mandatory set contains all selected features at a specific moment. There is no distinction between which features have been selected manually or automatically by the configurator. $S \subset F$ and $S = \{x \mid x \text{ is a selected feature}\}$. This set follows from *assumption 1 and 2*. A set of non selected features can be described as $F \setminus S$. The example Figure 3.2 shows a simple selection state of the configuration. The feature MP C4504ex would be part of S.

Item/Description

MP C4504ex

- Output Speed (Letter): 45-ppm
- Average Monthly Volume: 10,000 impressions/month
- Maximum Monthly Volume: 50,000 impressions/month
- Power Requirements: 120V-127V, 60Hz
- Weight: 228 lbs. (103.4 kg)
- W x D x H (inches): 23.1 x 27 x 37.9
- W x D x H (mm): 587 x 686 x 963

Note:
The DOSS does not overwrite the HDD for the Color Controller E-23C.

MP C6004ex

- Output Speed (Letter): 60-ppm

Figure 3.2: Simple selection inside Ricoh configurator[2]

- Set of alternative features A. This optional set contains sets of alternative features. $A = \{x, y \mid x,\ y \in F \text{ and } x \neq y,\ x \text{ and } y \text{ are features which are alternative}\}$. This set could be empty if a configurator does not contain any alternative feature. This set follows from *assumption 1, 2, and 3*.
The default value of A is the empty set $\emptyset$. An example is shown in Figure 2.1 with the feature group "*Handlebar*". The features "Casual", "Upright", "Classic", and "Powerful" are alternatives.

[2]cf. https://ricohconfigurator.com/configure.php?model=MP_C4504ex_C6004ex

https://doi.org/10.24355/dbbs.084-201812111501-0

- Set of automatically selected features S_A. This is an optional set only containing features which are automatically selected after selecting a feature x. $S_A \subseteq S$. This set follows from *assumption 1 and 2.* There exist two possibilities to fill the set. Firstly, the configurator gives the information directly to the customer/program, e.g., through a dialog. Secondly, it is possible to calculate the elements of that set by using the set S. However, this only will be done if the set cannot be filled automatically. We describe the calculation in Section 3.3.
  The default value of S_A is the empty set $\emptyset$. An example is shown in Figure 3.3. If the feature "Paper Feed Unit PB3250" is selected the feature "Caster Table Type M3" will be selected automatically.

Item/Description

Paper Feed Unit PB3250

Recommended Paper Feed Unit for environments that prefer their system at the lowest height possible, making it easier for all users to access the device (Section 508).

Provides an additional 550 sheets.

Paper Sizes up to 12" x 18".

Paper Weights up to 80 lb. Bond/166 lb. Index (300g/m^2).

Weight: 24 lbs. (10.9 kg)

W × D × H (inches): 23.1 × 27 × 4.7

W × D × H (mm): 587 × 686 × 119

Note:
1. By choosing this option, Caster Table Type M3 will be added to your configuration.
2. Paper Feed Unit PB3250 cannot be installed with Paper Feed Unit PB3240, Paper Feed LCIT PB3260, LCIT RT3030, Cabinet Type F, Finisher SR3210, Finisher SR3230, Booklet Finisher SR3220, Booklet Finisher SR3240, or any related options.

Caster Table Type M3

The caster table must be added to the configuration when the PB3250 Paper Feed Unit is selected. It provides a platform with wheels to easily move the device.

Figure 3.3: Example automatic selection inside Ricoh configurator[3]

- Set of automatically deselected features D_A. This optional set contains automatically deselected features after selecting a feature x. $S_D \subseteq F \setminus S$. This set follows from *assumption 1 and 2.* There exist two possibilities to fill the set. Firstly, the configurator gives the information directly to the customer/program, e.g., through a dialog. Secondly, it is possible to calculate the elements of that set by using the set S. However, this only will be done if the set cannot be filled automatically. We describe the calculation in Section 3.3.
  The default value of S_D is the empty set $\emptyset$. If one of the two selected features in the example Figure 3.3 is deselected the other will deselect automatically.

[3]cf. https://ricohconfigurator.com/configure.php?model=MP_C4504ex_C6004ex

https://doi.org/10.24355/dbbs.084-201812111501-0

Step 2: Controller Options

Item/Description

Interactive Whiteboard Controller Type 1

Ricoh Business Controller providing whiteboarding and collaborative features.

Note:
Interactive Whiteboard Controller Type 1 cannot be installed with Windows Controller or any related options.

Windows Controller

Windows® 10 Open controller.

The Windows Open Architecture controller is designed to provide users with the ability to install preferred apps and cloud solutions to meet their requirements.

This controller comes standard with Windows 10 Pro, Ricoh IWB lt software, Ricoh UCS and Ricoh Advanced software.

Note:
1. Ricoh IWB lt software has limited features.
2. Windows Controller cannot be installed with Interactive Whiteboard Controller Type 1 or any related options.

Figure 3.4: Simple example hierarchy inside Ricoh configurator[4]

- Set of hierarchies H. This optional set contains information about hierarchical structures, such as parent and child. $H = \{(x, y) \mid x \in F$ and $y \in F \cup root$, x is child of y$\}$. This set contains at least one hierarchy level which means that all features are under the same root and cannot be empty. An example is shown in Figure 3.4. The features ”Interactive Whiteboard Controller Type 1” and ”Windows Controller” are children of the feature ”Controller Options”.

We must differentiate between information which can be gained in a static way or which needs interaction with the configurator. Static means that the information can be readout directly from the configurator without any selection. A dynamic information is gained after an interaction. Usually, the interaction inside configurators is *clicking* to select or deselect a feature f.
The algorithm will use selections, configuration states, and state transitions of the configurator. Moreover our predefined sets are used. Therefore, we define the configurator c as a three tuple (F, A, H) and the state σ_i as a three tuple(S, S_A, S_D) and the state transitions as δ_j. A configurator is always in a state during the reverse engineering process. Going from a state σ_i into a new state σ_{i+1} is done by an interaction (δ). Because of the static configurator c it is not necessary to readout c in every new state σ_i because the sets F, A, H are also static. F always contains all features of the configurator in every state. A will be readout once at the beginning and does not change during the process. The hierarchy H will be built once at the beginning of the configuration process. It is not necessary to rebuilt H in every state because H will not change. The state σ_i is dynamic and the contained sets change in every new state. During a state transition δ it is possible that S, S_A and S_D all change or only one set changes. Therefore, every tuple σ_i will be build dynamically.

[4]cf. https://ricohconfigurator.com/configure.php?model=D5520

https://doi.org/10.24355/dbbs.084-201812111501-0

We define the sets which are valid in a successor state σ_{i+1} as:

- $S \implies S'$
- $S_A \implies S'_A$
- $D_A \implies D'_A$

For example, S' can be created as follows: $S' = S + \{f\}$ with f as a new selected feature and $f \notin S$.

3.3 An Algorithm for Feature Diagram Synthesis

After we made our assumptions and formalizations, we can start defining the algorithm. In this section, we specify our algorithm **FeatureDiagramSynthesizer** for feature diagram synthesis. We define two steps for the algorithm, *creating the feature diagram* and *adding constraints*. We also present selection strategies and simplifications for the feature model. At the end of that section, we show FeatureDiagramSynthesizer.
We divide the re-engineering process into two main steps. The first step creates the feature diagram and its hierarchies and the second step adds the cross-tree constraints to the diagram to built the feature model M. Furthermore, it must be checked continuously whether the analyzed configurator is in a valid state or not. Therefore, we use the function *checkValidity* which checks the validity of the current configuration and returns *true* or *false*. If an invalid state is reached, it will be necessary to go back to a valid state. If we continue with invalid states, our created feature model could contain wrong constraints. An exception has to be made for the beginning of the process. There are product configurators which start with invalid configurations. That could happen to configurators which do not use default configurations. In that case the customer must select the first features until the configuration is valid for the first time. FeatureDiagramSynthesizer considers that possibility. We allow invalid states at the beginning until the configuration is valid for the first time. FeatureDiagramSynthesizer generally analyzes states and examines the changes after a state transition. Analyzing states helps us to find all features and constraints between them. We use the states and state transitions to find the constraints.

3.3.1 Creating the Feature Diagram

In this section, we describe the static part of FeatureDiagramSynthesizer which creates the feature diagram.
The first step to create the feature model M is to readout the configurator c with the three tuple (F, A, H). Static information can be gained without interactions in the configurator. FeatureSynthesizer checks which features exist regardless of selected or deselected features. All found features are collected inside the set of all features F. Features f could be grouped up under another feature g. All features f belonging to g are considered as child features from g. These groups occur among other things in step-based configurators as in Figure 3.1. In that example, the groups correspond

https://doi.org/10.24355/dbbs.084-201812111501-0

to one of the steps. This can be combined to tuples of the set H. If no hierarchy is recognizable, all features have the *root* element as parent. The next step tries to find alternative groups and fill set A. As defined before radio buttons and drop-down menus are mapped to alternative groups. This is also done without any interaction with the configurator. All remaining features which are not in A are considered as optional features at this moment. After creating all sets for c we build the feature diagram from c. If the analyzed configurator uses default configurations we can fill S with all initially selected features.

3.3.2 Adding Constraints

We present in this section how constraints are added to the feature diagram. We consider the different possibilities to generate the constraints.
The second step of FeatureDiagramSynthesizer is an interactive (dynamic) step. It tries to find the constraints by selecting or deselecting additional features f to the actual configuration. For a new selected feature f, it has to be valid that the feature f is element of the set of deselected features ($f \in F \setminus S$). After the selection, f is part of the selection set of the new state ($f \in S'$). We use the change in tuples from status σ_i to σ_{i+1} to gain information about constraints. It is impossible or impractical to check all possible configurations to get the constraints because there exist so many configurations (cf. Section 2.2.2) [HNA+17]. Therefore, we rely on sample strategies. We discuss possible strategies for (de-)selections of additional features later in this section. Our algorithm works independently from the selection strategy. Thus, the strategy is exchangeable. The most constraints we will find are implications or exclusions of features. This depends on the selection strategy. The algorithm uses the function *checkValidity* to check whether a configuration is valid or not. If *checkValidity* returns false, we have two options. The function could return which features are missing or it returns false for an invalid configuration. If information about missing features is returned, we use it as a constraint to the model M. The constraint is an implication. The left side is a conjunction of all selected features (cf. Equation 3.1) of the actual state.

$$\bigwedge_{i \in S'} i \tag{3.1}$$

The right side contains all needed features N as disjunction (cf. Equation 3.2) because we do not know which of the needed features is required and in which relation they are. Therefore, we consider them all as optional. That does not excludes alternative or and relations between the needed features.

$$\bigvee_{j \in N} j \tag{3.2}$$

Our algorithm will react to the changes which occur due to the selection strategies. Therefore, it exists a function d_s *determineState* which needs the information of the previous state $\sigma_{i-1}(S, S_A, D_A)$ and returns the sets S', S'_A, and D'_A of the current state σ_i. That function is triggered by a transition δ if the actual configuration is valid. The function d_s fills S' by returning all selected features. As in Section 3.2 described the sets S_A and D_A can be filled in two ways. The function d_s always tries

https://doi.org/10.24355/dbbs.084-201812111501-0

to fill the two sets by gaining the information directly from the configurator. If this is not possible because the configurator does not return that information, the sets S_A and D_A are calculated with the help of S and S'. The sets S'_A and D'_A are always empty before d_s is executed. The set S'_A is calculated by removing the old selection set from the new selection set ($S'_A = S' \setminus S$). The set D'_A is calculated by removing the new selection set from the old selection set ($D'_A = S \setminus S'$). Based on the sets of σ_i and results of *checkValidity* we define constraint which FeatureDiagramSynthesizer adds as constraints to M. The first constraint is an implication and is derived as follows. The new selected feature f combined with all selected features from the old state (σ_{i-1}) without the elements of all automatically deselected features from the actual state (σ_i) build the left side of the implication:

$$f \wedge (\bigwedge_{i \in S \setminus D'_A} i) \tag{3.3}$$

We remove the automatic deselected features because we assume that the actual selection caused the constraint. The conjunction Equation 3.3 implies a conjunction of all automatic selected features of the new state without the automatic selected features of the previous state:

$$\bigwedge_{j \in S'_A \setminus S_A} j \tag{3.4}$$

The automatic selected features of the previous state should not be considered in the constraint because they are considered before and they would unnecessarily increase the formula. The second constraint is also an implication and uses the conjunction of Equation 3.3. The right part of the implication is a conjunction of all automatically deselected features of the actual set which results in:

$$\bigwedge_{j \in D'_A} \neg j \tag{3.5}$$

The step adding constraint provides the following constraints:

$$\begin{aligned} \bigwedge_{i \in S'} i &\implies \bigvee_{j \in N} j \\ f \wedge (\bigwedge_{i \in S \setminus D'_A} i) &\implies \bigwedge_{j \in S'_A \setminus S_A} j \\ f \wedge (\bigwedge_{i \in S \setminus D'_A} i) &\implies \bigwedge_{j \in D'_A} \neg j \end{aligned} \tag{3.6}$$

We call the presented constraints *complex constraints*. These constraints could not be strict enough and allow too many configurations in the feature model. For that reason we introduce a second method to create constraints that are more restrictive than complex constraints.

https://doi.org/10.24355/dbbs.084-201812111501-0

We only use requires and excludes relations between two features.

- Requires: $A \implies B$
- Excludes: $A \implies \neg B$

Furthermore, we assume that the last selected feature always causes the actual selection state of the a configurator. That includes the elements of the three tuple (S, S_A, S_D) of σ_i. We call these constraints *simple constraints*. The simple constraints also change the created constraints from FeatureDiagramSynthesizer (cf. Equation 3.6). The left side of all implications is always the last selected feature f_L. The right side of the implications does not change and we can use it to create the simple constraints. The resulting constraints are:

$$\begin{aligned} f_L &\implies \bigvee_{j \in N} j \\ f_L &\implies \bigwedge_{j \in S'_A \setminus S_A} j \\ f_L &\implies \bigwedge_{j \in D'_A} \neg j \end{aligned} \tag{3.7}$$

More information about simple constraints and whether it is possible to convert complex constraints into simple constraints is given elsewhere [KTM+17a]. We create only the constraints from Equation 3.6 and Equation 3.7 if both sides of the implication contains at least one element. The implications are not meaningful for the feature model if one side is empty. If we would allow implication, such as $A \implies$ which is the equivalent to $\neg A$, we would restrict the model too much. In that example, the feature A never could be selected in any valid configuration.

Different things can happen after selecting a feature. One case is that this feature is selected and nothing else happens. This leads to no constraints and the algorithm continues with the next step of the selection strategy. However, it is possible that a dialog opens which asks the user to make additional selections with other features. We basically support dialogs as we describe in *Assumption 4*. If the dialog contains features from F, the features $g \in G$ inside the dialog can be considered in a constraint as $S \implies G$. The features of G can be in an alternative-group, or-group, or and-group. We use the or group if the dialog does not explicitly inform about the relation. The resulting complex constraint is shown in Equation 3.8 and the simple constraint is shown in Equation 3.9.

$$f \wedge \bigwedge_{i \in S'} i \implies \bigvee_{j \in G} j \tag{3.8}$$

$$f_L \implies \bigvee_{j \in G} j \tag{3.9}$$

Afterwards the dialog will be reverted. The algorithm adds the found constraints as a cross-tree constraint to M.

https://doi.org/10.24355/dbbs.084-201812111501-0

The selection strategies could revert selections. It is not always possible to revert a click because a dialog may open and additional decisions have to be made before a reversion is possible. This must be considered by the selection strategy.
The algorithm considers found constraints. If the algorithm finds a constraint which is part of M, the constraint will not be added again to the model M. The algorithm just continues with the next step which the selection strategy proposes. The algorithm will use at least two selection strategies. Thus, more constraints can be found by our algorithm.

3.3.3 Strategy for Selecting Features.

There exist different strategies how features can be selected by an algorithm. We describe several strategies in this section.
Some strategies correspond to sampling methods [HNA+17]. Different strategies are possible for our use case. The algorithm will work with different strategies. It depends on the implementation which strategy will be used. It is meaningful to run the algorithm with different strategies to find more constraints. We introduce five established strategies which the algorithm possibly can use. They are discussed in [HNA+17, MKR+16, VAHT+18].

- **One-disabled**. This strategy selects all possible features except one feature. Inside alternative groups one feature is randomly selected. However, all combinations inside alternative groups fulfill the one-disabled strategy, but only one valid configuration is used. Therefore, not all features from alternative groups are used to find constraints and constraints could be missed. This strategy finds excludes of features.

- **All-one-disabled**. This strategy extends the one-disabled strategy. The alternative features are selected and every valid combination is used. It is very expensive to use that strategy but we would find many constraints such as implications and excludes.

- **One-enabled**. This strategy selects only one additional feature to the selected features from S_A for a configuration. All other options are disabled for that configuration. With the one-enabled strategy we find implications but no excludes.

- **Most-enabled-disabled**. This strategy uses two samples. In one sample it selects as many features as possible and in the second sample it deselects as many features as possible. Implications and excludes can be found by the most-enabled-disabled strategy. However, constraints which occur with numbers of selections between the most selected and deselected cannot be found by this strategy.

- **Random sampling**.This strategy selects a random number of features for a valid configuration. Not every random sample of features corresponds to a valid configuration. Therefore, it is necessary to built more samples. With that strategy it is not possible to find all constraints.

https://doi.org/10.24355/dbbs.084-201812111501-0

3.3.4 Algorithm for FeatureDiagramSynthesizer

In this section, we describe the algorithm of FeatureDiagramSynthesizer as plan of procedures. The needed variables with regard to contents are similar to the defined sets and states which we defined earlier in that section (cf. Section 3.2).

Step Creating the Feature Diagram

1. Readout configurator c and find all features and insert them into F.
2. Find hierarchies and insert them into H.
3. Find alternative features and insert the groups into A.
4. Create the feature diagram from F, A, and H.

Step Adding Constraints

1. Define at least two selection strategies *strategies*.
2. For all elements from *strategies* do: Choose a selection strategy from *strategies*.
   (a) Follow rules of selection strategy until the strategy ends.
      i. Select elements f from F as selection strategy proposes.
      ii. Check if dialog opens.
         A. TRUE, resolve dialog issue and add constraint.
         B. FALSE, continue.
      iii. Set f_L as last selected element from F.
      iv. *CheckValidity*.
         A. TRUE, continue.
         B. FALSE, break, create new configuration by selection strategy. Add constraints depending on constraint strategy:
         C. complex constraint: $\bigwedge_{i \in S'} i \implies \bigvee_{j \in N} j$
         D. simple constraint: $f_L \implies \bigvee_{j \in N} j$
      v. Switch case:
         A. Nothing else happens. Continue.
         B. Automatic selections were made. Add constraint depending on constraint strategy:
         C. Complex constraints: $\bigwedge_{i \in S} i \wedge f \implies \bigwedge_{j \in S'_A} j$ to M.
         D. Simple constraint: $f_L \implies \bigwedge_{j \in S'_A \setminus S_A} j$
         E. Automatic deselections were made. Add constraint depending on constraint strategy:
         F. Complex constraint: $\bigwedge_{i \in S} i \implies \bigwedge_{j \in D'_A} j$ to M.

https://doi.org/10.24355/dbbs.084-201812111501-0

G. simple constraint: $f_L \implies \bigwedge_{j \in D'_A} \neg j$

(b) Remove redundant constraints.

The algorithm works basically for all configurators which fulfill our assumptions. However, there are some restrictions to the algorithm. We discuss them in the following Section 3.4.

Simplify the Feature Model

After the algorithm created the model M, a simplification with respect to the model is started to create a simpler and more compact feature model. The constraints could possibly be simplified with the help of propositional logic and simplification. Different algorithms for that use case are discussed in [vRGA+15]. The algorithm eliminates redundant constraints. Redundant constraints will be eliminated one after the other. Removing more of them to the same time could possibly violate the validity of the model.

3.4 Discussion

The developed algorithm builds a basis to reverse engineer a feature model from product configurators. Because of many different configurators, we made assumptions which restrict the possible choice of a configurator. The algorithm must be extended with more functions if it should work for more product configurators. Dialogs or stage-based configurators need additional functions in the algorithm. The basis of the algorithm can be used to realize new algorithms which are able to reverse engineer different product configurators.

The configurator model can change at any time but we want to have a valid and good feature model from a configurator. Therefore, it is necessary to run the algorithm more than once. The algorithm has to run at least two to four times. If the first and second run result in the same model, we can end the algorithm. However, if we would run FeatureSynthesizer only two times, it may also happen that the two reverse engineered feature models differ. That can happen because the underlying model of the configurator changes during the execution of FeatureSynthesizer. Therefore, the algorithm must be repeated. Hence, three runs are not enough. If the change of the model was during the second run, we would receive a model from the first run without changes by the configurator, a model from the second run where changes to the configurator are done during the execution, and a model after the changes to the configurator. That leads to three different models. A fourth run should result in the same model as the third. If the fourth run also creates a different model than the other three runs, the algorithm stops. Then it is not possible at the moment to reverse engineer a feature model from the configurator. The reason could be maintenance during the execution of FeatureSynthesizer which results in different models. It could be tried to run FeatureSynthesizer again after some hours of waiting. If the resulting feature models differ again in all four

runs, the implementation of the algorithm should be checked concerning programming faults. However, it could be impossible to re-engineer the product configurator.

We know that it is not possible for our algorithm to find all constraints. If we want to find all constraints, we need all possible configurations and that is infeasible. However, the algorithm can use different selection strategies to cover as many configurations as possible to find constraints. We reduce missing constraints by using different selection strategies. The structure of our algorithm allows to plug in these different strategies. It is recommended to use more strategies.

For some configurators it can happen that many constraints are found. Our algorithm tries to minimize the constraints and reach the minimum amount of necessary constraints. It is possible to minimize the constraints in a better way. Therefore, it is conceivable to develop improved selection strategies. One possibility is to select ten features and add a new which makes the configuration invalid. After that deselect two of the ten selections and try to find out whether these two features cause the invalid configuration.

Todays product configurators are able to change their model and constraints on the fly without informing the user. Our algorithm cannot recognize these changes. The algorithm should be executed at least twice with the same selection strategies. The created models are compared and if all are equal, the algorithm ends successfully.

Our algorithm basically is able to find minimum and maximum constraints. *Assumption 7* excludes this fact. The reason is the exponentially rising amount of required constraints to represent minimum or maximum selection criteria with a growing amount of features. Therefore this will only work for small configurators with only a few features.

3.5 Summary

In this chapter, we developed an algorithm to reverse engineer a product configurator. Therefore, we made assumptions which the product configurator or the algorithm must fulfill. Furthermore, we formalize the configuration process with different sets and states. We defined two three tuples which our algorithm uses. The developed algorithm has two main steps. Firstly, the feature diagram is created in a static way. Secondly, the constraints are added to the diagram. Afterwards the model is simplified. Moreover, we present selection strategies and possible simplifications for the resulting model. At the end of this chapter, we discussed our algorithm.
In the next chapter, we will implement the algorithm with a prototype for a concrete product configurator.

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

4. Tool Support

We presented conceptually how a feature model can be reverse engineered from a web configurator. A test in the praxis with a real product configurator should be done to prove the concept against an existing product configurator. One possibility is to manually execute the algorithm FeatureDiagramSynthesizer. However, the manual approach is too expensive and would take much time. For that reason, we need tool support to automate FeatureDiagramSynthesizer. This chapter gives an overview how a prototype is implemented based on the developed concept in Chapter 3. We also show which supporting tools we use to implement the algorithm FeatureDiagramSynthesizer. The tools are *Selenium*, *Geckodriver*, *FeatureIDE*, and *SAT4J*. Furthermore, we show a generic implementation for FeatureDiagramSynthesizer with an instance for the Ricoh configurator. We implemented the program generic because it should be easy to extend the program with new configurators. Other adapters for configurators which match the assumption of our concept can be implemented and use the main structure of FeatureDiagramSynthesizer.
The tool is developed in the programming language Java with the help of the integrated development environment *Eclipse*[1]. We use the version 4.7 Eclipse Oxygen.

4.1 Used Tools

For the implementation of our prototype we need different functionalities. We need functions to interact with a website, creating feature models, or simplifying a feature model. We do not have to develop all functions by ourselves because there exist different open source tools. We use these tools which are already implemented and work well as support for our implementation. The tools undertake subtasks such as creating feature models.
With the help of Selenium and geckodriver, we create an interface between our program and an online configurator which means giving access for our program to the website. FeatureIDE supports us by creating the feature models. Sat4j helps to check whether a configuration is valid or not.

[1] cf. https://www.eclipse.org/

https://doi.org/10.24355/dbbs.084-201812111501-0

4.1.1 Selenium

Selenium is an open-source framework to automate web browsers [Sel]. It is most used for automating tests of web applications. Selenium is not limited to a specific platform or programming language. Supported browsers are *Firefox*, *Internet Explorer*, *Chrome*, and more. Supported programming languages are for example *Java*, *C#*, and *Python*. Established operating systems, such as Windows, Linux, and Apple OS X, are also supported by Selenium. The Selenium framework supports different use cases. There exist two main parts which can be used, *Selenium WebDriver* and *Selenium IDE*. The Selenium WebDriver is the successor of the deprecated Selenium Remote Control. Selenium WebDriver allows navigating trough a browser with the help of a browser specific driver. The WebDriver operates as a user would do [Sel]. The WebDriver is able to send commands (e.g., click on button) to the browser and receives the results from the web page. The Selenium WebDriver provides a programming interface. It also provides support for dynamic web pages and tests for them. Selenium IDE is a tool for creating test cases for websites. It is an add-on for the browser Firefox and Chrome. The test cases are created by recording user interactions inside a browser and defining additional parameters. For additional information about Selenium, such as implementation or integration, can be found on the Selenium website.[2] An overview on Selenium and its functions or maintainability related to other test frameworks is given elsewhere [LCRS13, JK15, NBN14].

We use the Selenium WebDriver for the implementation. The WebDriver provides the functionalities we need to interact with a website, e.g., find selected checkboxes or get text information from a dialog. The Selenium WebDriver provides the interface we will use to readout information from a website, such as selection status of a checkbox. The Selenium IDE is used to create tests. In contrast to the Selenium WebDriver, we cannot readout the needed information from a webpage because the interactions are limited. We could only create test cases. However, the test cases do not return all needed information. We need to create a feature with name from text information but they are not accessible within test cases. Therefore, we use the Selenium WebDriver.

4.1.2 Geckodriver

The geckodriver operates as a proxy for W3C WebDriver-compatible clients to interact with Gecko-based browsers, such as Firefox. Therefore, the driver provides a HTTP API to communicate with the browser [Gec]. The geckodriver is often used in combination with Selenium (cf. Section 4.1.1). It gives additional support for testing and interacting with web pages [Ani17, Gec].

4.1.3 FeatureIDE

FeatureIDE is an Eclipse-based IDE and provides support for feature-oriented software development which is part of software product lines [MTS+17]. It gives support for creating feature diagrams and configurations. We also use FeatureIDE to create

[2] https://www.seleniumhq.org/

https://doi.org/10.24355/dbbs.084-201812111501-0

and show the extracted feature model as a feature diagram.
We use the FeatureIDE library [KPK+17] in our program. It is not necessary to use the complete Eclipse version with FeatureIDE. The library gives us functionalities for creating and working with feature models, such as adding constraints to a feature model. Furthermore, we can export our program with the included library to work on other computers which do not have FeatureIDE installed. The main classes we use are from the packages:

- *de.ovgu.featureide.fm.core.base*
- *de.ovgu.featureide.fm.core.io.manager*
- *de.ovgu.featureide.fm.core.configuration*
- *org.prop4j*

They provide functionalities to build a structure for a feature model which includes adding features and constraints to that model. For that purpose, we use the class *FeatureUtils* which has the functionalities to add features, constraints, or changing the root of the corresponding feature model. Furthermore, we can create and use configurations from that feature model. We also use the functions to recreate an *IFeatureModel* from a feature model saved as XML file. The package org.prop4j gives us the functions for finding redundant features with the help of SAT solvers.

4.1.4 Sat4j

The open source Java library Sat4j consists of different SAT solvers [LBP10]. It is used to solve satisfiability problems. Furthermore, it is possible to solve decision and optimization problems by using pseudo-boolean solving. Sat4j is widely-used in different areas of application, such as software engineering or inside of the development platform Eclipse [LBP10]. With the help of the functionalities of Sat4j we simplify our feature model. That results in a more qualitative model.

4.1.5 Log4j

Log4j is an open-source framework for Java distributed under the Apache Software License. It is used to implement logging mechanics into a software program [LOG]. We use Log4j to log the important steps of the implementation and for debugging if necessary.

4.2 Implementation of a Prototype for FeatureDiagramSynthesizer

The program is developed in the language Java. We choose this programming language because there are tools, such as Selenium, which support our implementation. We implement the program against the browser Firefox. Firefox is one of the most used browsers. A migration to other browsers, such as Chrome or Internet Explorer, is possible. They would need other kinds of drivers, such as geckodriver, and maybe

https://doi.org/10.24355/dbbs.084-201812111501-0

small changes in the program but the main part of the program would be the same. For our experiments, it is not necessary to implement the program for all browsers. We describe in this section the general structure of the implementation of FeatureDiagramSynthesizer. Moreover, we present a prototype for the Ricoh configurator.[3]

4.2.1 Structure of the Prototype

In this section, we describe the main structure of the implementation of FeatureDiagramSynthesizer. We explain the important classes, relations between them, important methods and variables of the implementation of FeatureDiagramSynthesizer. We concentrate on the important methods and variables to understand the implementation and exclude get and set methods. The program of FeatureDiagramSynthesizer consists of two main packages, the adapter and the backend. There are also a few classes which give additional functionalities to the main packages, such as a file writer.

Package de.fdsynthesizer.backend

The package backend contains the classes of FeatureDiagramSynthesizer which are used to control the execution of the program and needed by other classes to work, such as *ConfiguratorFeature*. Objects from the class *ConfiguratorFeature* represent features of a configurator. The class is used to collect the needed information (cf. fields in Figure 4.1) from the configurator to create the feature model. Moreover, the objects of *ConfiguratorFeature* are used to find the corresponding elements on the website. Therefore, the field *id* or *name* can be used. It depends on the implementation of the configurator which field is used because not every configurator uses id's in the HTML context. The field *name* is mandatory and unique because it is also the name of a feature in the feature model. The other fields give additional information about the relations (abstract, mandatory, or groups) and positions (parent or root) which are used during the feature model creation.

Another important class is *Algorithm*. This class is responsible for the control of the execution of the program for FeatureDiagramSynthesizer. Figure 4.2 gives an overview of the important methods and fields. The class uses variables from the type *IFeatureModel* and *IConfiguratorAdapter*. The constructor of *Algorithm* needs the feature model on which features and constraints have to be added and a concrete implementation of an adapter. The class *Algorithm* has four important methods for the control of FeatureDiagramSynthesizer:

1. *runExtractFM(String)*
2. *runAddingConstraints()*
3. *simplifyModel()*
4. *compareXmlFiles(String, String)*

[3] https://ricohconfigurator.com/

https://doi.org/10.24355/dbbs.084-201812111501-0

ConfiguratorFeature

- abstractFeat: boolean
- alternative: boolean
- and: boolean
- attribute: Map<String,String>
- hasChild: boolean
- id: String
- isRoot: boolean
- mandatory: boolean
- name: String
- or: boolean
- parent: String

Figure 4.1: Attributes of ConfiguratorFeature

The first three methods are executed by the method *runFeatureDiagramSynthesizer(String)* which is the only method which needs to be executed by the main method to readout a feature model from a product configurator. The method *runExtractFM(String)* receives a link of a configurator as string input and needs to be executed first. This method corresponds to the static part of FeatureDiagramSynthesizer. The method reads out the features from a specific website with the adapter method *readOutFeatures(String)*. Finally, the method creates an XML file from the created feature model with support of our class *FeatureModelCreator* (cf. Listing 4.1).

Algorithm

- adapter: IAdapterConfigurator
- featureModel: IFeatureModel

+ compareXMLFiles(String, String): void
+ runAddingConstraints(): void
+ runExtractFM(String): void
+ runFeatureDiagramSynthesizer(String): void
+ simplifyModel(): void

Figure 4.2: Simplified overview Class Algorithm

```
public void runExtractFM(String link) {
  adapter.readOutFeatures(link);
  FeatureModelCreator fmc =
   new FeatureModelCreator(featureModel);
  fmc.featureMapToModel(adapter.getFeatures());
  xmlWriteFile();
}
```

Listing 4.1: Source code for *runExtractFM*

https://doi.org/10.24355/dbbs.084-201812111501-0

The method *runAddingConstraints()* is always executed after *runExtractFM(String)* because it needs a feature model. Theoretically, it is possible to execute the method for its own with a given feature model. However, a current model should be used because it is possible that the model of the product configurator changed since the last extraction of features, e.g., containing other features. The method *runAdding-Constraints* corresponds to the dynamic part of FeatureDiagramSynthesizer which includes the function *determineState* we defined in our concept (cf. Section 3.3.2). There are two selection strategies implemented in *runAddingConstraints()*, the one enabled strategy and a strategy which finds the biggest possible selection for a given configuration. We use the one enabled strategy because we assume that many constraints can be found by trying to select a feature. If the feature requires other features, we can use the information as a constraint. We abbreviate the second strategy as biggest configuration. The biggest configuration strategy builds a counterpart to the one enabled strategy by selecting as much as possible. The procedure for the biggest configuration strategy is selecting the given configuration and search for the next selectable feature until no selection is possible anymore.
The execution of both strategies can be dynamically controlled. It also can be controlled if there should be created only simple constraints (only $A \implies B$ or $A \implies \neg B$) or complex constraints. The method *runAddingConstraints* creates configurations which are committed to the specific adapter methods *findConstraints(Configuration, IFeatureModel)* or *findConstraintsBigConfiguration(Configuration, IFeatureModel)*. The method expects to get constraints from the adapter. Furthermore, the method *runAddingConstraints()* adds the constraints to the feature model and updates the XML file version of the feature model. During the creation of configurations the method *findConstraintsForFeature(IFeature)* is called to consider known constraints for the creation of configurations. It is possible to add more selection strategies to *runAddingConstraints()* without big effort. The implementation is open for new selection strategies.

The third method *simplifyModel()* tries to simplify the propositional representation of the feature model. This method removes redundant and tautology constraints from the feature model (cf. Listing 4.2). The method needs to be executed directly after the method *runAddingConstraints* because the feature model is saved inside the instance of the class *Algorithm*.

```
method simplifyModel() {
 featureModel = getFeatureModel();
  FOR(constraint in featureModel)
        IF(constraint is tautology)
           remove constraint;
    IF(constraint is redundant)
      remove constraint;

}
```

Listing 4.2: Pseudo code for *simplifyModel()*

https://doi.org/10.24355/dbbs.084-201812111501-0

The fourth method *compareXmlFiles(String, String)* compares two XML representations of a feature model. The comparison is done semantically. We use a simple strategy and only use the conjunctive normal form of a feature model to compare two models. Other methods for semantic comparisons are very expensive operations. More information about comparing and editing feature models is given elsewhere [TBK09]. The method *compareXmlFiles(String, String)* returns true or false. A syntactical comparison by comparing the string representation of two feature models is not implemented because it is error-prone, e.g., two equal constraints are at different position of the XML representation file. However, the feature models could be the same. Therefore, we use a semantic comparison with the conjunctive normal form of the feature models.

The method findConstraintsForFeature(IFeature) searches for all constraints in which a given feature is included and returns map of feature name and selection status. A map is build as *HashMap<String, Selection>*. The selection status for a feature can be selected or unselected. This information can be used by other methods of the class *Algorithm*, such as creating a configuration which considers the constraints.

There are two methods which are used to create and execute test configurations for the evaluation of a re-engineered feature model.

1. *checkConfigurationFM(String)*
2. *checkConfigurationPC(String)*

The methods *checkConfigurationFM(String)* and *checkConfigurationPC(String)* are used to create test configurations for a given feature model. The configurations are created randomly. In the method *checkConfigurationFM(String)*, the configurations are generated from the reverse engineered feature model and tested against the product configurator and in the method *checkConfigurationPC(String)* the other way around. A proportion from valid to invalid configurations in relation to the number of configurations is calculated. The result of the proportion, valid and invalid configurations, is written into a result file.

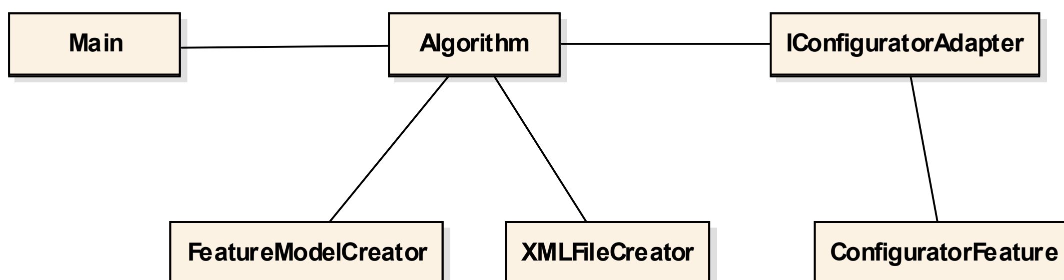

Figure 4.3: Relation between classes of the prototype

https://doi.org/10.24355/dbbs.084-201812111501-0

The relations between the main classes are shown in Figure 4.3 and the process of the implementation of FeatureDiagramSynthesizer is shown in Figure 4.4. The class *Algorithm* can be seen as controller. It is the starting point for the program. It uses classes, such as *FeatureModelCreator*. Furthermore, it starts the methods of a concrete adapter to extract information from a product configurator.

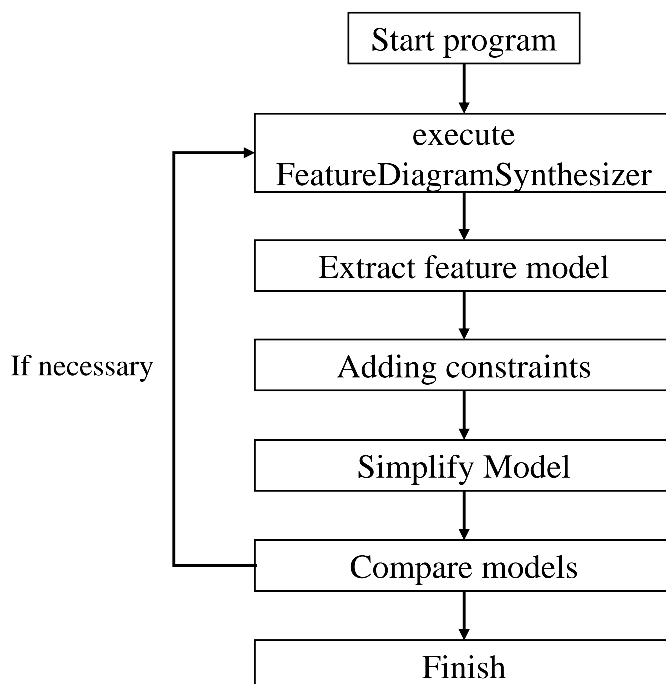

Figure 4.4: Process of Implementation of FeatureDiagramSynthesizer

Package de.fdsynthesizer.adapter

This package represents the adapter which has the main function to interact with the website. Therefore, we implement an abstract class *AConfiguratorAdapter* which implements an interface *IConfiguratorAdapter* to predefine needed methods and variables. A concrete adapter for a configurator extends the abstract class as shown in Figure 4.5. For every new configurator that should be re-engineered, it is necessary that an adapter class is created which extends the abstract class AConfiguratorAdapter. In AConfiguratorAdapter we define variables which are always needed for every concrete subclass. We decided to use that generic structure with an abstract class and interface because it raises the extensibility of the program. It is very easy to implement a concrete adapter. All needed methods and variables are predefined, so that a programmer knows what methods he has to implement. Furthermore, we ensure that the static and dynamic steps (cf. Section 3.2) of FeatureDiagramSynthesizer are prepared and considered, such as initializing needed variables and make them accessible.

https://doi.org/10.24355/dbbs.084-201812111501-0

The variables of the abstract class *AConfiguratorAdapter* are the objects shown below:

- *WebDriver* webDriver
- *List<ConfiguratorFeature>* featureList
- *JavascriptExecutor* jsExecutor
- *IFeature* lastSelectedFeature
- *boolean* findSimpleConstraint
- *Set<ConfiguratorFeature>* selection, previousSelection, autoSelection, previousAutoSelection, autoDeselection, previousAutoDeselection

The prototype has a good cohesion. The different functionalities of the prototype are distributed in their own closed program modules The classes WebDriver and JavascriptExecutor are given by Selenium and are used to open and interact with a website, such as finding elements, executing JavaScript commands, or returning text information. The featureList contains all found features for the adapter. A last selected feature during the reverse engineering process is saved in the variable lastSelectedFeature. We differentiate two kinds of constraints which can be found by the algorithm, complex and simple constraints. The constraint generation is controlled with the boolean variable findSimpleConstraint. The six sets of ConfiguratorFeature objects correspond to the sets we described in Section 3.2.
The interface *IConfiguratorAdapter* predefines the needed methods which must be implemented for the algorithm to be executed as wanted. The method *readOutFeatures(String)* takes a specific link of a product configurator website, finds all features from that configurator, and stores them in *List<ConfiguratorFeature>* featureList. The method *findConstraints(Configuration, IFeatureModel)* needs the feature model (*IFeatureModel*) and a configuration (*Configuration*) for the model. It tries to find all constraints resulting from the given configuration if it is selected in the product configurator and returns a list of constraints (*List<Constraints>* constraints). The method *findConstraintsBigConfiguration(Configuration, IFeatureModel)* is similar to the method *findConstraints(Configuration, IFeatureModel)* with one difference. This method tries to find the biggest possible selection for the given configuration. That means additional selections to the given configuration are done directly on the product configurator. The constraints follow from our algorithm definition (cf. Section 3.3). The validity of the current configuration can be tested by the method *checkValidity()*. This method also follows from the algorithm definition (cf. Section 3.3), returns a boolean value, and is overloaded. There exists a second implementation possibility of *checkValidity(Configuration, IFeatureModel)* which returns a list of constraints (*List<Constraints>* constraints). The second implementation is used for configurators which are able to list the features that are needed to result in a valid configuration.
It is always possible to include additional methods to a concrete adapter based on own knowledge of the configurator. The adapter is the interface between website and

https://doi.org/10.24355/dbbs.084-201812111501-0

algorithm. Own knowledge can improve that interface, such as additional methods for finding hierarchies from the configurator.

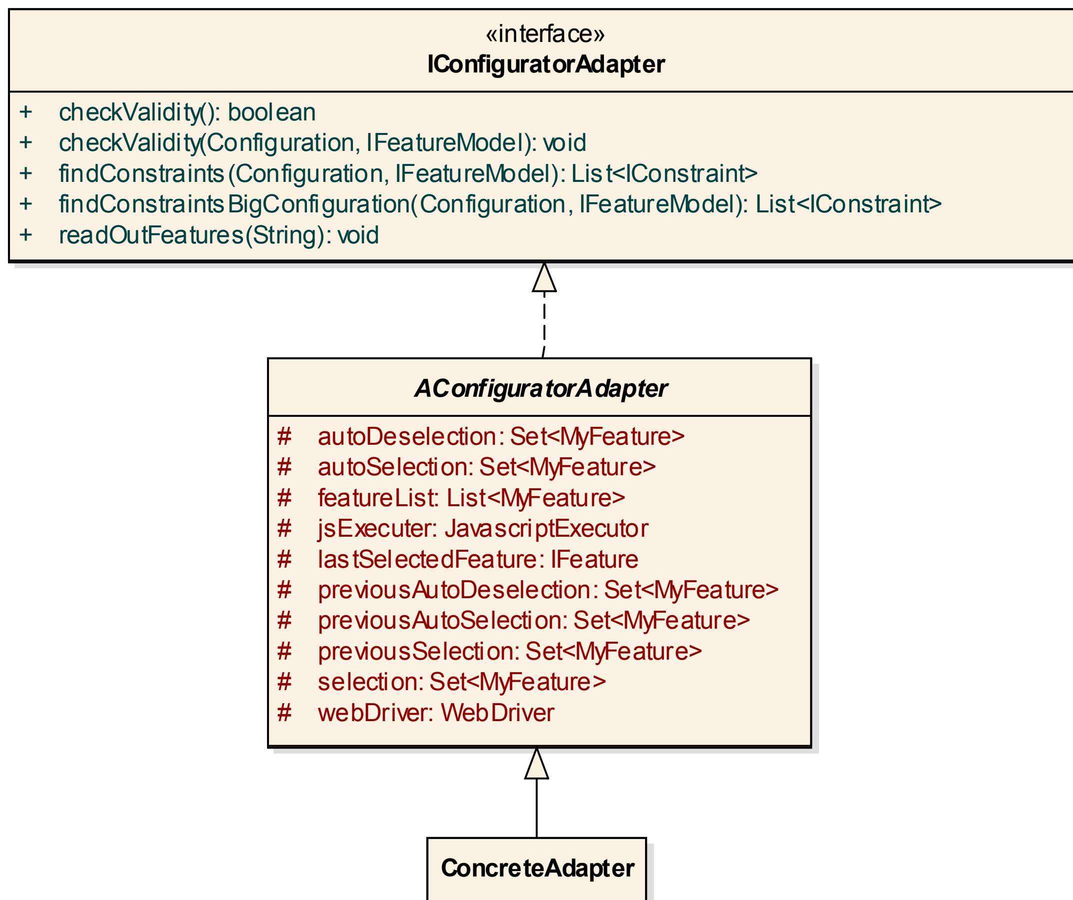

Figure 4.5: Simplified Overview Structure of Adapter

4.2.2 Prototype for Ricoh Configurator

In this section, we describe the implementation of a concrete adapter for the Ricoh configurator[4] with its special requirements and the methods we implemented additionally. With the Ricoh product configurator, you can configure different kinds of printers and plotters from the company Ricoh. We decided to use the Ricoh configurator because it matches all of our assumptions. We also checked different other configurators but they do not match all our criteria. Table 4.1 gives an overview of the different configurators(Ricoh, Volkswagen[5], One[6], Jori[7], MyMuesli[8]) in comparison to our assumptions. *Assumption 3 (one selection in a dropdown menu)*, *Assumption 5 (model change)*, and *Assumption 7 (minimum and maximum configuration)* are not rated. It is not meaningful to use the three assumptions as criteria

[4]cf. https://ricohconfigurator.com
[5]http://www.vw.com/builder/
[6]one.de
[7]https://www.jori.com/en/configurato
[8]https://uk.mymuesli.com/mixer

https://doi.org/10.24355/dbbs.084-201812111501-0

	Ricoh	Volkswagen	One	Jori	MyMuesli
A1: Recognizing all features at start	yes	no	partly[1]	partly[9]	yes
A2: Selection status	yes	yes	yes	yes	yes
A4: Simple dialog	yes	no	not rated[2]	no	not rated[2]
A6: Check validity	yes	yes	yes	yes	yes
A8: One step	yes	no	partly[1]	partly[1]	yes
A9: No manual deselected feature	yes	yes	yes	yes	yes
A10: Robots.txt allows bot access	yes	no	yes	yes	yes

[1]Partly means that it is not always guaranteed for all products to fulfill the assumption. [2]Not rated is used if the assumption does not effect the configurator.

Table 4.1: Overview about different configurators against assumptions.

for choosing a configurator. Assumption 3 does not consider one of the configurators, Assumption 5 cannot be rated before readout at least two feature models, and Assumption 7 cannot be answered before extracting a feature model from a configurator. The Volkswagen configurator is too complex for our concept. The configuration process has several steps and the robots.txt forbid the kind of bot we want to use. The other configurators we analyzed allow the access with a bot. However, One and Jori do not completely fulfill Assumption 1 and 8. The MyMuesli configurator fulfills nearly all assumption but assumption 4 could not be answered. We cannot say whether simple dialogs occur or not because during the analysis no dialog pops up. The Ricoh configurator satisfies all assumption. In comparison to MyMuesli we can say that simple dialogs are used inside the Ricoh configurator.

The Ricoh configurator only uses checkboxes for all features of a specific product which are not abstract. The checkboxes are grouped under different steps which could be seen as parents. The parents are not selectable and can be considered as abstract features. A checkbox can have the status selected, deselected, or disabled. We are also able to check the validity of a configured product of the Ricoh configurator. Therefore, we use the function (button) from the website "*Complete your configuration*". If the product is invalid, a dialog opens and gives information which features are missing. If the product is valid, a new website is opened and the product can be bought. We do not follow and go back to the start page. We implemented the concrete adapter class *ConfiguratorAdaterRicoh* with its special variables and methods to interact with the Ricoh configurator website. The methods predefined by the interface *IConfiguratorAdapter* are implemented with focus to the needs of the configurator. For all methods which interact with the website, we have to implement breaks before doing the next interaction. Moreover, the calculations are faster than the website. If we would not pause our interactions, the website could crash. We do not want to overload the servers with too many and fast queries in a short time.
In the following, we describe the content of the methods of the concrete adapter for the Ricoh configurator. The implementation corresponds to the sets and algorithm we defined in Section 3.2.

https://doi.org/10.24355/dbbs.084-201812111501-0

ConfiguratorAdapterRicoh: readOutFeatures(String)

The method readOutFeatures(String) does not interact with the website. It only reads out information from the HTML content. The method corresponds to the first main step of our algorithm FeatureDiagramSynthesizer. This method finds all features by searching HTML checkboxes and puts them into the set F. In the next step parent elements are connected to the features. Afterwards all parents are connected to a root element. The root element corresponds to the name of the product. The set H is filled with the information of the connection between the features to parents and the root object. We know that there is always a group of features where we have to select or choose a feature for every configuration in the Ricoh configurator. With that domain knowledge, we can fill the set A of alternative features. The features are always grouped under the parent "Step 1: Main Unit". Therefore, we set the features under that parent into an alternative group. If only one feature is under the parent "Step 1: Main Unit" we set the feature as mandatory feature. At the end of the readout process the feature diagram with the hierarchies of features is created.

ConfiguratorAdapterRicoh: findConstraintsFirstRound(Configuration, IFeatureModel)

The method *findConstraintsFirstRound(Configuration, IFeatureModel)* is a special method implemented for the Ricoh configurator and can be used optionally. If the method is used, it should be executed as first method for finding constraints (e.g., before *findConstraints(Configuration, IFeatureModel)*). The method receives a configuration which only contains mandatory features. We know about mandatory features by domain knowledge we included. For a valid configuration it is always necessary to select a feature from the group "Step 1: Main Unit". Without the selection of one feature from that group, the Ricoh configurator does not allow any selection. Therefore, we have the given configuration. The method receives the configuration and tries to select only one additional feature F. If there are dependencies to select that feature, a dialog opens. We readout the information from that dialog and save the information as constraint. The constraint is always an implication $F \implies neededfeatures$. At the end the method returns a list of constraints and the constraints are added to the feature model. The found constraints can be considered by following steps of FeatureDiagramSynthesizer.

ConfiguratorAdapterRicoh: findConstraints(Configuration, IFeatureModel)

This method interacts with the website and corresponds to the second main step of FeatureDiagramSynthesizer. The given configuration is selected by that method. If the selection opens a dialog where additional feature selections are needed, the constraint is added and the selection is made. Afterwards the sets of selected S, automatically selected S_A, and deselected D_A features are filled. In addition to the calculations D_A will be filled with all features which cannot be selected anymore. In the Ricoh configurator a checkbox is not selectable if it is disabled. Based on the three sets the constraints are created. At the end, the constraints are added to the feature model and a new configuration can be tested.

https://doi.org/10.24355/dbbs.084-201812111501-0

4.3 Discussion

In this chapter, we want to discuss our implementation. We describe extensibility and difficulties during the implementation.
The implementation of FeatureDiagramSynthesizer is flexible. It is possible to adapt the concept to different configurators which match the assumptions. Another programmer can use the core parts of the program. There are only a few interfaces which must be considered, such as call of methods of *Algorithm*. The concrete adapter is the only new implementation which is necessary for a new product configurator. Furthermore, we implemented different possibilities for the program execution. It is possible to control the number of selection strategies or which kind of constraints should be created.

The program has a good extensibility. We use Javadoc to describe our implemented methods to support the understandability and what a method needs and returns. The program can be executed easily if a concrete adapter is implemented because there is only one method which must be called to start FeatureDiagramSynthesizer. It is enough to call the method *runFeatureDiagramSynthesizer(String)* and transfer the link of the product configurator which should be reverse engineered. The interface for an adapter implementation supports a programmer because it is clear which methods have to be implemented. We have a modular implementation. There is no single 'god' method which is responsible for the complete algorithm. We have the core methods (cf. Section 4.2) and supporting methods. All these methods can be tested without big effort. We implemented a logging with Log4j to log the important events. It is documented what our program does and which errors occur. The log file can be analyzed after every run.

Our developed program can be extended well. It is possible to add more selection strategies without creating new classes. It is only necessary to add the selection strategy as method in the class *Algorithm*. A new adapter for a product configurator can be added by extending the abstract class *AConfiguratorAdapter*. The needed methods are predefined by the included interface *IConfiguratorAdapter*.

The program is robust against unexpected behavior. We implemented catch statements to react against exceptions which could occur during the execution. These exceptions could happen if interactions with the website have problems, such as executing a click twice instead of once. However, our program can handle these exceptions and continue running. This includes especially errors which could occur while interacting with the website. However, our program would terminate abnormally if the connection to the Internet is interrupted during the execution. A stable Internet connection is necessary. It is recommended to use a LAN connection because wireless connections are more error-prone.

The runtime of the program can be very different between different computers. The power of a CPU and the Internet connection have a high influence on the program. That is because we have computationally intensive methods such as *simplifyModel* which uses satisfiability queries. Moreover, there are queries against a website which needs more time with a bad Internet connection.

https://doi.org/10.24355/dbbs.084-201812111501-0

Our program needs some timeouts while interacting with the website. This is necessary because after an interaction, such as a click, the website needs time to react, especially if a dialog pops up. If we would continue with clicking for new selections, we might miss a dialog or end in a status where an interaction is not possible anymore. To avoid that, we implemented some very short timeouts (<400 milliseconds). This brings an additional advantage. We avoid to interact too fast with the website and do not overload the server of the website. With the timeouts we reduce the possibility that this happens.

4.4 Summary

We implemented the core parts for the algorithm FeatureDiagramSynthesizer and create a working prototype. This especially includes the class *Algorithm* to control the procedure of FeatureDiagramSynthesizer and the interface and abstract class for the adapter. We described the important methods and their functionality. The prototype is based on our developed concept (cf. Chapter 3) for re-engineering product configurators. The core parts can be adapted for every product configurator. Moreover, they can be extended with new functionalities such as new selection strategies. The program is implemented in a generic way to have a high extensibility. We implemented a concrete adapter for the Ricoh product configurator.

https://doi.org/10.24355/dbbs.084-201812111501-0

5. Evaluation

In this chapter, we evaluate our concept for the algorithm FeatureDiagramSynthesizer with the prototype for the Ricoh configurator. We want to answer the goal of the thesis whether it is possible to re-engineer a feature model from product configurators and with which exactness. In Section 5.1, we present the Ricoh configurator for printers and the feature models which are the result from FeatureDiagramSynthesizer executed against the Ricoh configurator. We describe the setup for our experiment and our research questions in Section 5.2. We present our results in Section 5.3 and discuss them in context with our research questions. All test results are from the tests of our prototype for the Ricoh configurator. At the end of that chapter, we give a short overview to threats to validity and what we do against them.

5.1 Subject Systems

We present the product configurator from Ricoh in this section. We also describe what usable information the Ricoh configurator provides. The second part of this section considers the feature models.

5.1.1 Ricoh Configurator

Ricoh is a brand with focus on printers for office management and industrial solutions and cameras[1]. We focus on the printer configurator from Ricoh which can be accessed at http://ricohconfigurator.com/. As described in Section 4.2.2 we chose the Ricoh configurator to evaluate our concept. The printer configurator matches all our assumptions such as presenting all features on one website. Every feature can be selected with help of a checkbox. A checkbox can have the status selected or deselected and enabled or disabled. Moreover, a checkbox can be automatically selected and deselected by the configurator during the configuration process. We are able to

[1] cf. https://www.ricoh.com/about/company/data/

https://doi.org/10.24355/dbbs.084-201812111501-0

readout that status of a checkbox at every time during the configuration process and whether the actual configuration is valid or not. We can use the information about the checkbox status to fill the sets of automatically selected (S_A) and deselected features(D_A) as well as all selected features (S). It is possible to check the validity of the model by clicking on a button called "Complete your Configuration". If the actual configuration is invalid, a dialog opens and shows what features are missing. For a valid configuration, a new website opens and shows the configuration. We use that behavior in FeatureDiagramSynthesizer and in the test case generation.
During the analysis of the printer configurator we found out that all features which are under the group "Step 1: Main Unit" are always necessary to receive a valid configuration. If there is more than one feature in that group, the features of that group are alternative. That domain knowledge is integrated into the prototype and test configuration generation.
We decided to evaluate five printers. Therefore, it is necessary to choose five different printers from the Ricoh configurator. The printers are randomly chosen because the prototype is developed to work with printers from the Ricoh configurator. The five printers are:

- MP_C501SP: 53 features (8 abstract)
- MP_C4504ex_C6004ex: 93 features (9 abstract)
- SP_C360SFNw_C361SFNw: 51 features (5 abstract)
- MP_305SPF: 42 features (7 abstract)
- SP_8400DN: 43 features (7 abstract)

They have a different number of features so that we can compare the results for different feature spaces.

5.1.2 Created Feature Models from the Printers

All models have a root which corresponds to the special name of the printer. The first hierarchy level contains between five and nine abstract features. The second hierarchy level is also the deepest level for the printer feature models. All features in that level are concrete features. They can be inside an *or* or *alternative* group. Figure 5.1 shows an extract from a reverse engineered printer from the Ricoh configurator as feature diagram.

5.2 Experiment Setup

We describe our experiment setup in this section. First, we present our three research questions. Moreover, we describe our procedure to test which includes the creation of random configurations and how they are tested. The last subsection presents the execution environment for the prototype.

The random configurations correspond to the configurations which are created from the feature model or the product configurator. In both cases the configurations

https://doi.org/10.24355/dbbs.084-201812111501-0

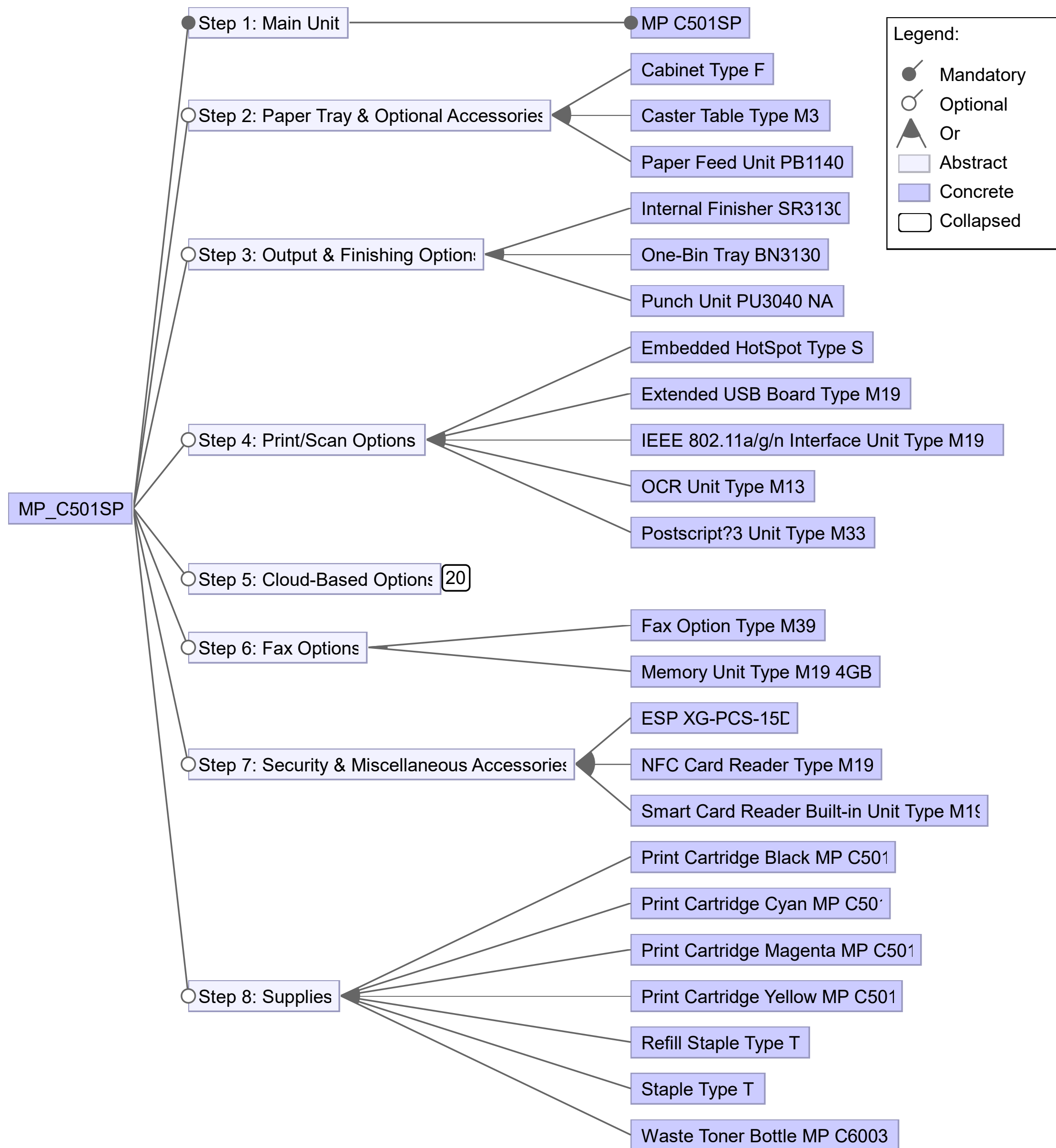

Figure 5.1: Extract from feature diagram for Ricoh product MP_C501SP

https://doi.org/10.24355/dbbs.084-201812111501-0

are randomly generated. We define two parameters which can be derived from the random configurations. The two values are false positives and false negatives. False positives means the number of configurations from the product configurator which are valid in the product configurator but invalid in the feature model. False negatives means the number of configurations from the feature model which are valid in the feature model but invalid in the product configurator.
We define the number of interactions with the configurators as another calculable parameter. We count every click the program does. This includes selecting and deselecting checkboxes, closing dialogs or clicking on buttons to check the validity.

5.2.1 Research Questions

We define three research questions (RQ) which we evaluate. These questions help to evaluate our concept with the support of the prototype for FeatureDiagramSynthesizer.

RQ1: How exact is the re-engineered feature model?

We know that it is impossible to recreate arbitrary feature models from a product configurator. Feature models and product configurators describe a set of valid variants (configurations). We have to use all valid variants from the product configurator to recreate an exact feature model. A feature model with 50 optional features would have 2^{50} possible configurations (cf. Section 2.2.2). That are too many configurations to test. Therefore, we decided to use a subset of 1000 random configurations out of all possible configurations for the evaluation.
We need a calculable value to evaluate the feature model. We decided to use false positives and false negatives for all strategies we proposed. The value is comparable to other feature models because we always use 1000 random configurations for the tests.
Our concept allows different selection strategies. The selection strategies are different approaches to create a feature model from product configurators. We implemented two different strategies, one enabled and find the biggest selection for a given configuration (abbreviated as biggest configuration). Which strategies gives the best result at its own? We also test whether the combination of both strategies results in a more precise feature model with higher accuracy. It is necessary to answer these questions by comparing different feature models. The feature models should be created only by one strategy or in combination.
Moreover, we have to differentiate how the constraints are build. The algorithm FeatureDiagramSynthesizer is able to create two kinds of constraints which are simple and complex constraints. The different constraints influence the test results. We compare the results concerning how good they fulfill the tests. We also evaluate what influence the different constraints have concerning the different strategies. Smaller values for false positives and false negatives means an exacter feature model. We calculate percentage values for all values. We define different exactness levels which are, exact, very good, good, bad, and very bad. The levels correlate with the false positive and false negative values of Table 5.1.

https://doi.org/10.24355/dbbs.084-201812111501-0

	False Positives	False Negatives
exact	$x \leq 1\%$	$x \leq 1\%$
very good	$1\% < x \leq 10\%$	$1\% < x \leq 10\%$
good	$10\% < x \leq 30\%$	$10\% < x \leq 30\%$
bad	$30\% < x \leq 60\%$	$30\% < x \leq 60\%$
very bad	$x > 60\%$	$x > 60\%$

Table 5.1: Exactness Levels

If we have less or equal than 1% of false positives and false negatives we assume that the re-engineered feature model is an exact model for the product configurator. We allow 1% variation (ten invalid random configurations) because it may happen that an issue occur while interacting with the website during the test execution, such an exception in one test. A false positive and false negative value between 1% and 10% corresponds to a very good feature model. A feature model with less or equal than 30% but more than 10% false positives and false negatives is a good model. The model would represent the product configurator in a good way but improvements are necessary. For more than 30%, the model is bad or with about 60% false positives and false negatives it is very bad. The feature model do not present the product configurator in a good way. There are many valid configurations missing.

RQ2: Which effort do we have to readout a feature model?

We can measure different parameters to determine the cost for the reverse engineering of a feature model from a product configurator. We decided to count the interactions with the website to measure the effort. We can set the clicks into relation with the number of features.
Measuring the costs with the runtime of the program is not meaningful. The performance depends on the performance of the computer where the program is executed. Moreover, the Internet connection influences the runtime. There are too many parameters which affect the runtime which disqualifies runtime as a useful cost measurement for FeatureDiagramSynthesizer.

RQ3: Which anomalies are in the feature model?

We can calculate how many constraints FeatureDiagramSynthesizer creates from a configurator. The found constraints can be redundant which includes tautologies. These constraints are not necessary for the feature model and could be removed from the model. Therefore, we analyze our model concerning these constraints. We can count how many redundant constraints exist and can set them into relation to all constraints. After removing the redundant constraints, we call the constraints *real constraints*.
It is possible that a feature model contains dead features. During the re-engineering process it might happen that we find dead features which result from our found constraints. We have to evaluate if this is a real dead feature or if the dead feature occurs because of our process to find constraints. We set the number of dead features in relation to the number of features of the printer model.

https://doi.org/10.24355/dbbs.084-201812111501-0

5.2.2 Procedure for Evaluating FeatureDiagramSynthesizer

It is necessary to create random configurations to answer the research questions. Moreover, we need different parameters such as number of clicks or valid configurations for the evaluation. To answer **RQ1**, we need the random configurations and we must check whether they are valid or not to match them into false positives and false negatives. This is possible after FeatureDiagramSynthesizer creates the feature model. The required data for **RQ2** and **RQ3** are collected during and after the execution of FeatureDiagramSynthesizer.
We test our concept with the prototype for the Ricoh configurator. Therefore, we create the random configurations from the product configurator and from the feature model. The first random configurations are created from a product configurator and tested against the created feature model. We count the false positives. The second tests use the other way around and create random configurations from the feature model and test them against the product configurator to count false negatives. In both cases, it is tested whether the configuration is valid or not. The false positives and false negatives give an overview how exact the re-engineered feature model is and help to answer RQ1. The random configurations from the feature model are created with a *RandomConfigurationGenerator* which is a class from FeatureIDE. All random configurations are randomly generated valid configurations for the feature model. For the direction product configurator against feature model, we create random configurations from the product configurator. We implemented an own functionality for that use case. The implemented function ensures that the random configuration is valid by using the domain knowledge whether a configuration is valid or not. The decision whether a feature is selected and in the random configuration has the probability of 50%. If the feature should be selected and needs other features to be selected, these additional features are randomly and automatically selected.
We create 1000 random configurations for each created feature model. We can have between one and six different feature models. That depends on which selection strategies are used and whether simple constraints are used or not. We consider all created feature models (six) for a printer from the product configurator. Overall, we create 6000 random configurations for one printer for each testing direction (product configurator against feature model and the other way around).
FeatureDiagramSynthesizer simplifies the feature model by removing tautologies and redundant constraints. Moreover, we use functionalities of FeatureIDE to gain all dead features of a feature model. We count the different constraints and dead features separately to answer RQ3.

5.2.3 Execution Environment

The prototype is implemented in Java. It is necessary to run the program in a Java Virtual Machine (JVM). We use the Java version 8 and the program can be executed in an Eclipse environment or as runnable jar file. The used computer for the executions and tests has an I7 6700k processor by Intel with 4 GHz and works with 16GB RAM on a Windows 10 system. Moreover, it is necessary that a stable internet connection is established. The connection is realized with a LAN connection. We use the version 63.0.1 of Mozilla Firefox to establish the access to the configurator website. We use the Selenium version 3.14.0. Using the Geckodriver

https://doi.org/10.24355/dbbs.084-201812111501-0

requires adding a variable to the environment variable "Path". The path to the file which contains the Geckodriver have to be added to the "Path" variable. We use version 0.23 of the Geckodriver.

5.3 Results

We present our results after the execution of the program in this section. Moreover, we answer the research questions and interpret them concerning our results. The tests were executed on November 6, 2018 for the printers MP_C4504ex_C6004ex, SP_C360SFNw_C361SFNw, and MP_C501Sp and on November 7, 2018 for the printers MP_305SPF and SP_8400DN.

The re-engineering process is always executed twice as in our concept described to ensure that the feature model is the same and our algorithm is deterministic. In only one case, we had a mismatch between the first two runs of FeatureDiagramSynthesizer. As in our concept described, FeatureDiagramSynthesizer is executed a third time to compare the first two runs. The third run was the same as the second and therefore we could use the second or third run for the evaluation.

In the diagrams below, we present the different results (exactness, effort factor, and anomalies for the feature model) for our strategies:

- one enabled with complex constraints
- biggest configuration with complex constraints
- combination of both selection strategies with complex constraints
- one enabled with simple constraints
- biggest configuration with simple constraints
- combination of both selection strategies with simple constraints

5.3.1 Exactness of the Re-Engineered Feature Model

We evaluate a feature model concerning exactness with the false positives and false negatives of the random configuration tests to answer RQ1. We tested 1000 random configurations for both testing directions for each strategy for each printer we tested. Moreover, we answer RQ1 in this section.

In Figure 5.2 we present the results for false positives and false negatives for the printer MP_C501SP. The best false positive value is reached by the strategies one enabled and biggest configuration with complex constraints with approximately 7%. The strategies biggest configuration and combination with simple constraints have a very high value of false positives (100%) which means zero valid configurations from the product configurator in the feature model and a very bad result. The

https://doi.org/10.24355/dbbs.084-201812111501-0

strategy one enabled with simple constraints also has a very bad result with 74% false positives. However, the strategy combination with complex constraints reaches a good value with 30% for the false positives. The results for false negatives are very bad (>85%) for all strategies except the simple one enabled strategy which have a good result of approximately 30%. The data set used for the diagram creation is described in Table A.1.

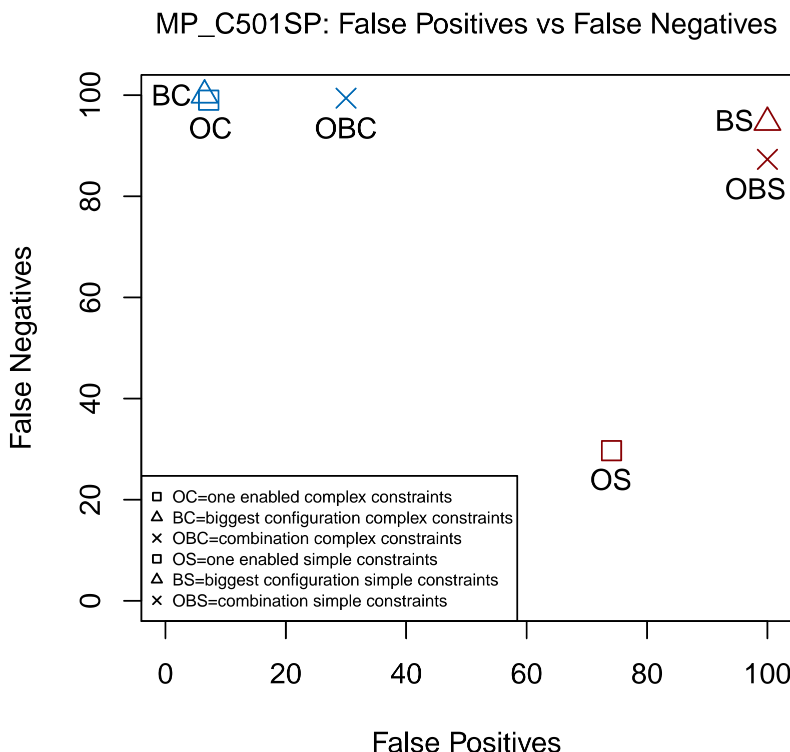

Figure 5.2: MP_C501SP: False Positive vs False Negatives

The false positives and false negatives for the printer MP_C4504ex_C6004ex are shown in Figure 5.3. The strategies which use complex constraints have a perfect value for the false positives with 0%. The strategies which use simple constraints have a very bad value for the false positives with about 65% for one enabled and 100% for biggest configuration. For the strategy combination with simple constraints, we could not test enough random configurations because of the low number of possible configurations (2). Both configuration were invalid and the false positive value would be 100%. However, the false negatives are very bad with nearly 100% for every strategy. The data set used for the diagram creation is presented in Table A.2.

In Figure 5.4, we present the test results for the exactness of the feature model of the printer SP_C360SFNw_C361SFNw. For this printer, we have a very bad result for false positives with 100% for all strategies. The results for the false negatives are well balanced for all strategies with approximately 50%. The data set used for the diagram creation is shown in Table A.3.

https://doi.org/10.24355/dbbs.084-201812111501-0

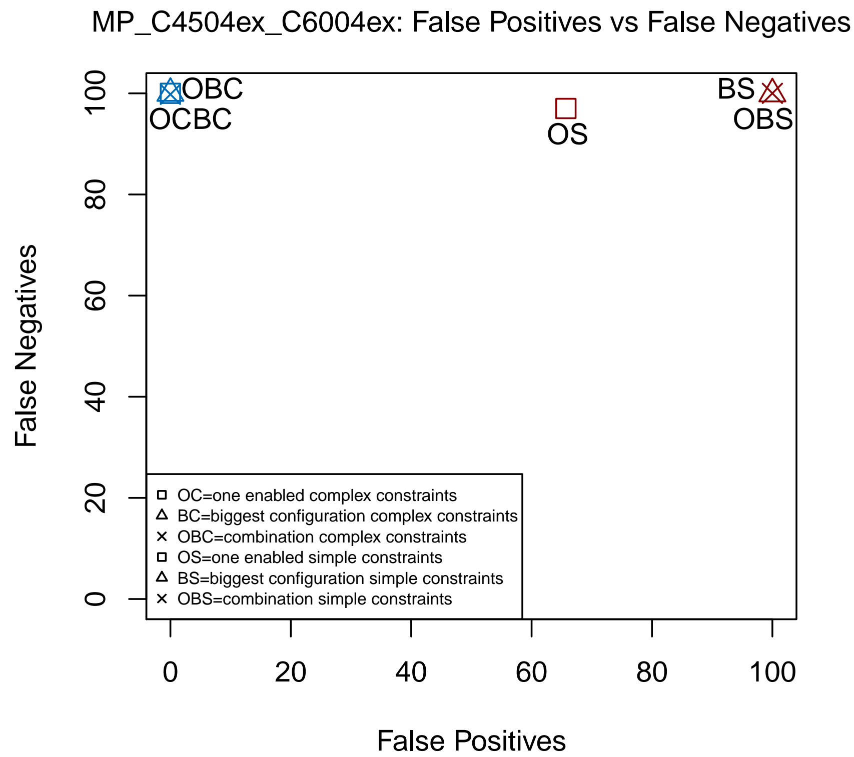

Figure 5.3: MP_C4504ex_C6004ex: False Positive vs False Negatives

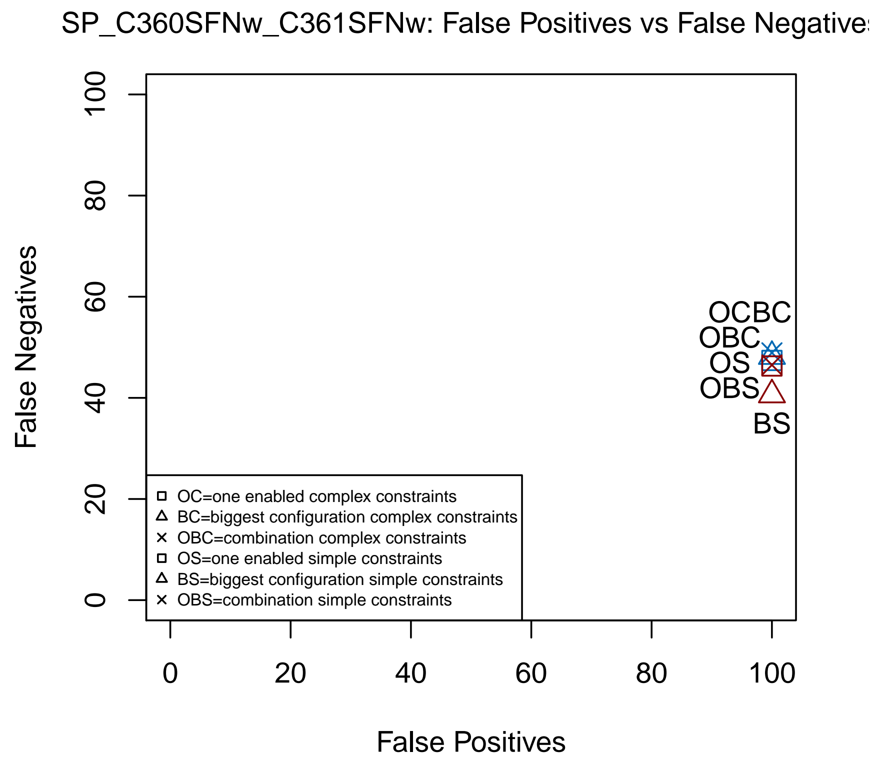

Figure 5.4: SP_C360SFNw_C361SFNw: False Positive vs False Negatives

https://doi.org/10.24355/dbbs.084-201812111501-0

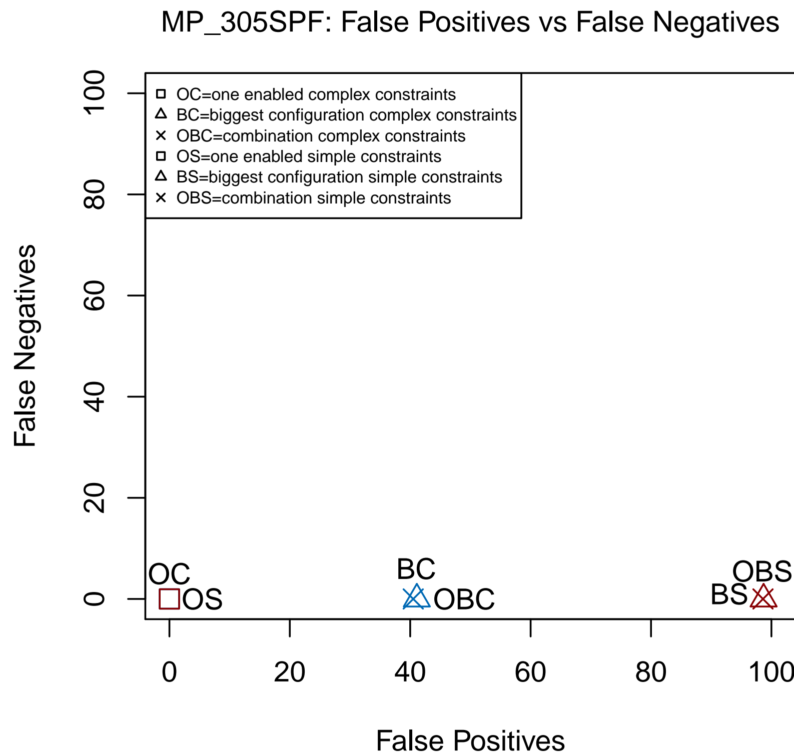

Figure 5.5: MP_305SPF: False Positive vs False Negatives

The results for the printer MP_305SPF are shown in Figure 5.5. For the false negatives we achieved a perfect result with 0% for all six strategies. The values for false positives are also perfect for both one enabled strategies. The strategies biggest configuration and configuration with complex constraints have a bad result with about 40% and biggest configuration and combination with simple constraints have a very bad result with nearly 100%. The data set used for the diagram creation is shown in Table A.4.

We present the results for the printer SP_8400DN in Figure 5.6. The false negatives for the strategies biggest configuration and combination with simple constraints cannot be rated because only one random configuration could be tested. There are too many dead features in the feature model. The only test was invalid and therefore we have a false negative value of 100%. In general, the false negatives are very bad with a best value about 92% for the one enabled strategy with simple constraints. The true negatives are very bad except the strategy biggest configuration with complex constraints which has a bad result of 57%. The data set used for the diagram creation is presented in Table A.5.

We have very different results for the different printers regarding exactness of the reverse engineered feature model. In some cases, we only have a few false positives but many false negatives for the different strategies, such as for the printer MP_C501Sp for the strategies which all use complex constraints. However, we have a feature model (printer: MP_305SPF) which has a perfect score of 0% for false negatives for all strategies. Moreover, for the same model, the false positive values are perfect for

https://doi.org/10.24355/dbbs.084-201812111501-0

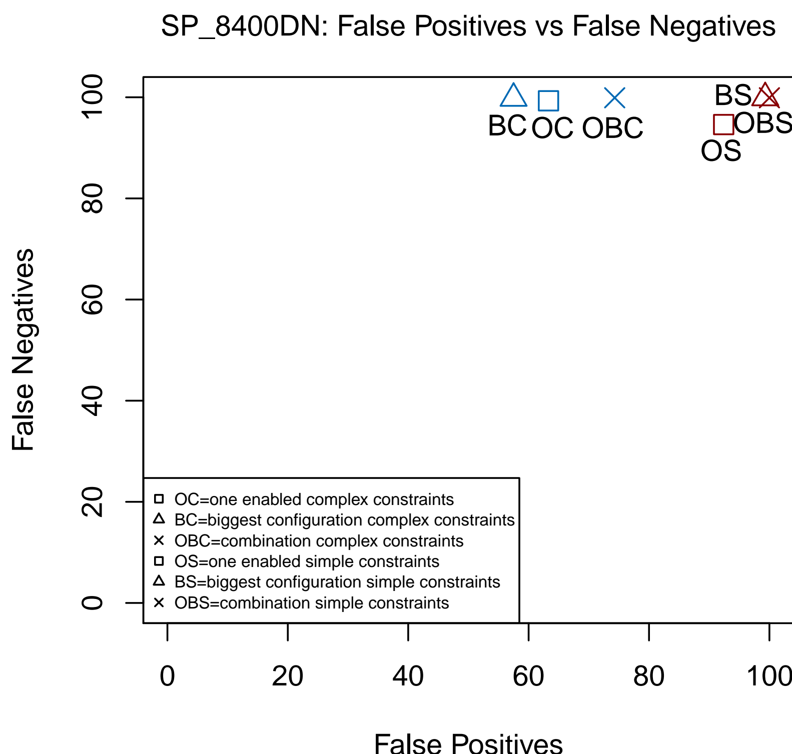

Figure 5.6: SP_8400DN: False Positive vs False Negatives

both one enabled strategies. The perfect score could follow from less constraints in the product configurator. There could only be a few restrictions for the configuration. However, for the model MP_305SPG, we can say that the one enabled strategy re-engineers the feature model exactly. We have models for which it was impossible to create more than one or two random configurations because the most features are dead in the corresponding feature models. We discuss these anomalies in Section 5.3.3. In four of the five tested printers the strategy one enabled with simple constraints reaches the best value for false negatives test configurations. However, the corresponding false positive values are much worse.

The following discussions to rate the exactness of our results use the rating system introduced in Table 5.1. There are two main factors which influence the created model, the selection strategy and the kind of constraint. The bad results for false negatives could result from too weak constraints which are created with the complex constraints. However, the weak constraints allow many configurations. Therefore, the false positives are very good in comparison to the false negatives, such as for the printer MP_C501Sp (cf. Figure 5.2 and MP_C4504ex_C6004ex (cf. Figure 5.3). The simple constraints are more restrictive than the complex constraints. That is the reason why the false negatives are better for strategies with simple constraints than for the strategies with complex constraints. The simple constraints limit the number of configuration possibilities more than complex constraints.
The results for false positives and false negatives for the different strategies do not

https://doi.org/10.24355/dbbs.084-201812111501-0

differ very much. The one enabled and biggest configuration strategy differ only in a few percent about 5% or less except one printer (MP_305SPF). In that one case, the one enabled strategy was much better than the other strategies (more than 40%). The combination of both strategies does not improve the results.
The number of features does not affect our results. We tested different printers with a different number of features, such as a smaller product with 43 features (SP_8400DN) and 93 features (MP_C4504ex_C6004ex). There are no recognizable dependencies from false positives and false negatives for the strategies in comparison to the number of features.

The exactness of our re-engineered feature models is worse than expected. We have either good results for false positives or good results for false negatives, but only for one tested printer (MP_305SPF) a very good result for both together for a specific strategy. It seems that our chosen strategies regardless of the kind of constraints are not able to reverse engineer a feature model in an exact way. The strategies either create a feature model which allows configurations which are invalid in the product configurator or forbids configurations which are valid in the product configurator. However, for printer MP_305SPF the one enabled strategies create an exact feature model. We conclude our algorithm is basically able to re-engineer a feature model but we possibly chose the wrong selection strategies. The biggest configuration strategy and the combination of one enabled and biggest configuration have worser results than the one enabled strategy. Therefore, we can say it is not meaningful to use the biggest configuration strategy and strategies which include the biggest configuration strategy. Based on our results, only the one enabled strategy has the potential to re-engineer a feature model with a high exactness.
The kind of constraints have different influence to the false positives and false negatives. The used selection strategy does not affect this influence very much. If we use complex constraints, the false positives have better results but the false negatives are very bad. If we use simple constraints, the false negatives have better results but the false positives are very bad.
We propose to evaluate the one enabled strategies with a different product configurator to exclude that the Ricoh configurator is responsible for the results because the one enabled strategies had the best results in our tests. We additionally propose to use other selection strategies, such as one-disabled, or develop own strategies for a product configurator. Our algorithm needs improvements concerning the selection strategies and possibly another kind of constraint to increase the exactness of the re-engineered feature model.

5.3.2 Effort for Re-Engineering a Feature Model

We measure the effort with the number of interactions which are necessary to reverse engineer a feature model to answer RQ2. The feature models are created on November 6, 2018. We set the number of interactions into relation to the number of features for the specific printer we tested to calculate the effort factor. The effort factor is calculated as follows: number of clicks divided by number of features.

https://doi.org/10.24355/dbbs.084-201812111501-0

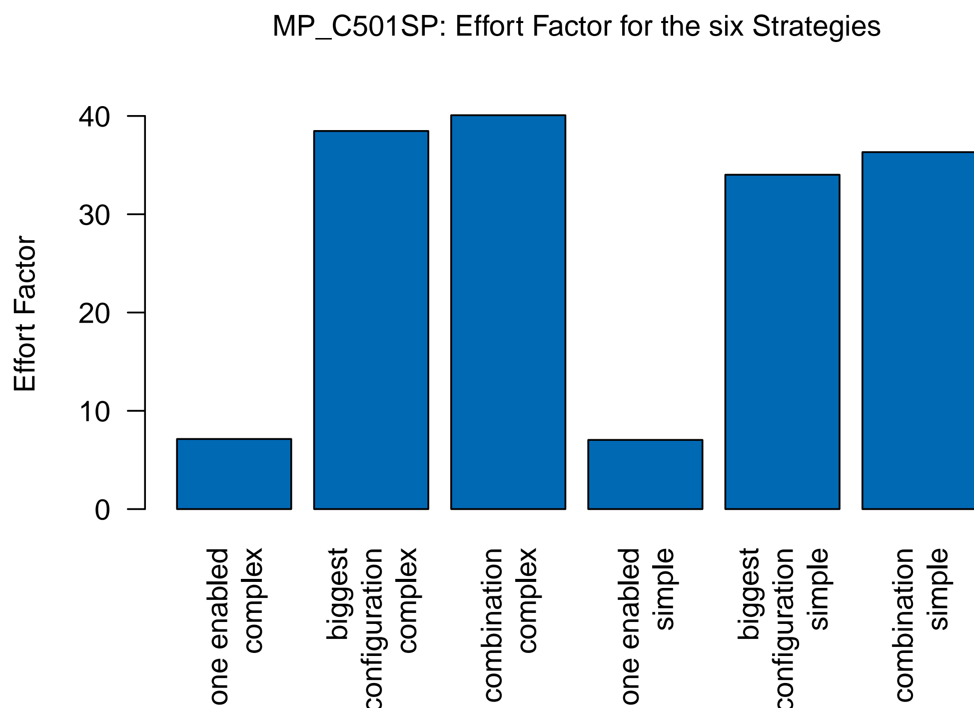

Figure 5.7: MP_C501SP: Effort factor

Figure 5.7 shows the effort for the printer MP_C501Sp. The one enabled strategies nearly have the same effort factor of approximately 7. The biggest configuration strategies have an effort factor which is about five times bigger than for the one enabled strategies. The combination strategies have an effort factor which is about plus 2 in comparison to the biggest configuration. However, the strategies biggest configuration and combination with simple constraints have a smaller effort factor than the corresponding strategies which use complex constraints. The difference is approximately 4. The data set we used to create the diagram is shown in Table A.6.

We present the effort factor for the printer MP_C4504ex_C6004ex in Figure 5.8. The effort factor for the two one enabled strategies are nearly the same with about 8. The two biggest configuration strategies also have nearly the same effort of about 56. The difference is only 1.64. The combination strategies do not differ. Both have an effort factor of 60. The effort factor for the combination strategies in comparison to the biggest configurations is 3 (with complex constraints), respectively 5 (with simple constraints). The difference between the one enabled strategies against the biggest configuration is about 7 times. Table A.7 shows the absolute values which are used to create the diagram.

Figure 5.9 presents the effort factor for the printer SP_C360SFNw_C361SFNw. The effort factor is not affected by the kind of constraints which are used by the strategies. The factor is the same for the one enabled (5.41), biggest configurations (21.02), and combination strategies (24.51). The effort for the biggest configuration strategy is about 5 times bigger compared to the one enabled strategy. The combination

https://doi.org/10.24355/dbbs.084-201812111501-0

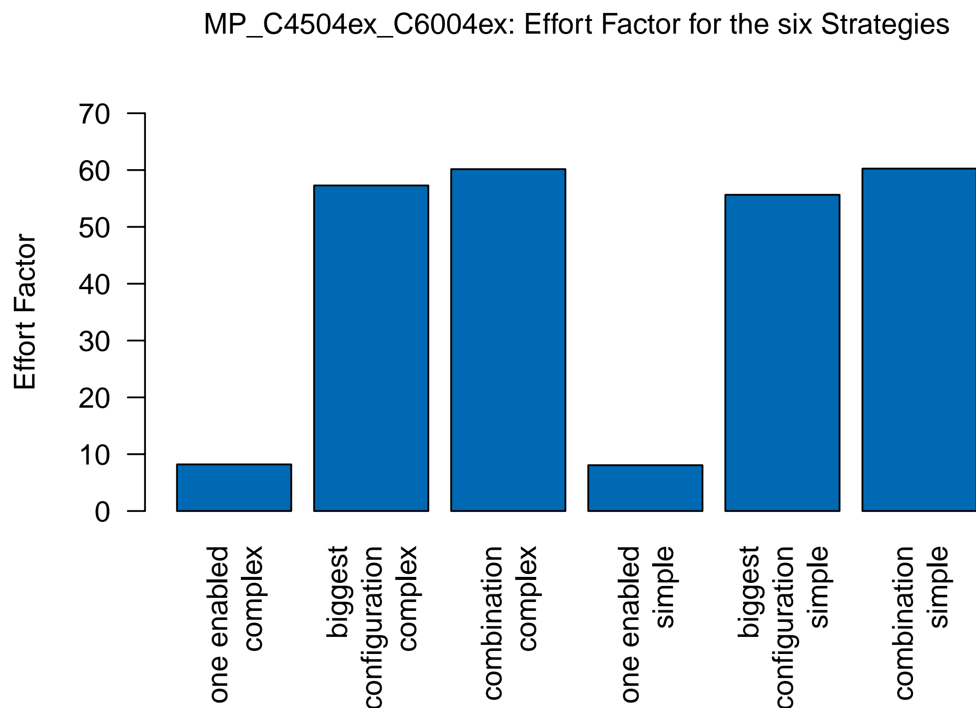

Figure 5.8: MP_C4504ex_C6004ex: Effort factor

strategy is about 6 times bigger than the one enabled strategy. The absolute values from which we calculated the effort factor are presented in Table A.8.

For the printer MP_305SPF, we show the effort factor in Figure 5.10. The effort factor for the two one enabled strategies is the same with 5.12. The biggest configuration strategy is 3 times bigger than the one enabled strategy. The difference between the biggest configuration strategies is only 0.07, so we can say that the effort is nearly the same. Both of the combination strategies have the highest effort with 18.52. The data set we used to create the diagram is shown in Table A.9.

Figure 5.11 presents the effort factor for the printer SP_8400DN. For this model, we have differences between the same strategies depending on the used kind of constraint. The one enabled strategy with complex constraints has an effort of 10.42 which is bigger than the one enabled strategy with simple constraints with 9.14. The biggest configuration strategies have an effort factor which is 5.5 times bigger than the corresponding one enabled strategies with the same kind of constraint. The combination of both strategies has the biggest effort factor of 60.98 for the complex constraints and 63.09 for the simple constraints. The absolute values which we use to create the diagram is presented in Table A.10.

The effort factor is very different for the different strategies without considering the kind of constraint which is used. The difference between the effort factors is smaller than 0.2 except for one tested printer (SP_8400DN with a difference of 1.26). The one enabled strategy always has the smallest effort factor between 5 and 10. The biggest configuration strategy is always bigger than the one enabled with a 3 to 7-fold increase. The effort factor for the combination strategies is always the biggest.

https://doi.org/10.24355/dbbs.084-201812111501-0

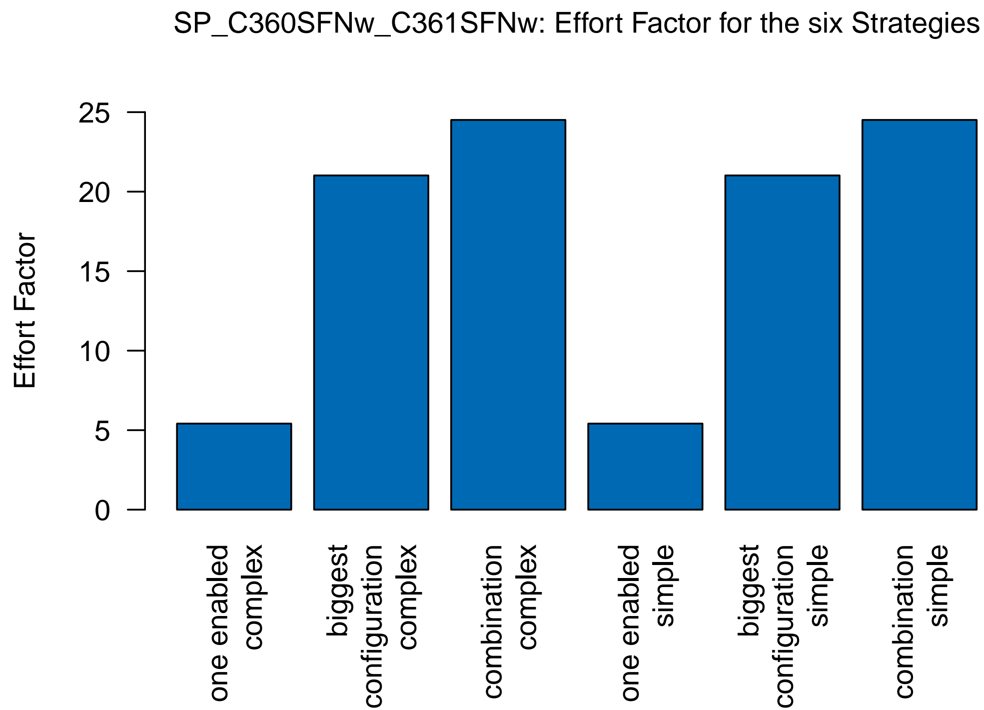

Figure 5.9: SP_C360SFNw_C361SFNw: Effort factor

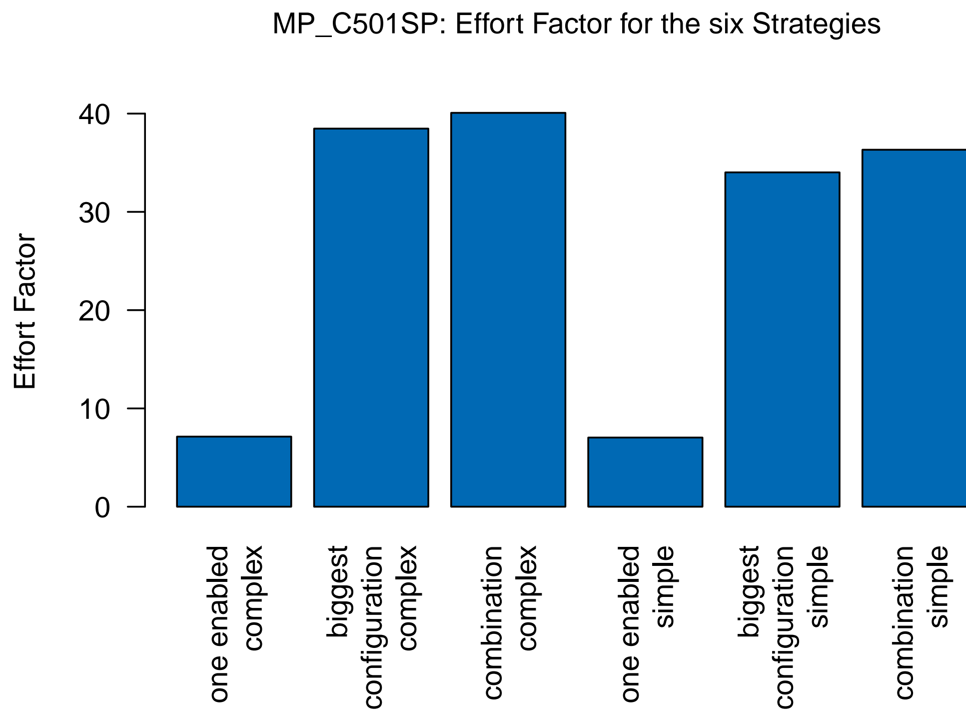

Figure 5.10: MP_305SPF: Effort factor

https://doi.org/10.24355/dbbs.084-201812111501-0

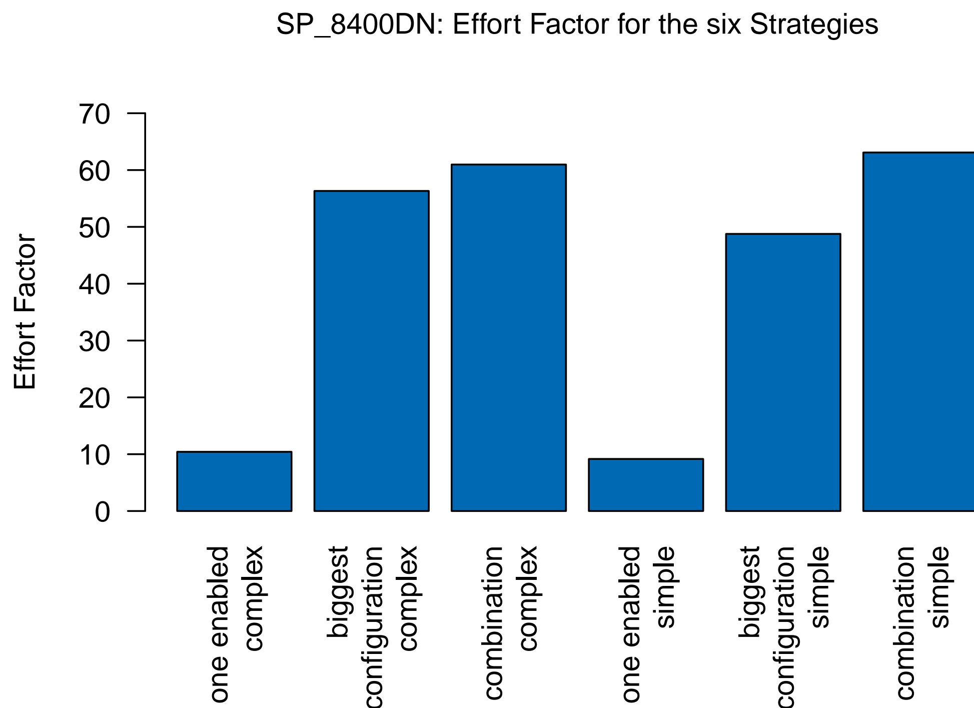

Figure 5.11: SP_8400DN: Effort factor

The difference between the biggest configuration and combination is between 2 and 5 except in one case the difference is 15 (cf: Table A.10).
The kind of constraint have not a big effect to the different strategies. The one enabled strategy is not affected, the values are nearly the same. The biggest configuration strategy effort factor is smaller for the simple constraints. The difference is up to 8 but mostly smaller than 4. The combination strategy effort is also smaller for the simple constraints except one case where it is bigger with a difference of about 2.

The effort to readout a feature model from a product configurator hardly depends on the chosen selection strategy. The kind of constraints which are used have a small effect. The strategies with simple constraints have a better effort factor but the difference is not so big. The one enabled strategies have the smallest effort factor and the combination strategies the biggest. The number of features does not directly affect the effort factor. We have printers with nearly the same number of features (MP_305SPF (42) and SP_8400DN (43)) and very different effort factors. For the printer SP_8400DN the effort factor for the one enabled strategy is 2 times bigger and the biggest configuration and combination strategy effort factor is nearly 4 times bigger than for the printer MP_305SPF. The reason could be many dialogs which open during the adding constraints step of FeatureDiagramSynthesizer which results in more required interactions and possible constraints in the model.
We also tested a printer with 93 features. The effort factor for the one enabled for that printer are not significantly bigger than the other printers. The biggest configuration and combination strategy effort factor is bigger in comparison to the other

https://doi.org/10.24355/dbbs.084-201812111501-0

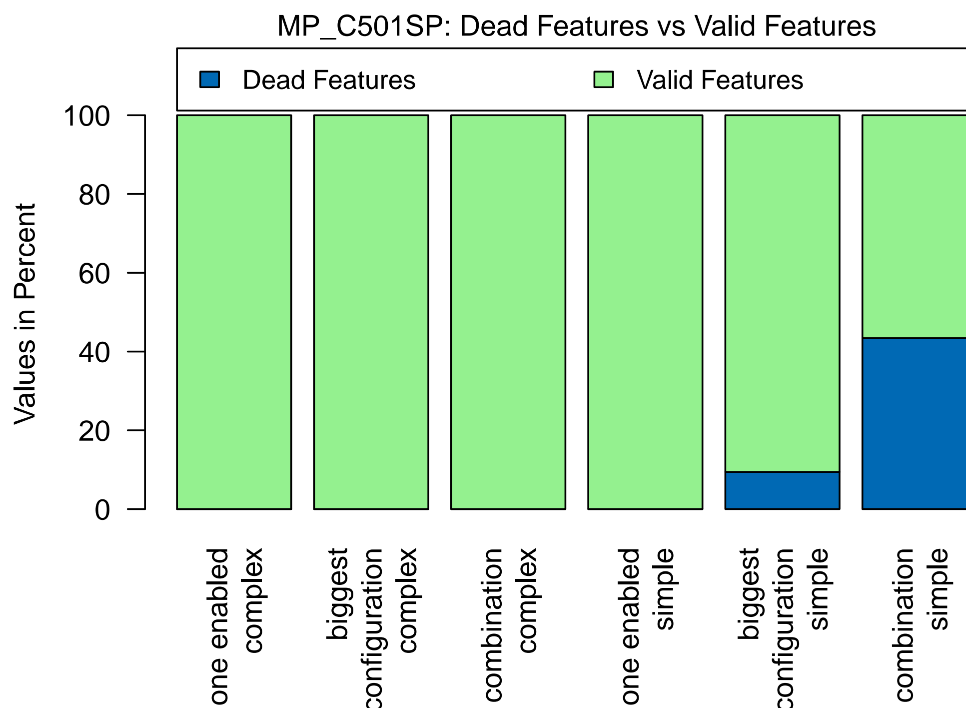

Figure 5.12: MP_C501Sp: Ratio of Dead Features

tested printers but in one case they are nearly the same. Therefore, we assume that the number of features does not affect the effort factor in a mathematically describable way. However, we assume a small raise for the selection strategy biggest configuration because this strategy tries to select as much as possible for the actual configuration. This selection strategy always ends in more interactions.

5.3.3 Anomalies in a Re-Engineered Feature Model

In this section, we describe the anomalies we found during the re-engineering of a feature model. We consider dead features, tautologies, and redundant constraints. We answer RQ3 in this section.

We present the overview about the dead features for the printer MP_C501Sp in Figure 5.12. The strategies which use complex constraints and the one enabled strategy with simple constraints have no dead features. The strategy biggest configuration with simple constraints has 5 dead features which are 9.4% of all features for that printer and the strategy combination with simple constraints has 23 dead features which are 43.3% of all features. The data set we used for the diagram creation are shown in Table A.11.

The overview about dead features for the printer MP_C4504ex_C6004ex is shown in Figure 5.13. The strategies which use complex constraints have no dead features. The other three strategies with simple constraints have at least one dead feature

https://doi.org/10.24355/dbbs.084-201812111501-0

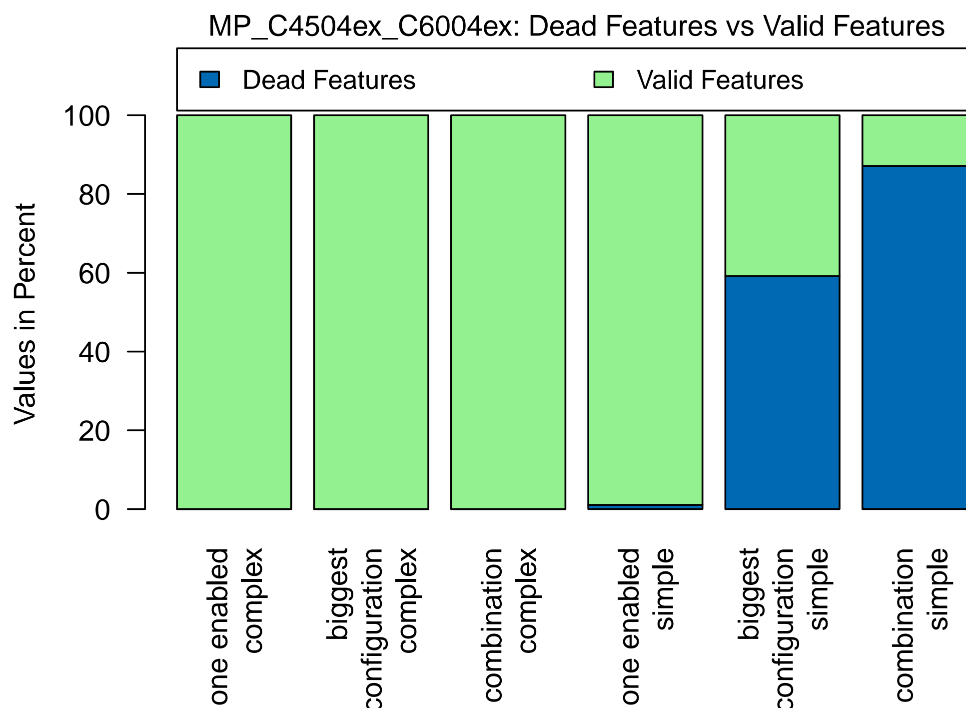

Figure 5.13: MP_C4504ex_C6004ex: Ratio of Dead Features

as in the one enabled strategy. The strategy biggest configuration with complex constraints has nearly 60% dead features which are 55 features. The combination strategy with simple constraints has 81 dead features which are 87.1% of all features of that printer. The data set we used for the creation of the diagram are presented in Table A.12.

Figure 5.14 presents the number of dead features for the printer SP_8400DN. Only the strategy biggest configuration and combination with simple constraints produce dead features. The biggest configuration with simple constraints has one dead feature which is 2.3% of all features. The feature model which is created by the combination with simple constraints has 79.1% (34) dead features. The data set we used to create the diagram is presented in Table A.13.

The printers SP_C360SFNw_C361SFNw and MP_305SPF have no dead features created during the readout process.
The dead features result from to restrictive constraints. They only occur if a selection strategy in combination with simple constraints is used. We have no dead features if complex constraints are used. The one enabled strategy with simple constraints creates only a few dead features in relation to the biggest configuration or the combination of both strategies with simple constraints. The combination strategy especially creates many dead features. We have a model with 87% dead features (printer MP_C4504ex_C6004ex). In such models a configuration is hardly restricted and these feature models do not represent a product configurator in a good way. The combination of the one enabled strategy and biggest configuration with simple constraints should not be used to re-engineer a feature model from a product

https://doi.org/10.24355/dbbs.084-201812111501-0

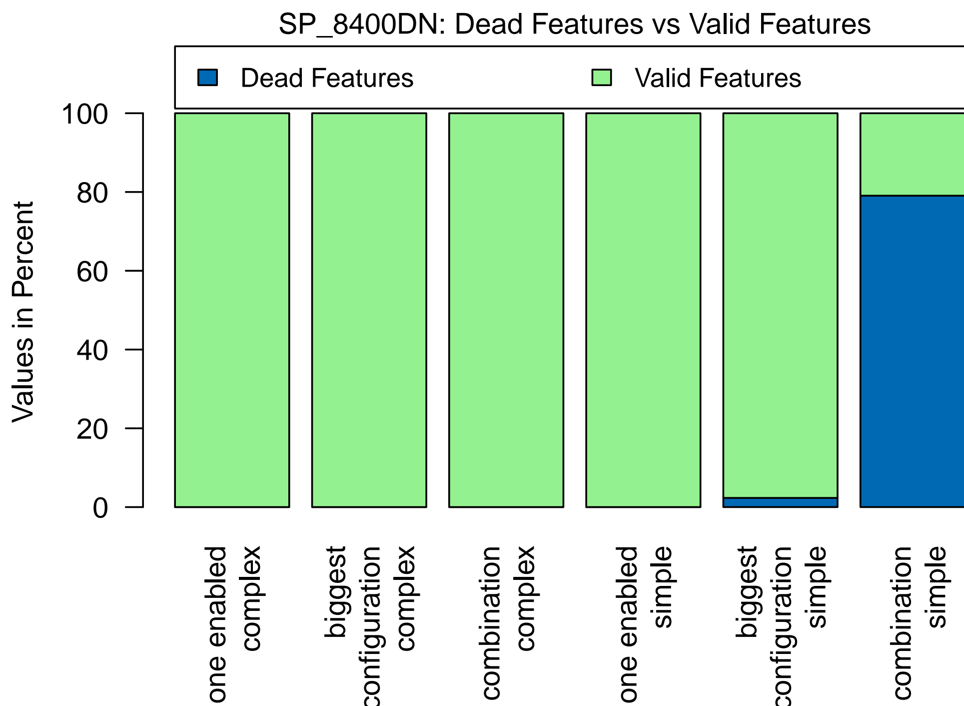

Figure 5.14: SP_8400DN: Ratio of Dead Features

configurator. The risk to create a large number of dead features is too high. The strategies which concern only complex constraints should be used if no or less dead features are preferred. We assume that these strategies do not create dead features.

We present the kind of constraints in the diagrams below. The diagrams contain tautology constraints. However, in our description, we consider the tautologies only if they have a significantly high value. In the other cases, the tautologies are counted to the redundant features.

For the printer MP_C501Sp, the relative distribution of constraints is presented in Figure 5.15. Both one enabled strategies have a good result for the real constraints with more than 55% and less than 45% of redundant constraints. The biggest configuration strategy with complex constraints finds many tautologies with approximately 50%. A quarter of the constraints are also redundant which results in 25% of real constraints. The combination with complex constraints found about 60% redundant features. About 90% of the found constraints are redundant for the strategies biggest configuration and combination with simple constraints. The absolute values are shown in Table A.14.

The relative distribution of constraints for the printer MP_C4504ex_C6004ex is shown in Figure 5.16. The strategies which use complex constraints and the one enabled strategy with complex constraints have found approximately 30% real constraints. Only the biggest configuration and combination strategy with complex constraints found a significant number of tautologies with about 25%. The strategies biggest configuration and combination with complex constraints have about

https://doi.org/10.24355/dbbs.084-201812111501-0

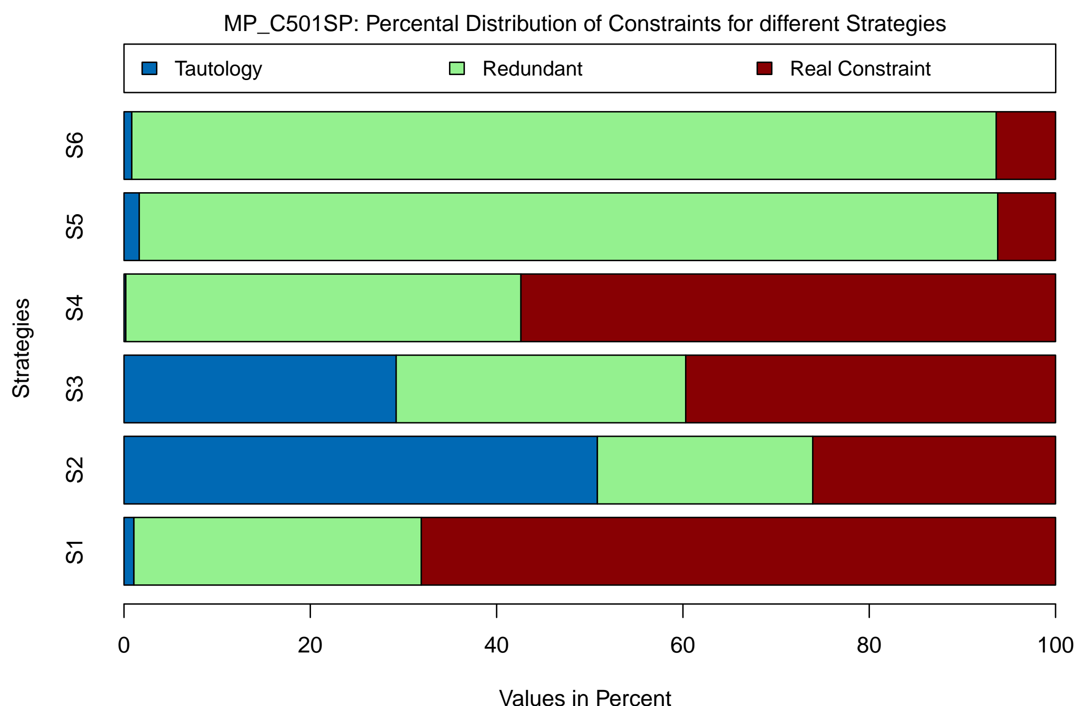

Figure 5.15: MP_C501Sp: Anomaly constraints

95% redundant features. Table A.15 presents the absolute values we used to create the diagram.

For the printer SP_C360SFNw_C361SFNw, Figure 5.17 shows the distribution of constraints. None of the strategies produce a recognizable number of tautologies. All six strategies differ between 50% (biggest configuration with complex constraints) and 25% (combination with simple constraints) of found real constraints. The absolute values are presented in Table A.16.

In Figure 5.18 we present the distribution of constraints for the printer MP_305SPF. The strategies one enabled and biggest configuration with complex constraints have about 55% redundant constraints. For the other four strategies, the redundant constraints are approximately 70%. Table A.17 describe the absolute values for the constraints distribution.

The distribution of constraints for the printer SP_8400DN is shown in Figure 5.19. The percentage of real and redundant constraints are approximately 50% for one enabled with complex constraints. The biggest configuration and combination with complex constraints have about 35%, respectively 50% of tautology constraints. The percentage of real constraints is nearly the same for both with about 30%. One enabled with simple constraints also has 30% real constraints. The strategies biggest configuration and combination with simple constraints have more redundant constraints with more than 80% and 90%. The absolute values are described in Table A.18.

https://doi.org/10.24355/dbbs.084-201812111501-0

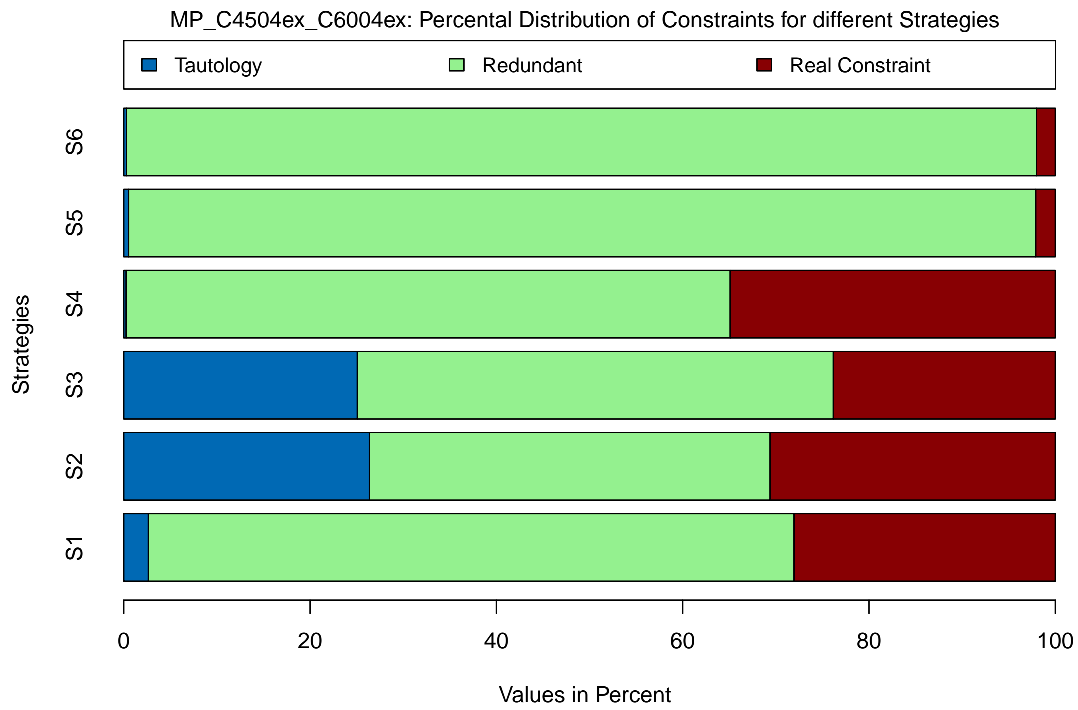

Figure 5.16: MP_C4504ex_C6004ex: Anomaly constraints

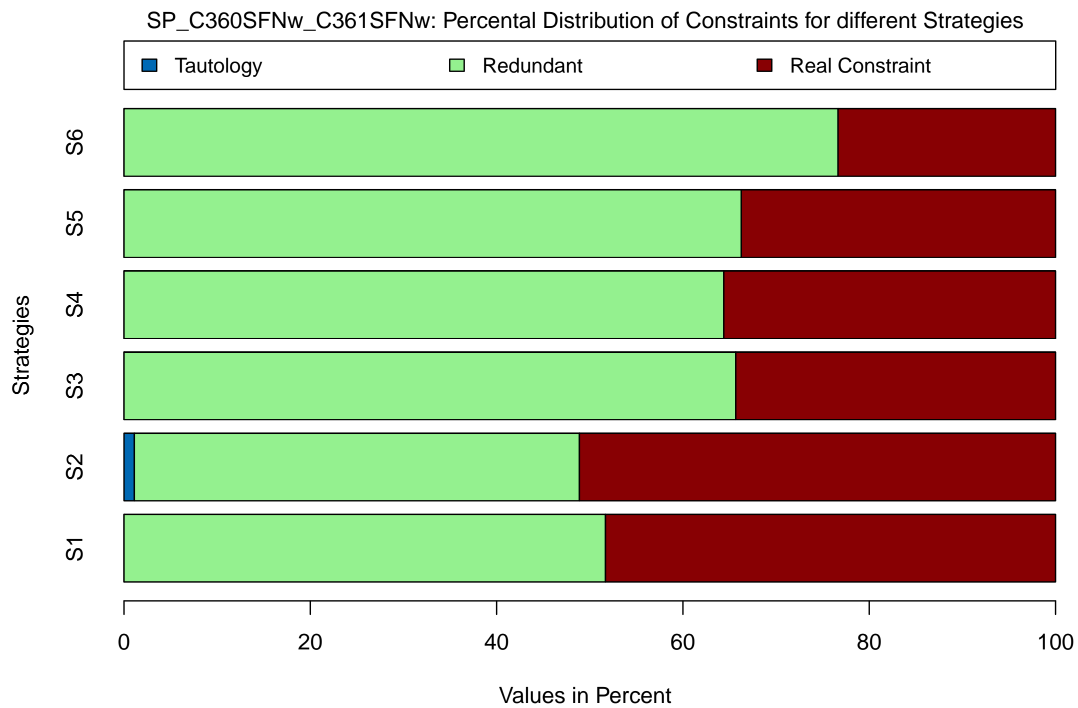

Figure 5.17: SP_C360SFNw_C361SFNw: Anomaly constraints

https://doi.org/10.24355/dbbs.084-201812111501-0

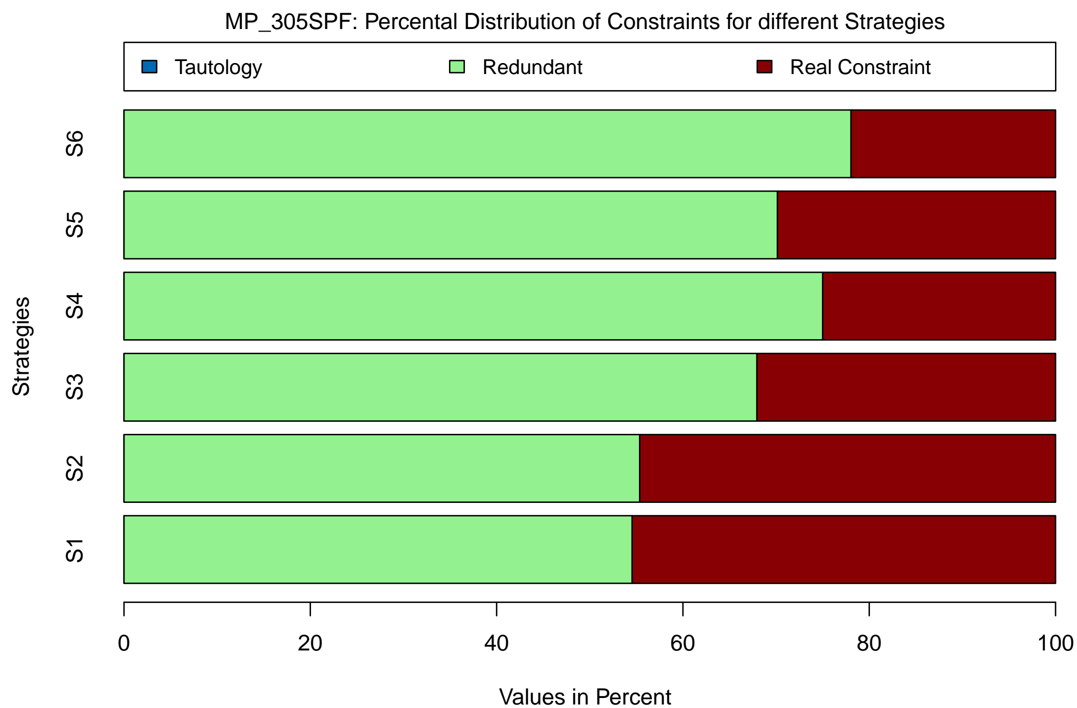

Figure 5.18: MP_305SPF: Anomaly constraints

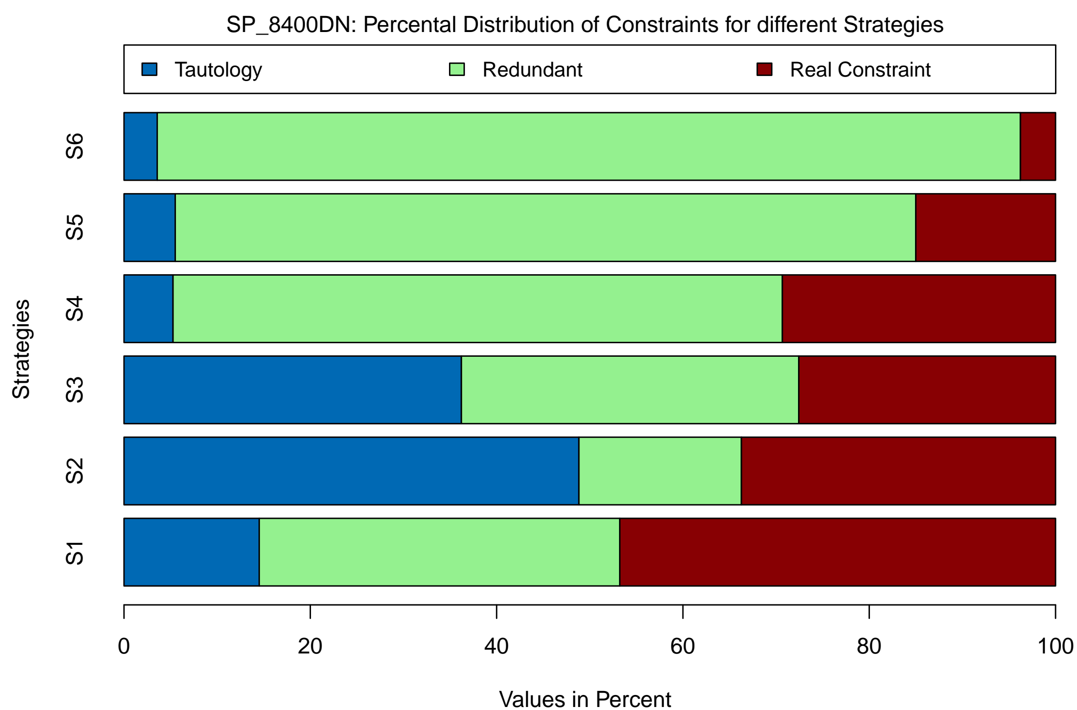

Figure 5.19: SP_8400DN: Anomaly constraints

https://doi.org/10.24355/dbbs.084-201812111501-0

Our strategies create different distributions for the constraints. It is conspicuous that only the strategies biggest configuration and combination with complex constraints sometimes create a big number of tautology constraints. These tautologies are always implications where one feature is on the left and on the right side of the implication. This could happen during the creation of the constraints from the sets of automatically selected (S_A) and deselected features (D_A). There could be a mistake in our implementation which cause the tautologies. However, there are constraints which are redundant and not necessary for the reverse engineered feature model. The strategy combination with simple constraints has the highest value for redundant constraints. This is also in most cases for the strategy biggest configuration with simple constraints. The reason is the amount of constraints which are created by the selection strategy biggest configuration. This is especially valid for the simple constraints and how we create the constraints from the sets S_A and D_A. The one enabled strategy with complex constraints often creates the fewest redundant constraints. The reason is the amount of constraints which are build during the algorithm execution. There are less constraints needed for complex constraints than for simple constraints to express the same. More information and how simple constraints can be build from complex constraints is given elsewhere [KTM+17b]. Redundant constraints have no bad influence for our feature model. All these constraints are found more than one time or are expressed by other constraints. They do not corrupt the feature model. Many redundant constraints only slow down FeatureDiagramSynthesizer because the search for redundant constraints is done with a SAT solver which is computationally expensive. However, our strategies create more than 50% of redundant features. We could improve the strategies to reduce the redundancies but it must be considered that we not miss any constraint which is not redundant. The creation of tautologies could be prevented. The algorithm can be improved so that no tautologies are created anymore. This would reduce the number redundant constraints, too.

5.4 Threats to Validity

In this section, we consider threats to validity and what can be done to alleviate the influence of the threats. We consider external validity which contains impacts from the outside against FeatureDiagramSynthesizer. Moreover, we consider internal validity which concerns measures we implemented into FeatureDiagramSynthesizer to operate valid and robust.

External Validity

During the execution of FeatureDiagramSynthesizer, it could happen that the underlying model of the product configurator changes. The algorithm would not recognize this and the result would be a wrong feature model. Therefore, the re-engineering process is always executed twice to compare the results. If the two models are not the same, the re-engineering process would start again at maximum of two times which result in a maximum of 4 runs to reverse engineer a feature model. The different models are compared among themselves to check if they are the same.

https://doi.org/10.24355/dbbs.084-201812111501-0

The Internet connection must be stable to use FeatureDiagramSynthesizer. Our algorithm cannot handle large connection losses. The algorithm stops with an error if the connection is lost. We propose to use a LAN connection to reduce possible errors which could happen with a wireless connection.
FeatureDiagramSynthesizer is implemented in Java. We have no influence over how the JVM of Java is executed and how the Garbage Collector of the JVM operates. We assume that these functionalities of Java always operate correctly.

Internal Validity

FeatureDiagramSynthesizer interacts with a website with many clicks. After every click, the website needs some time to react before the next interaction should be executed. Therefore, we use short timeouts (<400ms) to shortly pause the algorithm. We reduce issues which could occur if too many interactions are executed too fast and we minimize the risk to miss information the website gives to the algorithm, such as an opening dialog.

The Algorithm with the selection strategies is implemented by ourself. Even if we implemented the algorithm as proposed in Chapter 3, we cannot ensure that no errors can happen during the execution. This concerns the readout process and testing process. We reduce the possible impact by testing all of our implementations.

During the interaction with the website, it sometimes happened that an exception is thrown by Selenium. These exceptions have no influence on the result but they would interrupt the algorithm. FeatureDiagramSynthesizer is robust against exceptions, considers these kind of exceptions, and reacts. The algorithm is not interrupted and continues its operation.

We implemented a logging into the algorithm. With the logging it is always documented what FeatureDiagramSynthesizer does, such as which constraint is added. It is always comprehensible how the feature model is created. If errors, such as exceptions occur, the errors are logged.

5.5 Summary

We evaluated our developed algorithm FeatureDiagramSynthesizer in this section. We presented the subject systems we use for the evaluation and described the experiment setup. We created random configurations to test our recreate feature models for five different printers of Ricoh. The tests were executed for six different strategies we introduced in Chapter 3. The tests are executed with the prototype of FeatureDiagramSynthesizer.

We defined three research questions for our evaluation. We measure the exactness of the re-engineered feature model by calculating false positives and false negatives from our tests in the first research question. In the second research question,

https://doi.org/10.24355/dbbs.084-201812111501-0

we measure the effort to re-engineer a feature model from a product configurator. Therefore, we count the number of interactions which are necessary for the execution of FeatureDiagramSynthesizer and sets them in relation to the number features the product has. The third research questions considers anomalies in the resulting feature model. We investigate how many dead features are created and how the distribution of redundant and real constraints is in the feature model. All research questions consider the six different strategies we presented.

The best results, we achieved with the one enabled strategy. Even though our results are not good for different products from the Ricoh configurator, our strategies (especially one enabled) could be used with other product configurators to evaluate their exactness. However, the one enabled strategies have less effort compared to the other strategies we used, such as biggest configuration. The kind of constraints which should be used depends on the product configurator. Complex constraints allow more configurations in the feature model compared to simple constraints. The configurations could be invalid in the product configurator. The simple constraints are more restrictive. This could lead to invalid configurations in the feature model which are valid in the product configurator. Moreover, simple constraints could create dead features in the feature model.

We discuss threats to validity at the end of this chapter in which we consider internal and external parameters and what we do against possible influences.

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

6. Related Work

In this chapter, we present related work to this master thesis. We consider literature for product configurators and feature models.

The development of a product configurator follows different strategies. Haug et al. present three main and four additional strategies to create a product configurator [HHM12]. They describe the way from knowledge acquisition to knowledge representation in form of a configurator and the involved persons such as domain experts and software configuration experts. Case studies are also presented to have real examples for the strategies. Using a feature model, as promoted in this thesis, is a further strategy to develop product configurators. With our developed algorithm FeatureDiagramSynthesizer feature models can be created from product configurators. The gained feature models could be used to create random configurations from the feature model and evaluate the original product configuration with that configurations. Possible inconsistencies in the product configurator can be detected if invalid configurations are found.

Felfernig et al. discuss how requirements from different configuration systems can be described by a standard design language like UML and a logical description [FFJ01b]. Moreover, they show a way how the UML notation can be translated into first-order logic to describe the underlying knowledge. These two point of views help to understand the complete model for domain experts and programmers of a product configuration system.
Knowledge base (e.g., used for product configurators) grows strongly in size and complexity. Felfernig et al. [FFJ01a] present a technique for designing knowledge-based configuration systems. They start with a representation of knowledge on a conceptional level and translate it into a standard formal representation for configuration systems. On that basis UML is used as one design technique. The UML representation and a reverse engineered feature model are both a possible representation formats for domain knowledge of a product configurator. FeatureDiagramSynthesizer creates feature models which can be translated into a propositional logic

https://doi.org/10.24355/dbbs.084-201812111501-0

representation. This representation could be compared to the logic of the UML notation which Felfernig et al. propose to check whether both representations are the same for the same product configurator. With these representations (UML, logic representation and feature model), it is possible to check consistencies of a product configurator. If the consistency is checked during the development phase of a product configurator, the knowledge base can be improved and failures can be avoided.

Abbasi et al. present a method the reverse engineer web configurators [AAHC14]. They developed a tool-supported process to semi-automatically extract data from a product configurator. They use formal models to safe the extracted data. They implemented a web wrapper to gain structured data and a web crawler to get knowledge about the features. In comparison to Abbasi et al., we use the same approach to interact with a product configurator. In both cases, the interaction of a user is simulated. However, our approach results in a feature model. The process how the data are collected is also different. Abbasi et al. use pattern matchings to find data in the HTML code which they needed. Our algorithm analyses the HTML code only concerning potential features by using Selenium functionalities and not pattern matching algorithms. Moreover, our algorithm tries to find constraints from the product configurator by dynamically interacting with the website. The constraints are found by different selection strategies we implemented. In comparison to FeatureDiagramSynthesizer Abbasi et al. try to find constraints by analyzing the HTML code with there pattern matching algorithms. They derive kind of constraints, such as alternatives groups, of the code.

https://doi.org/10.24355/dbbs.084-201812111501-0

7. Conclusion and Future Work

With this chapter, we end this master thesis. We summarize the master thesis: Re-Engineering Feature Models from Product Configurators. Moreover, we present ideas for a future work based on our developed concept and prototype.

7.1 Thesis Conclusion

In this master thesis, we investigate whether it is possible to re-engineer a feature model from a web product configurator. We developed an algorithm called FeatureDiagramSynthesizer to try to support the overall vision is to unify product configuration and software configuration (e.g., feature models) and use properties and advantages from both sides.

FeatureDiagramSynthesizer has different assumptions which must be fulfilled before it can be executed because there exist many kinds of product configurators. Based on the assumptions, we formalize the configuration process. We define different sets and states for a configurator. The sets follow from our assumption, e.g., set of all features. With the sets and states, we define the algorithm FeatureDiagramSynthesizer. The algorithm consists of two main steps. The first step creates the feature diagram without constraints and is static and does not interact with the website of the product configurator. The second step adds constraints and is dynamic. For adding constraints the algorithm interacts with the website by using selection strategies. The constraints base on the sets we defined before. We use two kinds of of constraints, simple and complex constraints. At the end of the algorithm execution the feature model is simplified by removing redundant constraints.

We implemented a generic prototype of FeatureDiagramSynthesizer with a concrete implementation of the Ricoh printer configurator. For the implementation we use additional tools, such as Selenium, geckodriver, and FeatureIDE. The implementation realizes the main parts of FeatureDiagramSynthesizer which are creating the feature model, adding constraints, and simplifying the model. We implement two

https://doi.org/10.24355/dbbs.084-201812111501-0

different selection strategies and two kinds of constraint which can be created. The combination of the selection strategies is also possible. That result to six different strategies which can be used for the execution of FeatureDiagramSynthesizer.
For a concrete configurator, we implemented an adapter. We use abstract classes and interfaces for the adapter. A concrete adapter must extend the abstract class and implement the interface. With that structure we ensure a generic structure and extensibility. Furthermore, we describe the important methods which are implemented for the realization of FeatureDiagramSynthesizer.

To evaluate the algorithm, we used our prototype for the printer configurator of Ricoh. We tested five different printers. Our tests use random configurations which are created by the feature model and product configurator. The tests are executed for every of the six strategies. We introduce three research questions to measure the exactness, effort to re-engineer a feature model from a product configurator, and how many anomalies occur in the feature model after the reverse engineering process. Moreover, we consider threats to validity.

Overall, we develop a first prototype which is only basically able to re-engineer feature models from product configurators. The exactness of the created feature models is worse than we expected. The results differ very much between different strategies which is expressed by high values in false negatives and false positives. For the most test printers we reverse engineered, either the false positives have good results or the false negatives have good results. Only in one test model we achieve a very good result for both exactness factors for the same strategy. FeatureDiagramSynthesizer should be tested with another product configurator to exclude that the Ricoh configurator causes the bad results. A comparison between different configurators helps to evaluate our developed algorithm. The bad results could arise from the chosen selection strategies. Other selection strategies should be implemented and tested for FeatureDiagramSynthesizer.
Our concept and implementation can be extended by new strategies. It is possible to add more adapters and selection strategies to the algorithm. Both should be done in the future to create more feature models and to test if more exact feature models than our test models can be created. The algorithm FeatureDiagramSynthesizer could be used as base to create more and bigger feature models from product configurators.

7.2 Future Work

In this master thesis, we developed and implemented the algorithm FeatureDiagramSynthesizer. This algorithm can actually be executed with six different strategies. Moreover, the actual prototype is implemented with a concrete adapter for the Ricoh configurator. FeatureDiagramSynthesizer can only operate to product configurators which fulfill all assumptions we made in Chapter 3.

New Adapter

We implemented an adapter for the Ricoh printer configurator to evaluate our concept. That is the first adapter which is implemented for FeatureDiagramSynthesizer.

https://doi.org/10.24355/dbbs.084-201812111501-0

We created generic classes to make our implementation extensible. We help developer with the implementation of new adapter by giving them the needed methods and variables for a concrete adapter.
A new adapter can be implemented for all product configurator which fulfill the assumptions of FeatureDiagramSynthesizer. It is meaningful to implement more adapter to proof our concept with other configurators than Ricoh.

Expand FeatureDiagramSynthesizer

We have several assumptions for FeatureDiagramSynthesizer so that the algorithm operates correctly. Some of the assumptions can be removed if we expand the algorithm of FeatureDiagramSynthesizer. Assumption 4 can be removed or lessen if more dialogs are supported and not only basic dialogs. Assumption 8 can be removed if the algorithm can analyze the product configurator with more than one step. However, the implementation effort is high to remove that assumption because in the current implementation status only one step is considered.

New Selection Strategies

We present a variety of selection strategies in Section 3.3.3. We only implemented the one enabled strategy and the biggest configuration strategy. However, FeatureDiagramSynthesizer is extensible for more selection strategies. A new strategy can be implemented into the algorithm. We proposed to use the one enabled strategy to execute FeatureDiagramSynthesizer but we not know whether there are better strategies. It is necessary to implement new strategies to check this and find the best selection strategy or combination of them. It could also be meaningful to develop own selection strategies with focus on product configurators.
It is also possible that one selection strategy is only useful for one configurator and not for others. For that reason more selection strategies are meaningful. It would be possible to select out of a pool of different selection strategies for a specific product configurator.

New Kind of Constraint

FeatureDiagramSynthesizer uses two kinds of constraints to reverse engineer a feature model. The creation of the constraints depends on the sets of actual selected, automatically selected, and deselected features. From these sets only implications are build. There might be a possibility to build other constraints or new kinds of constraints for FeatureDiagramSynthesizer which represent the rules of a product configurator in a better way than simple and complex constraints do.

Extended Feature Models

Our concept is a new source to create realistic feature models even if the concept needs improvements. Product configurators have attributes such as costs. Basic models with a large database are missing in the research. One possibility to create such models is to take a real model and add randomly attributes with the help of a probability distribution [SSA17]. A new possibility is using our developed algorithm.

https://doi.org/10.24355/dbbs.084-201812111501-0

FeatureDiagramSynthesizer could be extended with new functionalities which consider attributes of features. There is implementation effort for the algorithm and for the used adapter which must find these attributes. However, FeatureDiagramSynthesizer is implemented extensible and these extension should be possible without too many changes. Basically the class *ConfiguratorFeature* of FeatureDiagramSynthesizer is already prepared to save information about attributes. The information have to be filled and converted to the feature model. With that extension it is possible to automatically create large feature models with attributes.

https://doi.org/10.24355/dbbs.084-201812111501-0

A. Appendix

In this chapter, we present data sets which are used as basis for the diagrams we created, such as false positives for the evaluation of a feature model, number of clicks, or the distribution of constraints.

Overview of valid Configurations during the Test Excecution

Strategies	Valid Configurations (Feature Model)	Valid Configurations (Product Configurator)
One enabled complex	10	928
Biggest configuration complex	0	935
Combination complex	6	700
One enabled simple	703	259
Biggest configuration complex	54	0
Combination complex	127	0

Table A.1: Valid configurations during tests for printer MP_C501SP

Strategies	Valid Configurations (Feature Model)	Valid Configurations (Product Configurator)
One enabled complex	0	928
Biggest configuration complex	0	935
Combination complex	2	700
One enabled simple	30	259
Biggest configuration complex	1	0
Combination complex	0[1]	0

[1]Only two possible configurations could be created because of the number of dead features.

Table A.2: Valid configurations during tests for printer MP_C4504ex_C6004ex

https://doi.org/10.24355/dbbs.084-201812111501-0

Strategies	Valid Configurations (Feature Model)	Valid Configurations (Product Configurator)
One enabled complex	526	0
Biggest configuration complex	518	0
Combination complex	511	0
One enabled simple	537	0
Biggest configuration complex	594	0
Combination complex	535	0

Table A.3: Valid configurations during tests for printer SP_C360SFNw_C361SFNw

Strategies	Valid Configurations (Feature Model)	Valid Configurations (Product Configurator)
One enabled complex	1000	1000
Biggest configuration complex	1000	589
Combination complex	1000	595
One enabled simple	1000	1000
Biggest configuration complex	1000	13
Combination complex	1000	14

Table A.4: Valid configurations during tests for printer MP_305SPF

Strategies	Valid Configurations (Feature Model)	Valid Configurations (Product Configurator)
One enabled complex	7	367
Biggest configuration complex	2	425
Combination complex	1	257
One enabled simple	54	76
Biggest configuration complex	1[1]	7
Combination complex	1[1]	0

[1]Only one possible configurations could be created because of the number of dead features.

Table A.5: Valid configurations during tests for printer SP_8400DN

Overview Effort Factor for different Printer of Ricoh

Strategy	Effort factor (#clicks)
One enabled complex	7.13 (378)
Biggest configuration complex	38.47 (2039)
Combination complex	40.08 (2124)
One enabled simple	7.04 (373)
Biggest configuration complex	34.02 (1803)
Combination complex	36.32 (1925)

Table A.6: Effort for printer MP_C501Sp

https://doi.org/10.24355/dbbs.084-201812111501-0

Strategy	Effort factor (#clicks)
One enabled complex	8.20 (763)
Biggest configuration complex	57.29 (5328)
Combination complex	60.16 (5595)
One enabled simple	8.06 (750)
Biggest configuration complex	55.65 (5175)
Combination complex	60.26 (5604)

Table A.7: Effort for printer MP_C4504ex_C6004ex

Strategy	Effort factor (#clicks)
One enabled complex	5.41 (276)
Biggest configuration complex	21.02 (1072)
Combination complex	24.51 (1250)
One enabled simple	5.41 (276)
Biggest configuration complex	21.02 (1072)
Combination complex	24.51 (1250)

Table A.8: Effort for printer SP_C360SFNw_C361SFNw

Strategy	Effort factor (#clicks)
One enabled complex	5.12 (215)
Biggest configuration complex	15.38 (646)
Combination complex	18.52 (778)
One enabled simple	5.12 (215)
Biggest configuration complex	15.31 (643)
Combination complex	18.52 (778)

Table A.9: Effort for printer MP_305SPF

Strategy	Effort factor (#clicks)
One enabled complex	10.42 (448)
Biggest configuration complex	56.32 (2422)
Combination complex	60.98 (2622)
One enabled simple	9.16 (394)
Biggest configuration complex	48.77 (2097)
Combination complex	63.09 (2713)

Table A.10: Effort for printer SP_8400DN

https://doi.org/10.24355/dbbs.084-201812111501-0

Absolute Values for Anomalies in Re-Engineered Feature Models

Strategy	Dead Features
One enabled complex	0
Biggest configuration complex	0
Combination complex	0
One enabled simple	0
Biggest configuration complex	5 (9.4%)
Combination complex	23 (43.3%)

Table A.11: Dead features for printer MP_C501Sp

Strategy	Dead Features
One enabled complex	0
Biggest configuration complex	0
Combination complex	0
One enabled simple	1 (1.1%)
Biggest configuration complex	55 (59.1%)
Combination complex	81 (87.1%)

Table A.12: Dead features for printer MP_C4504ex_C6004ex

Strategy	Dead Features
One enabled complex	0
Biggest configuration complex	0
Combination complex	0
One enabled simple	0
Biggest configuration complex	1 (2.3%)
Combination complex	34 (79.1%)

Table A.13: Dead features for printer SP_8400DN

Strategy	Tautology	Redundant	Real Constraint
One enabled complex	1	29	64
Biggest configuration complex	156	71	80
Combination complex	78	83	106
One enabled simple	1	221	299
Biggest configuration complex	20	1128	76
Combination complex	14	1558	107

Table A.14: Overview Anomalies in constraints for MP_C501Sp

https://doi.org/10.24355/dbbs.084-201812111501-0

Strategy	Tautology	Redundant	Real Constraint
One enabled complex	5	131	53
Biggest configuration complex	168	274	195
Combination complex	160	326	152
One enabled simple	4	994	535
Biggest configuration complex	23	4318	93
Combination complex	17	5663	117

Table A.15: Overview Anomalies in constraints for MP_C4504ex_C6004ex

Strategy	Tautology	Redundant	Real Constraint
One enabled complex	0	46	43
Biggest configuration complex	1	43	46
Combination complex	0	88	46
One enabled simple	0	1403	776
Biggest configuration complex	0	1487	757
Combination complex	0	2564	781

Table A.16: Overview Anomalies in constraints for SP_C360SFNw_C361SFNw

Strategy	Tautology	Redundant	Real Constraint
One enabled complex	0	24	20
Biggest configuration complex	0	31	25
Combination complex	0	53	25
One enabled simple	0	573	191
Biggest configuration complex	0	745	317
Combination complex	0	1127	317

Table A.17: Overview Anomalies in constraints for MP_305SPF

Strategy	Tautology	Redundant	Real Constraint
One enabled complex	9	24	29
Biggest configuration complex	42	15	29
Combination complex	46	46	35
One enabled simple	7	87	39
Biggest configuration complex	44	636	120
Combination complex	35	910	37

Table A.18: Overview Anomalies in constraints for SP_8400DN

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

Bibliography

[AAHC14] E. K. Abbasi, M. Acher, P. Heymans, and A. Cleve. Reverse engineering web configurators. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 264–273, Feb 2014. (cited on Page 68)

[ABKS13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. Feature-oriented software product lines. In *Springer Berlin Heidelberg*, 2013. (cited on Page 8, 9, and 10)

[AHA+13] Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. The anatomy of a sales configurator: An empirical study of 111 cases. In Camille Salinesi, Moira C. Norrie, and Óscar Pastor, editors, *Advanced Information Systems Engineering*, pages 162–177, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. (cited on Page 6 and 7)

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009. (cited on Page 8)

[Ani17] Ms R Anitha. A literature study on selenium webdriver 2.0. *International Journal of Engineering Research in Computer Science and Engineering*, 2017. (cited on Page 28)

[BAKF04] Thorsten Blecker, Nizar Abdelkafi, Gerold Kreutler, and Gerhard Friedrich. Product Configuration Systems: State of the Art, Conceptualization and Extensions. *Génie logiciel & Intelligence artificielle. Eight Maghrebian Conf. on Software Engineering and Artificial Intelligence (MCSEAI)*, 2004. (cited on Page 1, 3, 4, and 5)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005. (cited on Page 9 and 10)

[BH00] Barry Boehm and Wilfred J Hansen. Spiral development: Experience, principles, and refinements. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000. (cited on Page 8)

https://doi.org/10.24355/dbbs.084-201812111501-0

[BKM+10] Jörg Becker, Ralf Knackstedt, Oliver Müller, Alexander Benölken, Oliver Schmitt, Mayooran Thillainathan, and André Schulke. *Online-Produktkonfiguratoren – Status quo und Entwicklungsperspektiven*, pages 85–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. (cited on Page 3 and 6)

[Bri99] Axel Brinkop. *Variantenkonstruktion durch Auswertung der Abhängigkeiten zwischen den Konstruktionsbauteilen.* PhD thesis, 1999. Sankt Augustin 1999. (Dissertationen zur Künstlichen Intelligenz. 204.) Fak. f. Informatik, Diss. v. 28.1.1999. (cited on Page 3)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005. (cited on Page 8)

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* dissertation, Technische Universität Ilmenau, 1998. (cited on Page ix, 8, and 9)

[FFJ99] A. Felfernig, G. Friedrich, and D. Jannach. Intelligente produktkonfiguratoren als voraussetzung für maßgeschneiderte massenprodukte. *e&i Elektrotechnik und Informationstechnik*, 116(3):201–207, Mar 1999. (cited on Page 6)

[FFJ01a] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2):165 – 176, 2001. (cited on Page 67)

[FFJ01b] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. UML As Domain Specific Language For The Construction Of Knowledge-Based Configuration Systems. 02 2001. (cited on Page 67)

[FZ00] Alexander Felfernig and Markus Zanker. *Diagrammatic Acquisition of Functional Knowledge for Product Configuration Systems with the Unified Modeling Language*, pages 361–375. Springer, 2000. (cited on Page 1 and 3)

[Gec] Geckodriver. https://github.com/mozilla/geckodriver. Accessed: September, 11th 2018. (cited on Page 28)

[Hen04] Patrick Henseler. *Die Konfigurations-und Verträglichkeitsmatrix als Beitrag für eine differenzierte Betrachtung von Konfigurierungsproblemen*, volume 378. ETH Zurich, 2004. (cited on Page 3)

[HHM12] Anders Haug, Lars Hvam, and Niels Henrik Mortensen. Definition and evaluation of product configurator development strategies. *Computers in Industry*, 63(5):471 – 481, 2012. (cited on Page 3 and 67)

https://doi.org/10.24355/dbbs.084-201812111501-0

[HNA+17] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. *CoRR*, abs/1710.07980, 2017. (cited on Page 8, 19, and 22)

[JK15] CR Jain and Rajesh Kaluri. Design of automation scripts execution application for selenium webdriver and test ng framework. *ARPN J Eng Appl Sci*, 10:2440–2445, 2015. (cited on Page 28)

[KAT16] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. pages 132–143. ACM, 2016. (cited on Page 10)

[KCH+90] 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, 8, and 9)

[KPK+17] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 42–45. ACM, 2017. (cited on Page 29)

[KTM+17a] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 291–302. ACM, 2017. (cited on Page 21)

[KTM+17b] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, New York, NY, USA, 2017. ACM. (cited on Page 63)

[Lü12] Daniel Lüddecke. Extraktion von Feature-Modellen aus Implementierungsartefakten, 2012. (cited on Page 8)

[LBP10] Daniel Le Berre and Anne Parrain. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. (cited on Page 29)

[LCRS13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, JAMAICA 2013, pages 53–58, New York, NY, USA, 2013. ACM. (cited on Page 28)

https://doi.org/10.24355/dbbs.084-201812111501-0

[LOG] Logging with log4j 2. https://logging.apache.org/log4j/2.x/. Accessed: October, 30th 2018. (cited on Page 29)

[MKR+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA, 2016. ACM. (cited on Page 22)

[MSS01] Tomi Männistö, Timo Soininen, and Reijo Sulonen. Product Configuration View to Software Product Families. In *In Software Configuration Management Workshop (SCM-10)*, pages 14–15, 2001. (cited on Page 1)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 28)

[NBN14] T. J. Naidu, N. A. Basri, and S. Nagenthram. Sahi vs. selenium: A comparative analysis. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 967–970, Nov 2014. (cited on Page 28)

[PIVB93] B Pine II, Bart Victor, and Andrew Boynton. Making mass customization work. 71, 01 1993. (cited on Page 3)

[PM08] Klaus Pohl and Andreas Metzger. Variabilitätsmanagement in software-produktlinien. pages 28–41, 01 2008. (cited on Page 8)

[PR02] Frank Piller and Timm Rogoll. *Marktstudie Konfigurationssysteme für Mass Customization und Variantenproduktion: Strategie, Erfolgsfaktoren und Technologie von Systemen zur Kundenintegration.* 06 2002. (cited on Page 3, 4, and 5)

[Sel] Selenium browser automation. https://www.seleniumhq.org/. Accessed: September, 11th 2018. (cited on Page 28)

[SKSL+03] Rosmary Stegmann, Michael Koch, Martin S Lacher, Thomas Leckner, and Volker Renneberg. Generating personalized recommendations in a model-based product configurator system. *Conference: Proc. Intl. Joint Conf. On Artificial Intelligence (IJCAI) – Workshop on Configuration*, 01 2003. (cited on Page 6 and 7)

[SS18] Joshua Sprey and Chico Sundermann. Computing Attribute Ranges for Partial Configurations with JavaSMT, 2018. (cited on Page 2 and 10)

[SSA17] Norbert Siegmund, Stefan Sobernig, and Sven Apel. Attributed Variability Models: Outside the Comfort Zone. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 268–278. ACM, 2017. (cited on Page 71)

https://doi.org/10.24355/dbbs.084-201812111501-0

[Sto07] Henrik Stormer. Kundenbasierte produktkonfiguration. *Informatik-Spektrum*, 30(5):322–326, Oct 2007. (cited on Page 3 and 7)

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Science, May 2009. (cited on Page 33)

[TKS18] Thomas Thüm, Sebastian Krieter, and Ina Schaefer. Product configuration in the wild: Strategies for conflicting decisions in web configurators. 2018. to appear. (cited on Page 7)

[VAHT+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A classification of product sampling for software product lines. 2018. (cited on Page 22)

[VDK01] Arie Van Deursen and Paul Klint. *Domain-specific language design requires feature descriptions*. Citeseer, 2001. (cited on Page 9)

[vRGA+15] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-Condition Simplification in Highly Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 178–188. IEEE Computer Science, 2015. (cited on Page 24)

https://doi.org/10.24355/dbbs.084-201812111501-0

https://doi.org/10.24355/dbbs.084-201812111501-0

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

Braunschweig, den 15. November 2018

https://doi.org/10.24355/dbbs.084-201812111501-0