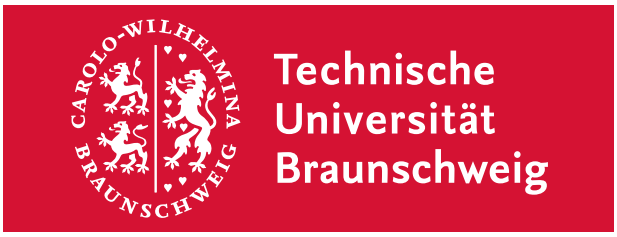


Bachelor's Thesis

Deriving Subset Software Product Lines Using Partial Configurations with FeatureIDE

Author:

Paul Westphal

27th July 2020

Advisors:

Prof. Dr. Ina Schaefer, Dr. Lukas Linsbauer

Instituite of Software Engineering and Automotive Informatics

https://doi.org/10.24355/dbbs.084-202011201423-0

Westphal, Paul:
Deriving Subset Software Product Lines Using Partial Configurations with FeatureIDE
Bachelor's Thesis, TU Braunschweig, 2020.

https://doi.org/10.24355/dbbs.084-202011201423-0

Abstract

Software product lines are commonly used in software engineering to mass-produce custom software. Over time, software product lines can become very large and complex and, thus, difficult to work with. To hide information in a software product line, or to reduce its size it may be necessary to remove features from it. To support process, we introduce subset software products lines and an approach to generate them using partial configurations. This allows developers to create a new software product line without unwanted features and focus on the task at hand. To demonstrate our approach, we implemented it for two composers in FeatureIDE, an extensible framework for feature-oriented software development. We chose one annotation-based (Antenna) and one feature-oriented (FeatureHouse) composition mechanism to show the versatility of our concept. We evaluate our implementation by deriving partial subset software product lines from known software product lines with partial configurations. The evaluation shows that our implementation delivers correct results in a runtime that is usable for further research.

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

Zusammenfassung

Softwareproduktlinien werden häufig in der Softwaretechnik verwendet um kundenspezifische Software massenproduzierbar zu machen. Mit der Zeit können Softwareproduktlinien sehr groß und komplex und damit schwierig zu benutzen werden. Um Informationen zu verstecken, oder die Größe einer Softwareproduktlinie zu verringern, kann es nötig sein, Features aus ihr zu entfernen. Um diesen Prozess zu vereinfachen, führen wir Teilmengen von Softwareproduktlinien und eine Art sie mithilfe von partiellen Konfigurationen zu generieren. Dies erlaubt es Entwicklern, eine neue Softwareproduktlinie ohne ungewollte Features zu erstellen und sich auf das Wichtige zu konzentrieren. Um unser Konzept zu beweisen haben wir es für zwei Composer in FeatureIDE, einem erweiterbaren Programmiergerüst für Feature-orientierte Softwareentwicklung, implementiert. Wir haben uns für einen annotations-basierten (Antenna) und einen feature-orientierten (FeatureHouse) composer entschieden um die Vielseitigkeit dieses Konzepts zu demonstrieren. Wir werten unsere Implementierung aus indem wir mithilfe von partiellen Konfigurationen Teilmengen von bekannten Softwareproduktlinien ableiten. Unsere Auswertung zeigt, dass unsere Umsetzung korrekte Ergebnisse liefert, in einer Laufzeit die für weitere Forschung nutzbar ist.

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

Contents

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

1. Introduction

Reusing Software parts is crucial to large scale development of custom software. Creating, as well as managing software families is a difficult task. Software product line engineering has emerged as an effective way to leverage the commonalities of different software products, while managing the variabilities [BSRC10]. Configurations offer the developer choices as to which features from a software product line should be included in a product. For each selectable feature, conventional configurations give developers a binary decision, whether to inlucde or exclude it.

During the configuration process, developers may not want to commit to selecting some features immediately. However, for other features they may know in advance whether they should be part of the eventual product or not. Partial configurations allow developers to defer this choice to a later time by not only allowing the selection or deselection of a feature, but to leave it undecided [BSRC10].

To further support the testing and development process with partial configurations, it is possible to derive a new software product line from these partial configurations, which is a subset of the original software product line. This process removes variability from selected features and entirely removes unselected features and their corresponding code. This simplifies the development process by reducing the size of the software product line.

It allows developers to focus on the task at hand, without having to worry about compatibility with unneeded features. To accomplish this, we have developed a prototype in FeatureIDE, a framework for feautre-oriented software development. We have then applied it to well known datasets to verify its correctness and examine its performance.

Goal of this Thesis

The goal of this thesis is to explore the possibilities of partial configurations for the derivation of subset software product lines and provide a usable framework for further research. To accomplish this, we develop a prototype that allows the derivation of subset software product lines and show its correctness.

https://doi.org/10.24355/dbbs.084-202011201423-0

Structure of this Thesis

Our thesis is structured in the following way: In Chapter 2 (Motivating Example), we present an example scenario that illustrates a use case for the derivation of subset software product lines. We introduce the concepts and terminology that will be used in this thesis. Next, in Chapter 3 (Background) we explain these concepts more in-depth and introduce the definitions that we will be working with in the following chapters. In Chapter 4 (Concept), we define subset software product lines and the expected output of the derivation of subset software product lines using partial configurations. We also introduce algorithms that we use for this purpose. In Chapter 5 (Implementation), we present the FeatureIDE framework and explain in detail our implementation of the concepts of Chapter 4 into FeatureIDE. In Chapter 6 (Evaluation) we put our implementation to the test and evaluate its correctness and performance. In Chapter 7 (Related Work) we present other publications with similar goals and compare them to our work. Last, in Chapter 8, we summarize our contributions and findings and propose ideas for possible future research.

https://doi.org/10.24355/dbbs.084-202011201423-0

2. Motivating Example

In this chapter, we present a motivating example to illustrate a use case with a problem we are attempting to solve.

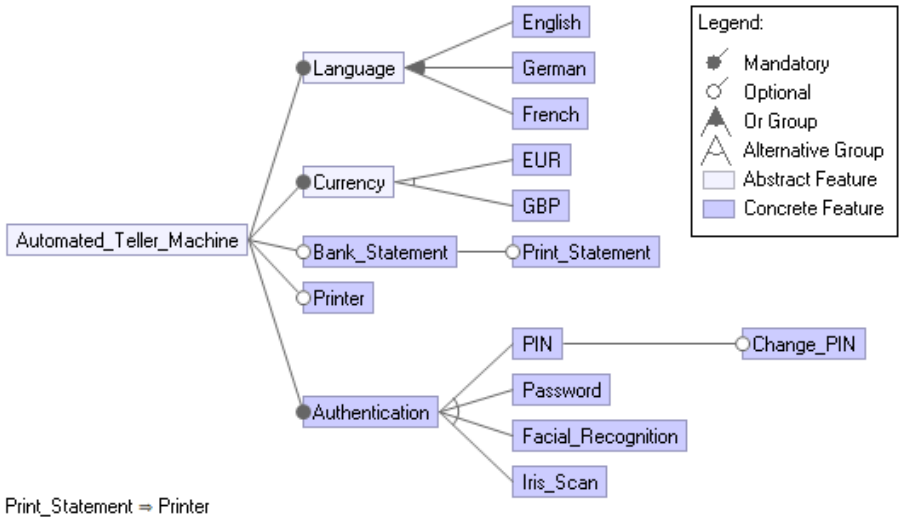


Figure 2.1: Feature model representing an ATM software family

Figure 2.1 shows a feature model representing a family of automated teller machines (ATM). An ATM requires a currency, an Authentication method, and at least one language. Additionally, developers may choose to include any of the optional features (e.g. `Printer`). Features in the feature model can either be abstract or concrete. Abstract features represent a concept and do not have any code associated with them, while each of the concrete features represents a software part. In our fictitious scenario, a firm A producing ATMs wants to delegate the rest of the development of an ATM software project to a subcontractor B. However, the software product line contains source code involving new technologies that are security-critical. `Iris_Scan` and `Facial_Recognition` are features that are not needed for subcontractor B to finish their project. Sharing these software parts with another software firm could

https://doi.org/10.24355/dbbs.084-202011201423-0

potentially increase the risk of a leak of these security-related software parts; firm A wants to avoid this.

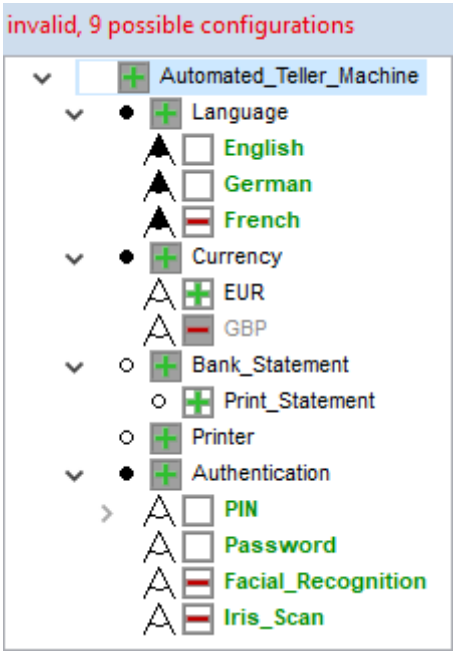


Figure 2.2: Partial configuration early in the development process of ATM

The software developers are early in the development process. As can be seen in Figure 2.2, it has been decided that the machine will use Euro as a currency and have the ability to print bank statements. The language French has been ruled out, and as such its corresponding feature was deselected. It has also been decided that no information about the facial recognition or iris scan authentication methods will be given to the developers of firm B finishing the project. As a result of these decisions, the features `French`, `Facial_Recognition` and `Iris_Scan` have been deselected. Additionally, resulting from the alternative relationship between the currency features, selecting the `EUR` feature automatically invalidates the `GBP` feature. Figure 2.3

```
1  // #if Facial_Recognition
2  package atm.authentication;
3  public class FacialRecognition extends BiometricAuthentication {
4      // secret code currently in development
5      public boolean authenticate() {
6          System.out.println((String) bundle.getObject("authenticate_facial_recognition"));
7          return true;
8      }
9  }
10 // #endif
```

Figure 2.3: Code excerpt from the ATM Software Product Line

shows java code from the software product line, which is associated with the feature `Facial_Recognition`.

The goal is to derive a new software product line, which does not include these four features and their corresponding code. Firm A can share this generated part of the

https://doi.org/10.24355/dbbs.084-202011201423-0

software product line with firm B, without having to worry about their security-critical code being exposed to third parties.

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

3. Background

This chapter provides the reader with the required knowledge and introduces the terminology used in the following chapters. This includes software product lines, feature models, configurations, assets, the generation of products. First, we roughly describe each concept, then we define it in an abstract way. Last, we illustrate it with an example.

3.1 Software Product Lines

A software product line (SPL) is a set of related software products. These products share features. A product can be derived from a valid selection of features (see Section 3.5). SPLs are used to efficiently reuse code, rather than having to create libraries from old code or even starting over from scratch with each software project [PBL05].

Definition 3.1 (Software Product Line)**.** *A software product line $S = (M, A)$ consists of a Feature Model $M = (F, P)$ (see Section 3.2) and corresponding assets A (see Section 3.4).*

3.2 Feature Models

Feature models are widely used to define the set of valid feature selections for a software product line [BSRC10][HFACA13]. A feature model consists of features and the relationships between those features. Figure 2.1 depicts an example feature model showing a representation of an ATM software product line.

Definition 3.2 (Feature Model)**.** *A feature model $M = (F, P)$ consists of a finite set of features F and a propositional formula P in CNF, which is a set of clauses. Each clause $p \in P$ is a disjunction of literals. P defines the relationship between features. The set of features F is the set of all variables in P.*

https://doi.org/10.24355/dbbs.084-202011201423-0

Each feature can be either abstract or concrete. Abstract features do not have assets mapped to them. For example, the abstract feature `Language` represents the concept of language and allows the selection of a concrete feature, but does not have any code directly associated with it.

There are two types of relationships between features: The first type is a relationship between a parent feature and its children; in FeatureIDE (see Section 5.1) these can be *and/or/alternative* groups.

- An *or* relationship means that if the parent feature is selected, at least one child feature must be selected. The ATM software product line represented by Figure 2.1 supports one or more of the languages `English`, `German`, and `French`.
- An *and* relationship means that selecting the parent feature allows the selection of any number of its child features. Any of the features `Language, Currency, Bank_Statement, Printer,` and `Authentication` can be selected.
- An *alternative* relationship means that if the parent feature is selected, exactly one of the child features must be selected. In Figure 2.1 upon selected `Authentication`, there is a choice between `PIN, Password, Facial_Recognition,` and `Iris_Scan` as an authentication method, but they are mutually exclusive.

For the example feature model shown in Figure 2.1, the set F is

$\{$*Automated_Teller_Machine, Language, English, German, French, Currency, EUR, GBP, Bank_Statement, Print_Statement, Printer, Authentication, PIN, Change_PIN, Password, Facial_Recognition, Iris_Scan*$\}$.

Features can be marked as *mandatory*. A mandatory feature has to be selected, if its parent feature is selected. There are constraints that cannot be expressed with a feature model's hierarchy. These are cross-tree relationships, which are modeled via constraints. For example, in the ATM feature model in Figure 2.1, by selecting the feature `Print_Statement` you have to select the feature `Printer` as well, despite these features not being directly related via the feature model's hierarchy.

All of these relationships are modeled by the feature model's propositional formula P. The propositional formula P for the example feature model Figure 2.1 is:

$\{(Automated\_Teller\_Machine), (Bank\_Statement \vee \neg Print\_Statement), \dots\}$ [1]

Each cross-tree constraint or added feature adds more clauses to the feature model's propositional formula. For example, the *or* relationship between the mandatory feature `Language` and its children is realized by adding the following clauses to the feature model's propositional formula P:

$(Language \vee \neg Automated\_Teller\_Machine), (Language \vee \neg English), (Language \vee \neg German), (Language \vee \neg French), (English \vee German \vee French \vee \neg Language)$

[1] The full propositional formula can be found in the Appendix Chapter A.

https://doi.org/10.24355/dbbs.084-202011201423-0

3.3 Configurations

A configuration is a selection of features. Each feature can be selected, deselected, or left undecided. A full configuration is a configuration where no feature is left undecided. A configuration where at least one feature is left undecided is a partial configuration.

Definition 3.3 (Configuration)**.** *Given a set of Features F, a configuration C is a set of tuples $c = (f, b)$ with $b \in \{true, false, ?\}$ that assign one truth value b to each feature $f \in F$. The set of possible truth values {true,false,?} represent selection ($true$), deselection ($false$) or undecided ($?$).*

Definition 3.4 (Full Configuration)**.** *A configuration C is a full configuration,*

iff $\nexists x \in C : x.b = ?$.

Definition 3.5 (Partial Configuration)**.** *A configuration C is a partial configuration,*

iff $\exists x \in C : x.b = ?$.

Figure 2.2 shows a configuration. The features *English, German, French, PIN,* and *Password* are still undecided.

A configuration can be valid or invalid for a feature model's propositional formula P. If the assignments of truth vales to the features evaluate to true, it is a valid configuration. It is invalid if they evaluate to false.

Definition 3.6 (Evaluation of Propositional Formula with a Configuration)**.** *Given a propositional formula P and a set of tuples C with $c = (f, b) \in C$, we define the operation*

$P(C) = r \in \{true, false\}$

as an evaluation of the propositional formula P, using the truth value assignments of the tuples $c = (f, b) \in C$ as variables in the clauses in C. If a variable $f \in V_p$ does not have a truth value assignment in C or is assigned the truth value $?$, it is treated as $false$ for the purpose of evaluation the propositional formula.

Definition 3.7 (Valid Configuration)**.** *A configuration C is valid for a Feature Model $m = (F, P)$ if and only if $P(C)$ evaluates to true.*

Definition 3.8 (Invalid Configuration)**.** *A configuration C is invalid for a Feature Model $m = (F, P)$ if and only if $P(C)$ evaluates to false.*

Definition 3.9 (Core Features)**.** *Given a set of all valid configurations C_{valid} and a set of features F, a feature f is a core feature if*

$\forall c \in C_{valid} : f_c = true$.

In practice, undecided features are often implicitly treated as unselected. Partial configurations allow a developer to defer the choice which features to include in the final product to a later point in time. A partial configuration can not be used to create a software product unless undefined features are treated as unselected. In this thesis, we develop a way to use partial configurations to derive a subset of a software product line.

https://doi.org/10.24355/dbbs.084-202011201423-0

3.4 Assets

The assets of an SPL consist of a payload and its mapping to the SPL's features. The payload is usually source code, but it can be anything on a computer's file system. For example, FeatureIDE supports the composer `ImageComposer` that supports the generation of `.png` files by superimposing them[2].

Definition 3.10 (Assets)**.** *Given a Feature Model $M = (F, P)$, we define assets A as a set of tuples $a = (s, U)$ where s is a unique payload and its location in the context of the SPL and U is a propositional formula in* ***CNF*** *using the features F as variables. If the propositional formula U is empty, it is treated as a tautology. S_{all} is the set of all payloads in A.*

Definition 3.11 (Reachable Asset)**.** *Given an SPL $S = (M, A)$, an asset $a \in A$ with a propositional formula U is reachable if $\exists C : (U(C) = true) \wedge (M.P(C) = true)$.*

Definition 3.12 (Reachable with partial configuration)**.** *Let $C_{undecided}$ be the set of tuples $c = (f, b)$ with $b = ?$. Given an SPL $S = (M, A)$, an asset $b \in A$ with a propositional formula U is reachable with a partial configuration C if there is a set $C_{decision}$ of tuples $a = (f, b)$, such that*

- $U((C \setminus C_{undecided}) \cup C_{decisions}) = true$
- $M.P((C \setminus C_{undecided}) \cup C_{decisions}) = true$

Figure 2.3 shows some of the assets of the ATM software product line. There is a direct mapping between the lines 2 to 9 of this source code and the feature `Facial_Recognition` through the preprocessor statement `// #if Facial_Recognition`. The propositional formula U of this asset is $\{(Facial\_Recognition)\}$. The payload s is the Java code in lines 2 to 9.

The asset is reachable because the propositional formula $P \wedge U$ is solvable. However, it is not reachable with the configuration pictured in Figure 2.2, because the feature `Facial_Recognition` is deselected. This means the propositional formula U evaluates to `false`, making the asset unreachable.

3.5 Generating Products

Given a software product line and a configuration, a product can be generated. We define a product as the set of all payloads of the assets in an SPL that evaluate to true, with a given configuration.

Definition 3.13 (Generating a product)**.** *Given a software product line $S = (M, A)$ with $M = (F, P)$, and a configuration C, the function $generate(S, C)$ returns the following set of payloads:*

$$generate(S,\ C) = \begin{cases} \emptyset & P(C) = false \\ \{a.s | (a.U(C) = true) \wedge (a \in A)\} & sonst \end{cases}$$

If the configuration is invalid, an empty set is returned. Otherwise, the payloads of all assets with propositional formulas that evaluate to true for the given configuration are included in the generated product.

[2] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.core.images

https://doi.org/10.24355/dbbs.084-202011201423-0

4. Concept

In this chapter, we describe our concept for the derivation of subset SPLs. First, we define the problem we are attempting to solve. Second, we define a solution to the problem and provide a step-wise overview of how we get to the solution. Then we go further into detail for each step and introduce the algorithms that we use to solve the problem.

4.1 The Problem Statement

Over time, software product lines can become very large and complex or contain unwanted features. For certain stakeholders, a smaller version of a software product line can be enough, but manually reducing the size of both the SPL's feature model and corresponding assets is a very time-consuming and error-prone process.

4.2 Our Approach to Solving the Problem

Our goal is to derive a subset SPL S_1 from an SPL S_0 using a configuration C. In this section, we first define subset SPLs. Second, we define the expected output of the operations we use to derive subset SPLs using configurations. Then, we give an overview over the three steps that are applied to an SPL to produce this output. Configurations are a widely used and understood mechanism to generate products, we found that they can also be used to simplify a software product line.

A subset SPL S_1 is a reduced version of a software product line S_0. The set of features in S_1 and the set of products that can be generated are subsets of those sets in S_0.

Definition 4.1 (Subset Software Product Line)**.** *We have a software product line $S_1 = (M_1, A_1)$ with $M_1 = (F_1, P_1)$ and $A_1 = (S_1, U_1)$ and a software product line $S_0 = (M_0, A_0)$ with $M_0 = (F_0, P_0)$ and $A_0 = (S_0, U_0)$. Let C_p be a set of all possible configurations $C = (f, b)$, i.e. permutations of features $f \in F_1$ and truth values $b \in \{true, false\}$.*

Then, S_1 is a subset of product line S_0 ($S_1 \subset S_0$) if and only if:

https://doi.org/10.24355/dbbs.084-202011201423-0

1. $F_1 \subset F_0$
2. $\forall C \in C_p : generate(S_0, C) = generate(S_1, C)$

4.1 implies that for an SPL S_1 to be a subset of another SPL S_0, its set of features F_1 has to be a subset of F_0 and each possible configuration for S_1 has to generate the same product for both SPLs.

However, the goal is not to create any subset of a software product line. As described in the motivating example in Chapter 2, we want to use a configuration C to derive a subset sofware product line S_1 from a software product line S_0. We now define the expected output of this process.

Definition 4.2 (Output: Subset Software Product Line derived with a Configuration)**.** *Given a software product line $S_1 = (M_1, A_1)$ with $M_1 = (F_1, P_1)$ and $A_1 = (S_1, U_1)$, a software product line $S_0 = (M_0, A_0)$ with $M_0 = (F_0, P_0)$ and $A_0 = (S_0, U_0)$, and a configuration C_{base} with C_{base} being a set of tuples $a = (f, b)$.*

We define $F_{undecided}$ as a set of features that are not assigned $true$ or $false$ in C_{base}:

$F_{undecided} = \{\forall f \in F : f | (f, true) \notin C \wedge (f, false) \notin C\}$

We define C_{base_p} as the set of all possible configurations of undecided features in C_{base}, i.e. permutations of features $f \in F_{undecided}$ and truth values $b \in \{true, false\}$.

Then S_1 is the subset of S_0 based on the configuration C_{base} if and only if:

1. $S_1 \subset S_0$
2. $\forall C \in C_{base_p} : generate(S_0, c) = generate(S_1, C)$
3. $\forall p \in P_1 : \forall f \in F_{desel} : f \notin p$
4. $\forall a \in A_1 : \forall u \in a.U : \forall f \in F_{desel} : f \notin u$

4.2 implies that the expected output of our operation is an SPL_1 where all configurations that are still reachable with the selections in C_{base} generate the same products from S_1 and S_0. Also S_1 no longer contains any features that were deselected in C_{base}.

We now describe our procedure to deriving subset SPLs based on a configuration and generating this output. It can be summed up in the following three steps applied to a copy of a software product line S_0 in the following order:

1. Remove features specified as unselected in C_{base} from feature model.
2. Turn features specified as selected in C_{base} into core features.
3. Modify propositional formulas of assets.

We explain each of these steps in detail in the following sections.

https://doi.org/10.24355/dbbs.084-202011201423-0

Removing features specified as unselected from feature model

In this section, we explain the first step of our approach to solving the problem outlined in Section 4.2. We define the set of features F_1 and present an algorithm we use to modify the propositional formula of the feature model M_0.

The input for this step is an SPL $S_0 = (M_0, A_0)$ and a configuration C, which is a set of tuples $a = (f, b)$. The goal of this step is to create a feature model $M_1 = (F_1, P_{modified})$ with all deselected features removed. The assets of our SPL will be modified at a later time. We modify the propositional formula P_0, such that all deselected features are replaced with $false$ and thus removed. We define F_{desel} as the set of features that are assigned the value $false$ in the configuration C:

$$\forall x \in C : (x.b = false) \implies f_x \in F_{desel}$$

Then, we modify the propositional formula using the Algorithm 1, with P_0 and F_{desel} as inputs:

Algorithm 1: Remove deselected features from propositional formula in `CNF`

```
input : propositional formula P in CNF
        set of features F, each f ∈ F corresponding to a variable in P
output: propositional formula P_modified, with all features f ∈ F removed.
foreach clause c ∈ P do
    foreach feature f ∈ F do
        if c contains f then
            replace all occurrences of f in c with false
    if c is tautology then
        remove clause c from P
        continue
    else if c is contradiction then
        replace clause c with (false)
        continue
    else
        remove all occurrences of false from c
        continue
foreach clause c ∈ P do
    if P contains duplicate of c then
        remove clause c from P
return P
```

This algorithm iterates over every clause in the propositional formula and replaces all occurrences of features in F with false. Afterwards, it iterates over all clauses in P a second time to remove any duplicate clauses (i. e., clauses with an equivalent propositional formula).

Assume we have a propositional formula $P = \{(Authentication \vee \neg Iris\_Scan), (PIN \vee Password \vee Facial\_Recognition \vee Iris\_Scan \vee \neg Authentication)\}$ and a set of features $F = \{Facial\_Recognition, Iris\_Scan\}$ as inputs for Algorithm 1.

- The first clause is removed, because $(Authentication \vee \neg False)$ is a tautology.

https://doi.org/10.24355/dbbs.084-202011201423-0

- The second clause is changed to $(PIN \vee Password \vee False \vee False \vee \neg Authentication)$ and all occurences of $False$ are then removed.
- The second iteration over P does not change the propositional formula, since there are no duplicates.

The algorithm then returns the propositional formula $P = \{(PIN \vee Password \vee \neg Authentication)\}$.

The return value of this algorithm with P_0 and F_{desel} as inputs is a propositional formula $P_{modified}$ in `CNF`. The set of features F_1 is defined as the set of features variables in $P_{modified}$. After removing all features in F_{desel} from the propositional formula, the SPLs set of features F is now F_1. With this step completed, the feature model $M = (F_1, P_{modified})$ now fulfills the following condition $F_1 \subset F_0$ of Definition 4.1.

Turning Selected Features into Core Features

Next, we explain the second step of our approach and the reasoning behind it. This section covers how to turn features in a feature model, which are `selected` in a configuration C, into core features. We define the set of selected features F_{sel} and introduce an algorithm to turn features into core features.

As described in Section 3.9, a core feature is a feature that has to be included in every valid configuration. Features that have been selected in the configuration that is used to derive a subset SPL will definitely be included in the final product. If they could still be deselected, there would be a difference between the products that can be generated from the partial configuration and generated SPL, which means it can not be a subset SPL according to 4.2. Additionally, not all composition mechanisms (e.g., FeatureHouse [AKL09]) can include assets if they do not have an associated feature. Therefore, we preserve these features and turn them into core features, rather than removing them. This solution works for every composition mechanism we are aware of. Furthermore, we avoid having to merge the payloads of selected features into unrelated features to preserve their assets.

For this step, we further modify our feature model $M = (F_1, P_{modified})$ from the last step. We define F_{sel} as the set of features that are assigned the value $true$ in the configuration C: $F_{sel} = \{f | (f, true) \in C\}$. We then use Algorithm 2 with $P_{modified}$ and F_{sel} as inputs.

Algorithm 2: Turn selected features into core features

input : propositional formula P in `CNF`
set of features F, each $f \in F$ corresponding to a variable in P
output: propositional formula $P_{modified}$, with all features $f \in F$ turned into core features
foreach *feature* $f \in F$ **do**
  add a clause (f) to P
return P

This algorithm iterates over all features in F and adds a clause mandating that feature to the propositional formula.

https://doi.org/10.24355/dbbs.084-202011201423-0

Suppose we have a propositional formula

P = {(PIN ∨ Password ∨ ¬ Authentication)}

and a set of features F = {Automated_Teller_Machine, Language, Currency, EUR, Bank_Statement, Print_Statement, Printer, Authentication}, which is the set of selected features in Figure 2.2, as inputs.

Algorithm 2 returns a propositional formula P = {(PIN ∨ Password ∨ ¬ Authentication), (Automated_Teller_Machine), (Language), (Currency), (EUR), (Bank_Statement), (Print_Statement), (Printer), (Authentication)}.

The return value with $P_{modified}$ and F_{sel} as inputs is a finished propositional formula P_1. With this step completed, only the set of assets has to be modified for all conditions in Definitions 4.1 and 4.2 to hold.

Modifying Propositional Formula of Assets

In this section, we explain the last step of our approach. We explain how the assets A of an SPL have to be modified to derive a subset SPL S_1 with a configuration C and present Algorithm 3 for this purpose.

Assets consist of a payload and a propositional formula in `CNF`. If the propositional formula is empty or evaluates to `true`, the payload is included in the product. We exclude assets from the subset SPL that are not reachable after their associated features have been removed. Additionally, propositional formulas of assets that are still reachable are modified, such that features that are no longer in the set of features $F_1 \in M_1$ are not being used as variables.

To accomplish this, we use Algorithm 3 with the asset of assets A_0 from the original SPL S_0 and the set of deselected features F_{desel} from configuration C.

Algorithm 3: Modify propositional formula of assets

input : set of assets A_0 with $a = (s, U_0) \in A_0$
set features F, each $f \in F$ corresponding to a variable in $a.U$
output: set of assets A_1 with adjusted propositional formulas $a.U_{modified}$,
with all features $f \in F$ removed.
and with unreachable assets removed

```
foreach Asset a ∈ A do
    a.U := Algorithm1(a.U, F)
    if a.U is contradiction then
        A := A \ a
return A
```

This algorithm iterates over every asset $a \in A$ and calls Algorithm 1 to modify the asset's propositional formula. Afterwards, the algorithm evaluates the propositional formula. If it is a contradiction, the asset a is removed from the set of assets A. After the algorithm is done with this process for every asset, it returns the remaining set of assets $A_{modified}$.

Suppose we have a set of assets A = {(*source code implementation of facial recognition*, (Facial_Recognition)), (*source code to initialize authentication*,(Authentication),

https://doi.org/10.24355/dbbs.084-202011201423-0

(Iris_Scan ∨ Password ∨ Facial_Recognition ∨ PIN))} and a set of features F = {Facial_Recognition, Iris_Scan} as inputs for Algorithm 3 as inputs for Algorithm 3.

- The first asset is removed from A, because the propositional formula ($false$) is a contradiction.
- The propositional formula of the second asset is changed: (*source code to initialize authentication*,(Authentication), (Password ∨ PIN))

Algorithm 3 returns the set A = {(*source code to initialize authentication*,(Authentication), (Password ∨ PIN))} with the implementation of facial recognition removed.

The return value of this algorithm with A_0 and F_{desel} as inputs is the set A_1. The assets of the SPL have been modified such that assets that are unreachable with C_{base} are removed and the propositional formulas of all assets $a \in A_0$ have been modified to exclude deselected features F_{desel}.

With this step completed, the assets A_1 now fulfill the following necessary conditions according to Definition 4.1

- $F_1 \subset F_0$
- $\forall C \in C_p : generate(S_0, C) = generate(S_1, C)$
- $\forall C \in C_{base_p} : generate(S_0, C) = generate(S_1, C)$
- $\forall p \in P_1 : \forall f \in F_{desel} : f \notin p$
- $\forall a \in A_1 : \forall u \in a.U : \forall f \in F_{desel} : f \notin u$

Therefore, all conditions are fulfilled and S_1 is a subset SPL of S_0 based on C_{base}.

In this chapter, we described a series of steps that allow the derivation of a subset SPL from an existing SPL with a configuration. In the next chapter, we will apply this concept by implementing it in FeatureIDE. Then, we evaluate the correctness and runtime of that implementation.

https://doi.org/10.24355/dbbs.084-202011201423-0

5. Implementation

In this chapter, we explain how we implemented the concepts from the last chapter in FeatureIDE. First, we present the tool FeatureIDE and explain why we chose it to implement derivation of subset SPLs. Next, we describe in detail how we implemented the concepts from the previous chapter into FeatureIDE. We implemented the above concepts for two concrete variability mechanisms: The annotative composer Antenna and the feature-oriented composer FeatureHouse. We later explain the syntax and differences in feature-code mapping of Antenna and FeatureHouse. All code described in this chapter is available on github[1].

5.1 FeatureIDE

FeatureIDE is a framework for feature-oriented software development [MTS+17a]. It has been under constant development as an open-source project since 2004. FeatureIDE can be used as a standalone library or as a plugin for the popular open-source IDE Eclipse. FeatureIDE offers developers a wide variety of tools for the development of software product lines, such as a graphical feature diagram editor, automated analysis of feature models, a large selection of composers (which we explain in further detail in Section 5.5), a configuration editor that supports partial configurations, and support for the generation and testing of products.

We chose FeatureIDE for this project, because it already offers a lot of functionalities (e.g. partial configurations editor, support for propositional formulas) that are useful for the derivation of subset product lines. Additionally, the large number of existing tools in FeatureIDE allow us to further use and test newly generated subset SPLs in a variety of ways. FeatureIDE is written in Java, so our implementation is written in Java as well. FeatureIDE's different functionalities are spread out over multiple plugins such that users can pick and choose which parts of FeatureIDE they want to install in their Eclipse application. Our implementation of subset software product lines is built into existing FeatureIDE plugins; we extended the plugin `FeatureIDE`, as well as the composer plugins `Antenna`, and `FeatureHouse`. To use the functionality presented in this chapter, one only needs to install these plugins.

[1] https://github.com/PaulWestphal/FeatureIDE-subsetSPL

https://doi.org/10.24355/dbbs.084-202011201423-0

5.2 Deriving Subset SPLs in FeatureIDE

As shown in Figure 5.1, the derivation of subset SPLs is split over three distinct classes, each solving distinct tasks. First, `NewPartialFeatureProjectWizard` allows the user to input the configuration, project name, and target path through a graphical user interface (GUI). The class uses this information to set up the new project. Second, the project is passed to the `PartialFeatureProjectBuilder`. This class removes deselected features from the project's feature model and creates a constraint, mandating the selection of selected features. Last, the project is passed to the feature project's composer. Here, the assets will be manipulated such that unreachable code is removed. Due to the differences in mapping betwen code and features between the different composers, this step has to be implemented for each composer individually. For our prototype, we decided to extend one annotation-based composer (Antenna Section 5.5.1) and one composer from the feature-oriented programming paradigm (FeatureHouse Section 5.5.2).

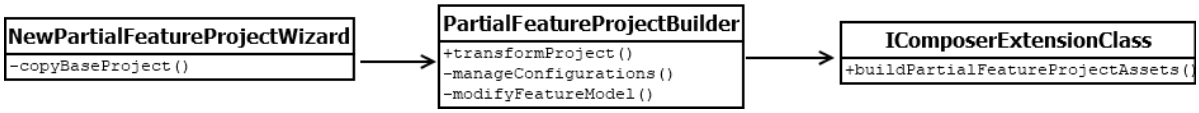


Figure 5.1: Simplified Class Diagrams Showing the Tasks of each Class involved in the Building of Subset SPL

In the following sections, we describe in more detail how these classes and their methods work on a technical level.

5.3 Composer-independent steps

In this section, we explain the implementation of steps one and two of the derivation of subset SPLs that were outlined in the concept chapter Section 4.2. Namely, modifying the feature model's propositional formula and turning selected features into core features. These steps are independent of the composer being used and do not need to be changed if one wants to implement this process for any additional composers.

NewPartialFeatureProjectWizard

In this section, we explain how the `NewPartialFeatureProjectWizard` is used and how it is implemented in FeatureIDE. This class provides users a graphical interface to decide the inputs to use for the steps outlined in Section 4.2 in the concept chapter.

The `NewPartialFeatureProjectWizard` offers users a simple graphical interface to derive subset SPLs from existing FeatureIDE projects. To access this wizard, one only needs to right-click any FeatureIDE project which supports the derivation of partial feature projects (Section 5.5), navigate to the `FeatureIDE` submenu, and click the menu entry `Derive Partial Project`. The wizard is instantiated and passed the selected FeatureIDE project by our handler class `NewPartialProjectWizard-Handler`. This class extends the FeatureIDE class `AFeatureProjectHandler`, which is an abstract class for operations in FeatureIDE that work on a FeatureIDE project.

https://doi.org/10.24355/dbbs.084-202011201423-0

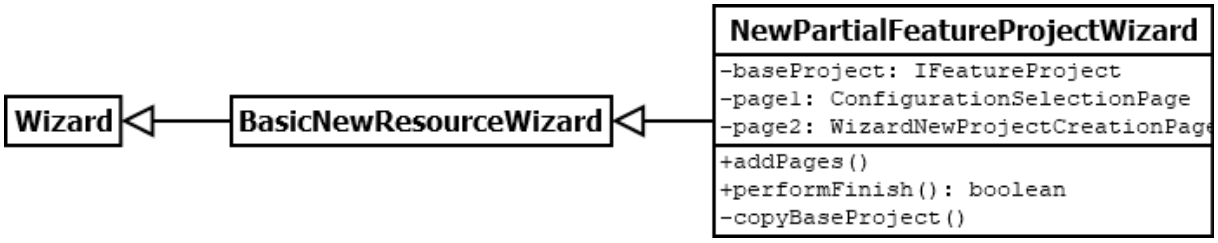


Figure 5.2: Class Diagram for New Partial Feature Project Wizard

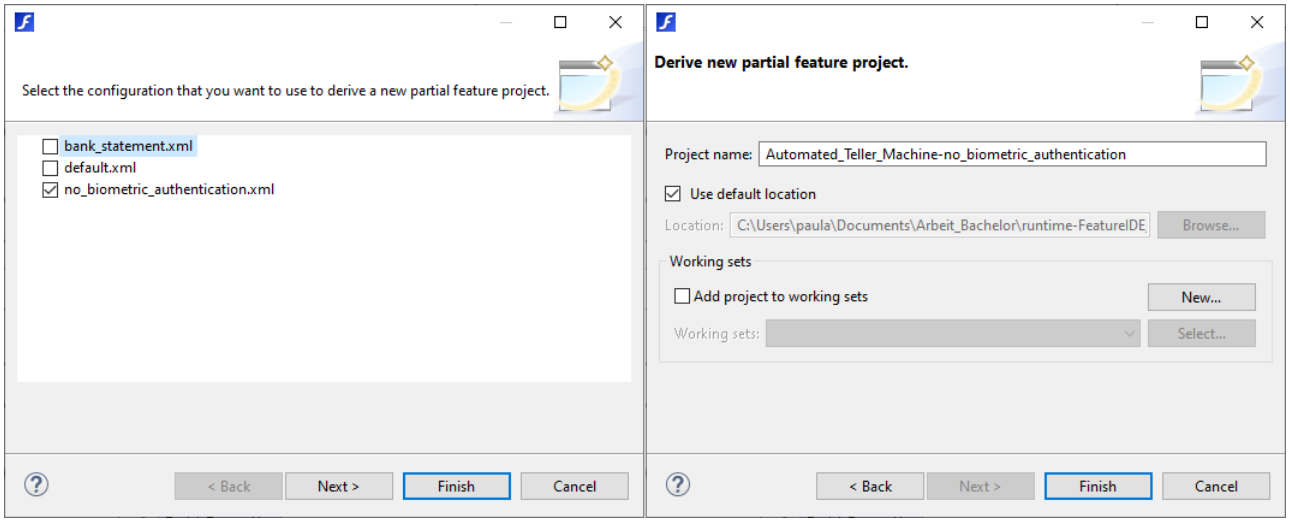


Figure 5.3: Screenshots of the NewPartialFeatureProjectWizard pages

The class `NewPartialFeatureProjectWizard` is located in the package `de.ovgu.featureide.ui.wizards`. As shown in Figure 5.2, our class `NewPartialFeatureProjectWizard` is an extension of Eclipse's `BasicNewResourceWizard`. That means it consists of Eclipse `WizardPage` objects.

The wizard has two pages, which can be seen in Figure 5.3. The first page `ConfigurationSelectionPage` is a simple extension of the Eclipse `WizardPage` and offers the user a simple menu where they can choose one configuration. We wrote this page to not be specific to the derivation of partial feature projects, such that it could be used for other tasks in the future that require the selection of a configuration in FeatureIDE. The page is constructed by calling the constructor `ConfigurationSelectionPage()` and passing a list of configuration names and the name of the configuration that should be selected by default.

The second page is a `WizardNewProjectCreationPage`, which is a wizard page that is commonly used in Eclipse to set up new projects. This page allows users to input a name and location for the new partial FeatureIDE project and allows adding it to the working sets.

Before beginning the steps described in Chapter 4, we first make a copy of the base project to then work on. This task is also handled by the `NewPartialFeatureProjectWizard`. We implemented copying of the base project by using Eclipse's `CopyProjectOperation`, which is created and executed after pressing finish. After

https://doi.org/10.24355/dbbs.084-202011201423-0

this step, the class `PartialFeatureProjectBuilder` is instantiated by passing the newly created copy and the path to the selected configuration.

PartialFeatureProjectBuilder

In this section, we explain how the `PartialFeatureProjectBuilder` class takes a configuration and a FeatureIDE project as inputs to fulfill steps one and two of our approach to solve the problem outlined in the concept chapter Section 4.2.

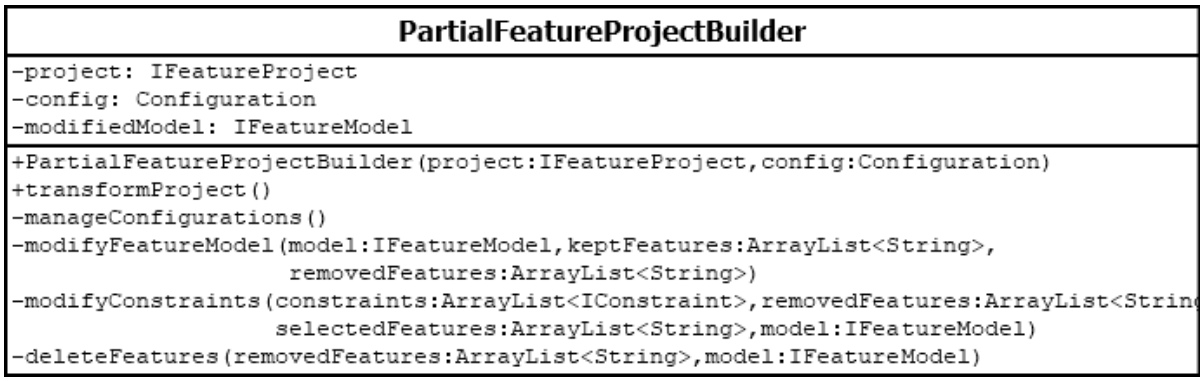


Figure 5.4: Class Diagram for Partial Feature Project Builder

The class `PartialFeatureProjectBuilder` has been developed such that it can be used independently from the wizard we introduced in the last section. Only a FeatureIDE project and a configuration are required to instantiate and run the `PartialFeatureProjectBuilder`. Upon calling the method `transformProject()`, the builder class begins working on the project.

First, the builder calls the method `manageConfigurations()`. This method deletes all configuration files in the new FeatureIDE project because they will be unusable after the subset SPL derivation process. The method then creates a copy of the configuration that was used to instantiate the `PartialFeatureProjectBuilder` class in the new FeatureIDE project's configuration directory.

Then, the two lists `removedFeatureList` and `selectedFeatureList` are created from the configuration file. These lists will be used in the `modifyFeatureModel()` and `buildPartialFeatureProjectAssets()` methods.

Previously, only the method `getSelectedFeatureNames()` was available in FeatureIDE's `Configuration` class. We implemented the methods `getUnselectedFeatureNames()` and `getUndefinedFeatureNames()` in `Configuration` to return the names of deselected and undecided features. Analogous to `getSelectedFeatures()`, these methods create a list of strings with features names. This accomplished by iterating over the features in the configuration and adding the names of each feature with the corresponding selection type to the list.

These lists, along with the FeatureIDE project's feature model, are then passed to the method `modifyFeatureModel()`. This method is used for fulfilling steps one and two in Section 4.2. First, the feature model's cross-tree constraints are modified. All constraints with unselected features have their unselected features replaced with

https://doi.org/10.24355/dbbs.084-202011201423-0

`false` and are then simplified. The exact way that propositional formulas are processed in FeatureIDE will be explained in the next section. `modifyconstraints()` also adds a constraint mandating the selection of each feature that was selected in the configuration the derivation is based on. For the configuration shown in Figure 2.2 for example, the constraint Automated_Teller_Machine ∧ Language ∧ Currency ∧ EUR ∧ Bank_Statement ∧ Print_Statement ∧ Printer ∧ Authentication is added to the feature model. Users may simply delete this constraint if they don't want these features to always be selected. Then, features that are deselected in the confguration are deleted. If these features are in a `or` or `alternative` relationship and only have one remaining sibling, that sibiling is set to `mandatory`. This is done to preserve the logical constraints of that relationship by forcing the selection of at least one child feature if the parent was selected. Last, the source folder of the FeatureIDE project and a list of deselected features is passed to the `builPartialFeatureProjectAssets()` method of the project's composer. The way assets are mapped to features in the feature model is different for each composer, therefore it needs to be implemented seperately for each composer.

5.4 Modifying Propositional Formulas in FeatureIDE

For the sake of simplicity, we defined all propositional formulas in the concept chapter as `CNF`. In FeatureIDE however, formulas for feature models can be any propositional formula that can be expressed through `prop4j` nodes. `prop4j` is a Java library for propositional formulas. Its source code is publicly available [2]. Propositional formulas in `prop4j` are made up of nodes, which may consist of just a feature (`Literal`), or a combination of features and logical operators (`Not`, `And`, `Or`, `Equals`). `prop4j` includes a SAT solver, which we use to evaluate propositional formulas.

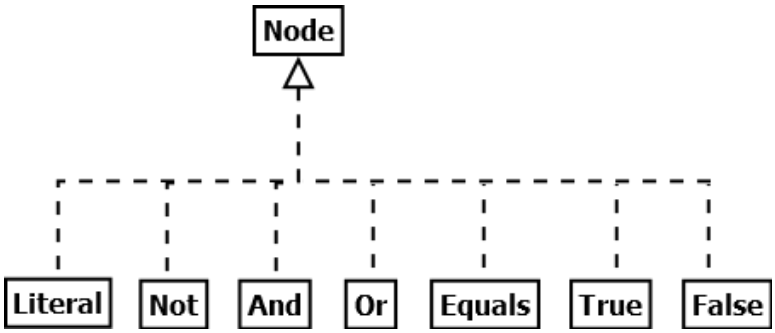


Figure 5.5: Simplified Class Diagram for `prop4j` Node. Only includes Nodes that are relevant to this Thesis.

For this thesis, we extended the abstract class `Node` with the classes `True` and `False`, which represent literals that are a tautology or a contradiction. We also extended the Node class by adding the method `replaceLiterals()`. This method takes a list of features that should be replaced with `false` in the specified Node as an input. One can also specify if the propositional formula should be resolved and simplified further by passing a boolean `resolve`. This method is based on Algorithm 1 from the concept chapter. This method is used to both modify a feature model's constraints and later to modify Antenna preprocessor statements.

[2] http://spl2go.cs.ovgu.de/projects/1

https://doi.org/10.24355/dbbs.084-202011201423-0

5.5 Composer-Dependent Steps

In FeatureIDE, composers offer developers different ways to generate software products [MTS+17a]. The mapping between features and source code in each of these composers is handled differently. In this section, we give an overview over the syntax and underlying mechanics of Antenna and FeatureHouse. We then explain how the assets are modified in the derivation process using an example. This is equivalent to the third step of our our approach to solving the problem described in Section 4.2 in Chapter 4. Furthermore, we explain why we chose to extend these specific composers with the capability to derive partial feature projects.

5.5.1 Antenna

One of the composers we use to derive subset software product lines is Antenna[3]. It is a simple preprocessor for Java files that allows the user to include or exclude code based on the truth-values of features. It is implemented in FeatureIDE as a composer. We chose Antenna, because it is very intuitive and easy to understand and, thus, can be used to illustrate the derivation process. However, the simplicity of preprocessor annotations has a downside. Preprocessor annotations are written directly in a software project's source code, which is a common point of criticism because it can make code hard to read and maintain [BM01][EBN02][MKR+].

Each line of code can be annotated by `//#if condition expression //#endif`. The annotation corresponds to the asset's propositional formula U, while the enclosed code corresponds to its payload s. If the condition evaluates to `true`, the lines enclosed by the `//#if` and `//#endif` annotations are turned into java comments and thus deactivated. In addition to `//#if`, Antenna also supports `//#else` and `//#elif` annotations [MTS+17b].

```
68
69          // #if Iris_Scan
70  //@      auth = new IrisScan();
71          // #elif PIN
72  //@      auth = new PINAuthentication(scan);
73          // #elif Password
74  //@      auth = new PasswordAuthentication(scan);
75          // #elif Facial_Recognition
76  //@      auth = new FacialRecognition();
77          // #endif
78
```

Figure 5.6: Code Excerpt Showing a Series of Antenna Preprocessor `#elif` Annotations.

Figure 5.6 and Figure 5.7 show two example code excerpts from the ATM SPL from Chapter 2. Figure 5.6 shows a series of mutually exclusive code blocks. Depending on which authentication method is chosen, the object `auth` is initialized with a different constructor. Figure 5.7 shows the initialization of a language selection menu. The annotation makes it so the menu is only built if at least two languages are selected.

[3] http://antenna.sourceforge.net/

https://doi.org/10.24355/dbbs.084-202011201423-0

```
125     // #if (English && German) || (German && French) || (English && French)
126 //@ public void setLanguage(Locale locale) {
127 //@     if (!currentLocale.equals(locale)) {
128 //@         currentLocale = locale;
129 //@         initialize(false);
130 //@         buildMainMenu();
131 //@     }
132 //@ }
133     // #endif
```

Figure 5.7: Code Excerpt Showing a Complex Antenna Preprocessor Annotation.

We explain how these code excerpts are modified when deriving a subset SPL from the ATM SPL using the configuration from Figure 2.2. We remove code blocks that are unreachable due to the removal of the features `French` `Facial_Recognition`, and `Iris_Scan`. Additionally, we want to modify annotations, such that removed features are no longer used as variables. We extended the Antenna Preprocessor with a number of methods to accomplish this task. These methods can be seen in Figure 5.8, we will explain them in detail in the next paragraphs.

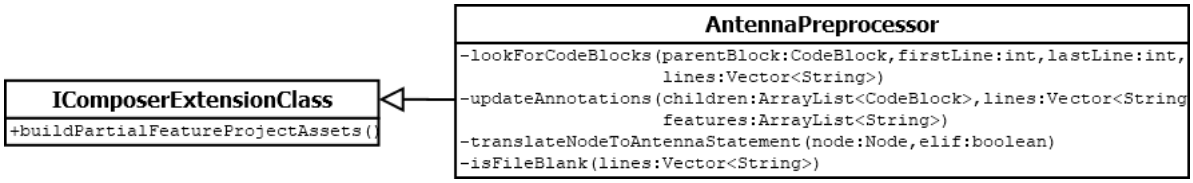


Figure 5.8: Simplified Class Diagram of AntennaPreprocessor, Showing the Methods we Implemented.

The implementation of the method `buildPartialFeatureProjectAssets()` in the class `AntennaPreprocessor` iterates over all files in the specified folder. For each .java file, it first identifies a list of code blocks using the method `lookForCodeBlocks()`. This method returns a list. The `CodeBlock` class is a data type that holds the first line, the last line, and the code block's annotation as a `prop4j` Node. It may also have children, which are code blocks contained inside of the code block. The method `lookForCodeBlocks()` iterates over the lines of each file and looks for an `//#if` condition. It then looks for an `//#endif`, `//#elif` or `//#else` annotation that ends this code block and the code block object is created.

For the codeblock in Figure 5.6 for example, it creates an `IfBlock` object and four `ElifBlock` ojects. The propositional formula of `IfBlock` objects is exactly the preprocessor statement. For example, the propositional formula of the `IfBlock` is

Iris_Scan.

The propositional formula of an `ElseBlock` consists of a conjunctions of the negations of all prior propositional formulas. Each `ElifBlock` additionally has a conjunction of its own annotation added. For example, the propositional formula of the last `ElifBlock` in Figure 5.6 is:

P = (Facial_Recognition ∧¬Iris_Scan ∧¬PIN ∧¬Password).

https://doi.org/10.24355/dbbs.084-202011201423-0

Then, the method is called recursively for the lines between the first and last line of the code block, to identify its child code blocks.

The .java file's lines and its list of code blocks are then passed to the method `updateAnnotations()`. This method evaluates each code block's node (`beforeNode`) before and after the removal of deselected features with the method `replaceLiterals()` (`afterNode`). Code blocks that do not contain features that are deselected in the configuration are left unchanged. Then, a decision is made for every other code block whether to keep it, change the preprocessor annotation, delete the preprocessor annotation, or to delete it and its child code blocks.

- Node is now a contradiction $\Longrightarrow$ code block and all of its children are removed.
- Node is now a tautology $\Longrightarrow$ the code block's annotation is removed, the associated code remains.
- Node has a solution, but is not a tautology $\Longrightarrow$ the code block's annotation is modified, all deselected features are set to false and the formula is simplified.

```
68
69
70
71 //      #if PIN
72 //@     auth = new PINAuthentication(scan);
73 //@     elif Password
74 //@     auth = new PasswordAuthentication(scan);
75 //@
76 //@
77          // #endif
78
```

Figure 5.9: Code Excerpt Showing a Series of Antenna preprocessor `#elif` Annotations.

```
125 //  #if English  &&  German
126 //@ public void setLanguage(Locale locale) {
127 //@     if (!currentLocale.equals(locale)) {
128 //@         currentLocale = locale;
129 //@         initialize(false);
130 //@         buildMainMenu();
131 //@     }
132 //@ }
133     // #endif
```

Figure 5.10: Code Excerpt Showing a Complex Antenna Preprocessor Annotation.

The results of this process for the examples in Figure 5.6 and Figure 5.7 with the configuration in Figure 2.2 can be seen in Figure 5.9 and Figure 5.10. The first and the fourth annotations in Figure 5.6 became contradictions with `Iris_Scan` and `Facial_Recognition` being replaced with `false`. The enclosed Java code was removed as well. The second annotation had it's propositional formula changed from

https://doi.org/10.24355/dbbs.084-202011201423-0

¬ `Iris_Scan` ∧ `PIN` to just `PIN`, and was thus changed to an `// #If` notation. With the feature `French` being replaced with `False`, the clauses (`German` && `French`) and (`English` && `French`) were found to be contradictions, and thus removed, the annotation was changed to reflect this.

5.5.2 FeatureHouse

FeatureHouse is a feature-oriented composition mechanism that is also supported by FeatureIDE[MTS+17a]. FeatureHouse generates software products by merging Feature Structure Trees (FST) of each feature[AKL09]. An FST is a hierarchical representation of a software artifact. Each inner node of an FST represents the structure of a software artifact (e.g. in java folders, packages, classes, methods,..) and determine the location. Only the leaf nodes of an FST represent the contents (e.g. in java the instructions inside a method). FeatureHouse is built such that it supports the merging of FSTs for any format of which it can generate FSTs. FeatureHouse is open-source and can be extended[4].

We chose FeatureHouse, because it is widely used in SPL research [AVF17] [BAS15] [SKLA12]. Moreover, FeatureIDE has a wide variety of example SPLs using FeatureHouse, which allow us to test our implementation under real-world circumstances.

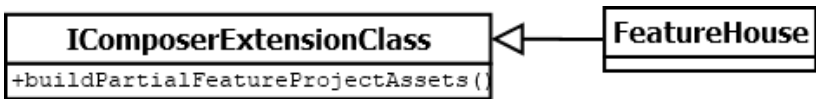


Figure 5.11: Simplified class diagram of FeatureHouse, showing the method we implemented.

Figure 5.11 shows the method we implemented to modify the assets of FeatureHouse projects when deriving a subset SPL. The implementation of the method `buildPartialFeatureProjectAssets()` in the `FeatureHouseComposer` class iterates over the FeatureIDE project's feature folders and deletes all feature folders of deselected features.

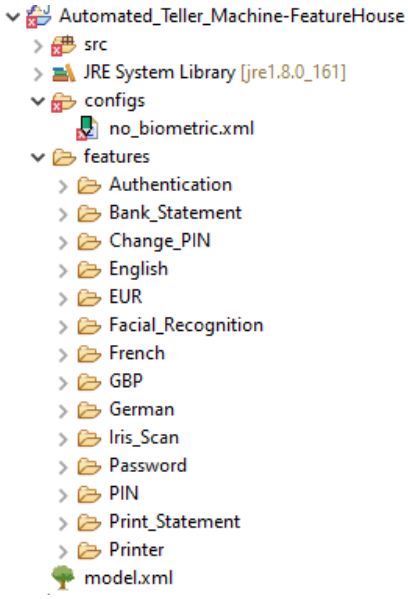
The mapping between features and code with FeatureHouse is much more straightforward than Antenna. Each feature in FeatureHouse has a feature folder containing structure and code that is merged with the other folders, which is the payload s. Each feature folder can be included or excluded, each assets propositional formula U is just the corresponding feature. Due to this, there are no complicated feature relations like in Antenna. Adding a feature will for example never *exclude* code from the eventual product.

Deselecting a feature turns the propositional formula of an asset in FeatureHouse into a cotradiction. Therefore, the third step our approach to the derivation of subset SPLs with partial configurations outlined in Section 4.2 is realized by deleting the feature folders of each deselected feature.

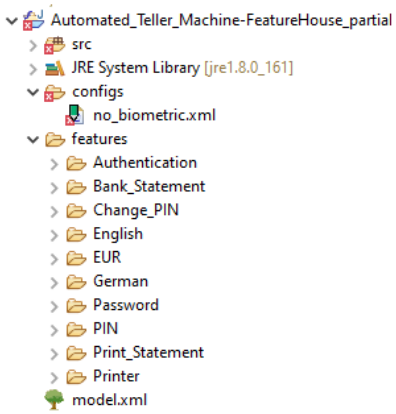
5.12(a) shows an example FeatureIDE project implemented with FeatureHouse. 5.12(b) shows the subset of that FeatureIDE project generated with the configuration from Figure 2.2.

[4]https://github.com/joliebig/featurehouse

https://doi.org/10.24355/dbbs.084-202011201423-0



(a) ATM Software Product line implemented with FeatureHouse.



(b) Subset of ATM Software Product line based on configuration from Figure 2.2 implemented with FeatureHouse.

https://doi.org/10.24355/dbbs.084-202011201423-0

6. Evaluation

In this chapter, we evaluate our procedure to derive subset SPLs with partial configurations. In order to do this, first, we first formulate research questions. Then, we introduce projects from different domains that we use as an input for our evaluations. We then present results from our tests and conclude on our research questions.

6.1 Research Goal

The goal of this thesis is to explore the possibilities of partial configurations for the derivation of subset software product lines and provide a usable framework for further research. To accomplish this, we develop a prototype that allows the derivation of subset software product lines and show its correctness.

To show that we have accomplished this goal, we need to demonstrate for a variety of different inputs that our implemented solution delivers correct results. For this to be true, each valid configuration from a subset SPL S_1 has to generate the same product from SPLs S_1 and S_0. Additionally, we need to demonstrate that it is *usable* under real-world circumstances. This means that our approach to deriving subset SPLs needs to scale to deliver solutions for large software product lines with hundreds of features and source code files in a reasonable amount of time.

6.2 Research Questions

In this section, we specify the research questions that we are attempting to answer in this chapter.

- `RQ1`: Are the products generated from subset SPLs equal to corresponding products generated from the original SPL?
- `RQ2`: What is the runtime of the derivation of subset SPLs?

https://doi.org/10.24355/dbbs.084-202011201423-0

6.3 Selection of Software Product Lines

FeatureIDE offers a variety of example software product lines. All software product lines we use to evaluate our implementation are available in the FeatureIDE github repository[1] and can be accessed via the FeatureIDE example wizard. In this section, we briefly introduce each of these SPLs and explain our reasoning behind using them.

We have decided to implement a command-line dialog in Antenna for the ATM software product line from our motivating example and evaluate it. We have added this example to the FeatureIDE example wizard under the name `Automated_Teller_Machine-Antenna` to offer more data for preprocessor-based composition mechanisms. We chose this example for our evaluation, because readers of this paper are already familiar with it through prior chapters.

One of the FeatureIDE examples is a small software product line `Elevator-Antenna-v1.4` implemented with Antenna. This software product line represents software simulating the movement of an elevator and its control panels. These aspects can be configured in a variety of ways [MTS+17a]. There is also an equivalent implementation for FeatureHouse called `Elevator-FeatureHouse-v1.1`. We have chosen these examples because they allow for a direct comparison between the two approaches.

Another software product line we use for our evaluation is `GPL-FH-Java`. It is an implementation of a graph data type. The features allow for different types of graphs to be created, for example graphs with weighted and unweighted edges. Additionally, algorithms that can be run on the graphs can be selected. There are also cross-tree constraints, e.g. the selection of Kruskal's algorithm mandates the selection of weighted edges. Lopez-Herrejon et al. proposed this product line as a standard problem for the evaluation of operations on product lines [Bos01].

Last, we chose the example software product line `BerkeleyDB-FH` to measure the time it takes to derive subset SPLs from it. This SPL was composed by Apel et al. to demonstrate the scalability of FeatureHouse [AKL09]. Berkeley DB is a software library that offers operations for for key/value databases. We chose BerkeleyDB to test the performance of our tool with a large SPL and this is the largest software product line implemented with either Antenna or FeatureHouse that we have access to.

Figure 6.1: Table Showing all SPLs used in the Evaluation

	#Features	#Constraints	#Preprocessor Directives	#Feature Folders
Automated_Teller_Machine-Antenna	22	2	93	N/A
Elevator-Antenna-v1.4	21	3	109	N/A
Elevator-FeatureHouse-v1.1	21	3	N/A	9
GPL-FH-Java	38	16	N/A	27
BerkeleyDB-FH	119	68	N/A	99

[1] https://github.com/PaulWestphal/FeatureIDE-subsetSPL

https://doi.org/10.24355/dbbs.084-202011201423-0

Figure 6.1 gives an overview over the selection of software product lines that we use to evaluate our implementation.

6.4 Evaluation process

In this section, we explain the evaluation process. First we briefly state each step. Then, we describe each step in more detail and explain how these steps help us answer the research questions.

1. Derive `subset SPLs` from `Original SPL_0`.
2. Generate all valid products from these `subset SPLs`.
3. Generate these products from the `Original SPL_0`.
4. Compare valid products generated from `subset SPLs` to the same valid products generated from `Original SPL_0`.

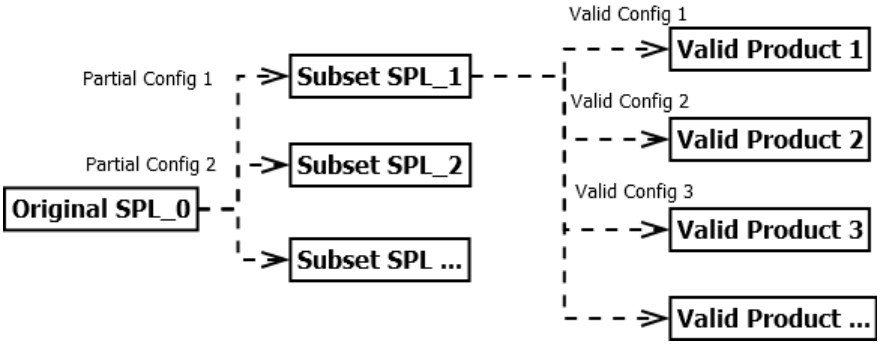


Figure 6.2: Diagram Showing the Evaluation Process

Figure 6.2 shows how we derive subset SPLs and products for the evaluation. For each software product line `Original SPL_0`, we randomly generate five partial configurations. We aimed to leave progressively more features undecided with each configuration. The first configuration is a full configuration, the second has a few features left undecided and the fifth one has almost no decisions made yet. We decided on this process to cover a wider variety of different configurations. We use these partial configurations to derive `Subset SPL_1` to `Subset SPL_5`. For each of these derivation processes, we record how much time they take to gather data to answer `RQ2`. Then, we generate all valid products (up to a maximum of 100) from the newly generated `Subset SPL`.

Figure 6.3 shows the comparison step. We use the same configurations to generate all valid products for the original and the derived software product line. We then compare the assets of each valid product of the `Subset SPL` with the product generated from the `Original SPL` with that same configuration. We do this by recursively iterating over the folder structure of the generated source files of the `Original SPL` and comparing their contents with the corresponding files from the subset SPL.

If we are able to empirically demonstrate correct results for each of the different software product lines and randomly generated configurations, we can answer `RQ1`.

https://doi.org/10.24355/dbbs.084-202011201423-0

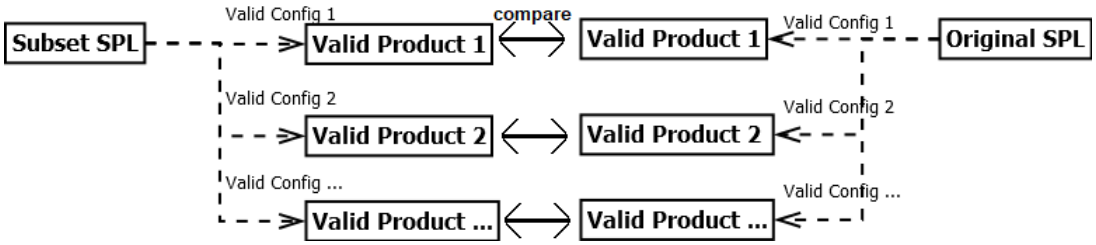


Figure 6.3: Diagram Showing Comparison between Products Generated from Original SPL and Subset SPL

Hardware and Software used for Evaluation

In this section, we specify the hardware and software we used to run the evaluation of our implementation.

- Intel Core i5-6600k CPU 4 cores, 4 threads @ 3500 MHZ
- 2x8 GB DDR-4 RAM @ 3000 MHZ
- 525GB Crucial MX300 SSD
- Windows 10 64 bit
- Eclipse Photon 4.8.0
- Java 1.8

This is important context for `RQ2` because each of these factors may change the time needed for all operations to finish. The implementation and performance of some of Eclipse's methods we used may have changed with more recent versions of Eclipse, or may change in the future.

6.5 Evaluation of Correctness of Valid Products of Subset SPLs

In this section, we evaluate the correctness of valid products derived from subset SPLs, by comparing them to the same products generated from the SPLs the subsets were derived from. First, we present the data we collected for this evaluation. Then, we discuss the implications of this data for the correctness of our approach.

Table 6.1 to Table 6.4 show the comparison results. For each configuration used in the evaluation process, we show the number of selected, deselected, and undecided features. The row "#.java File Comparisons Passed" denotes the number of `.java` files that were equal in generated products from the `Original SPL` and the `Subset SPL`, while the row "#.java File Comparisons Failed" denotes the number of unequal files. Before doing a line-by-line comparison on each file, we removed all Java comments and leading and trailing whitespaces. We did this to avoid possible errors from differing indentations, despite the contents of the files being equal.

https://doi.org/10.24355/dbbs.084-202011201423-0

Automated_Teller_Machine-Antenna

Configuration Index	1	2	3	4	5
#Selected Features	15	6	3	3	0
#Deselected Features	7	5	4	2	4
#Undecided Features	0	11	15	17	18
#Valid Configurations	1	16	60	75	336
#.java File Comparisons Passed	24	384	1440	1800	2400
#.java File Comparisons Failed	0	0	0	0	0

Table 6.1: Comparison Results of `Automated_Teller_Machine-Antenna`

Elevator-Antenna-v1.4

Configuration Index	1	2	3	4	5
#Selected Features	13	5	4	2	1
#Deselected Features	8	5	3	3	3
#Undecided Features	0	11	14	16	17
#Valid Configurations	1	2	32	96	96
#.java File Comparisons Passed	12	24	384	1152	1152
#.java File Comparisons Failed	0	0	0	0	0

Table 6.2: Comparison Results of `Elevator-Antenna-v1.4`

Elevator-FeatureHouse-v1.1

Configuration Index	1	2	3	4	5
#Selected Features	13	5	4	2	1
#Deselected Features	8	5	3	3	3
#Undecided Features	0	11	14	16	17
#Valid Configurations	1	2	32	96	96
#.java File Comparisons Passed	12	24	384	1152	1152
#.java File Comparisons Failed	0	0	0	0	0

Table 6.3: Comparison Results of `Elevator-FeatureHouse-v1.1`

GPL-FH-Java

Configuration Index	1	2	3	4	5
#Selected Features	16	12	8	8	9
#Deselected Features	7	6	4	2	1
#Undecided Features	0	5	11	13	13
#Valid Configurations	1	12	44	96	102
#.java File Comparisons Passed	18	200	656	1484	1598
#.java File Comparisons Failed	0	0	0	0	0

Table 6.4: Comparison Results of `GPL-FH-Java`

Regarding `RQ1` (*Are the products generated from subset SPLs equal to these products generated from the original SPL?*), one can see from the tables that a total of 15452 comparison tests succeeded and a total of 0 comparison tests failed. In 100% of cases, the generated products of the subset SPL were equal to the corresponding

https://doi.org/10.24355/dbbs.084-202011201423-0

product generated from the original SPL. This indicates that our implementation of the derivation of subset SPLs indeed **delivers correct results**.

Our way of evaluating the correctness has limitations, however. For one, the set of products that can be derived from the `Subset SPL` may differ from the set of products that can be derived from the `Original SPL` starting from the `Partial Config`. This may occur if the propositional formula of the subset SPL is modified incorrectly. Additionally, despite attempting to cover a wide variety of domains and concepts with our selection of SPLs for the evaluation, is still a small sample and a test case with a bug may have been missed. Especially for Antenna, there is a lack of example SPLs with very long and complex preprocessor annotations.

6.6 Evaluation of Runtime of the Derivation Process

In this section, we evaluate the runtime of our implementation of the subset SPL derivation process. First, we present the data we collected for this evaluation. Then, we discuss the implications of this data for `RQ2`.

Automated_Teller_Machine-Antenna

Configuration Index	1	2	3	4	5
#Selected Features	15	6	3	3	0
#Deselected Features	7	5	4	2	4
#Undecided Features	0	11	15	17	18
#Valid Configurations	1	16	60	75	336
t in ms	1535	887	1149	968	1040

Table 6.5: Measured Time Results of `Automated_Teller_Machine-Antenna`

Elevator-Antenna-v1.4

Configuration Index	1	2	3	4	5
#Selected Features	13	5	4	2	1
#Deselected Features	8	5	3	3	3
#Undecided Features	0	11	14	16	17
#Valid Configurations	1	2	32	96	96
t in ms	3286	1471	1660	611	1866

Table 6.6: Measured Time Results of `Elevator-Antenna-v1.4`

Elevator-FeatureHouse-v1.1

Configuration Index	1	2	3	4	5
#Selected Features	13	5	4	2	1
#Deselected Features	8	5	3	3	3
#Undecided Features	0	11	14	16	17
#Valid Configurations	1	2	32	96	96
t in ms	2498	1254	964	1096	1104

Table 6.7: Measured Time Results of `Elevator-FeatureHouse-v1.1`

https://doi.org/10.24355/dbbs.084-202011201423-0

GPL-FH-Java

Configuration Index	1	2	3	4	5
#Selected Features	16	12	8	8	9
#Deselected Features	7	6	4	2	1
#Undecided Features	0	5	11	13	13
#Valid Configurations	1	12	44	96	102
t in ms	2176	2622	2318	1056	1393

Table 6.8: Measured Time Results of `GPL-FH-Java`

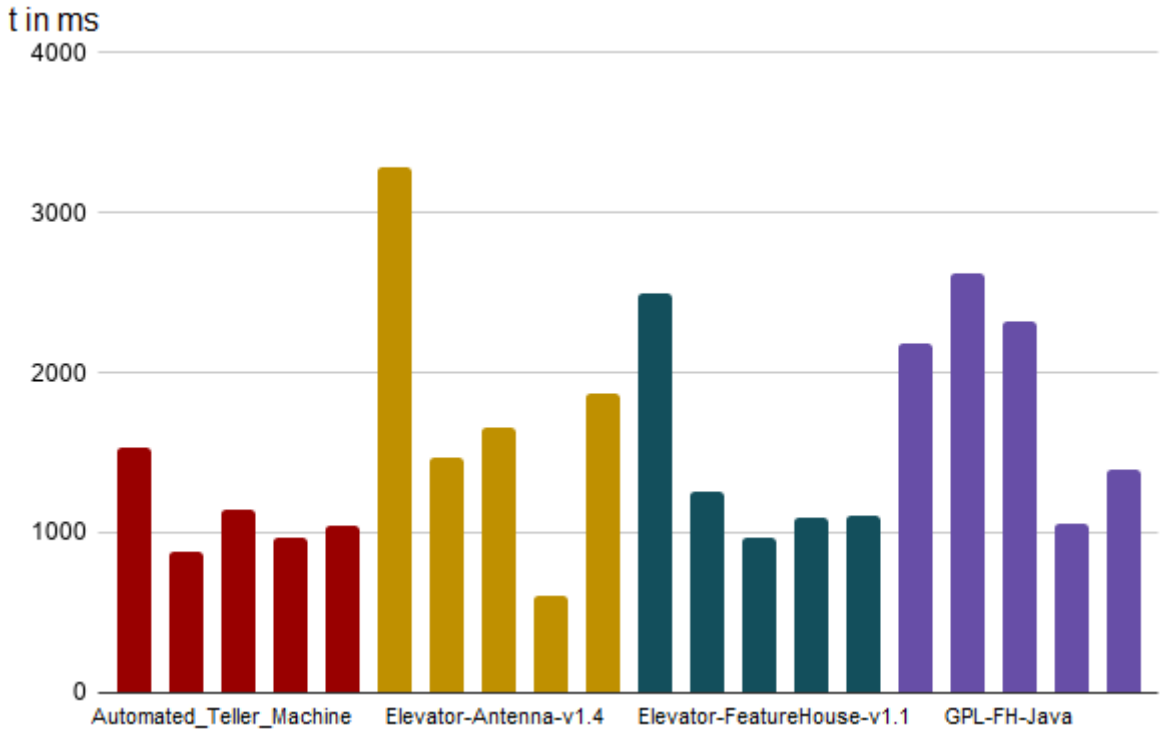


Figure 6.4: Diagram showing time needed for derivation of subset SPLs

Table 6.5 to Table 6.8 show the results of the first four example SPLs. Figure 6.4 shows the required time of configurations 1 to 5 for each SPL as a graph. The times required for the derivation of subset SPLs all lie very close together, with the fastest process being 611ms (`Elevator-Antenna-v1.4`, configuration 3) and the slowest being 3286ms (`Elevator-Antenna-v1.4` configuration 1).

During the evaluation process, we noticed a strong variation in the times required for the derivation process. Even restarting the operation with the same inputs delivered widely varying results. Therefore, we decided for the last example to separately record the time needed for operations on the feature model and its constraints and operations on the SPL's assets to get more insight. The results of this evaluation are shown below.

We found that operations on the feature model took only a few milliseconds, while operations that require writing in the file system (in this case, using Eclipse's `IResource.delete()` method to delete feature folders) took a comparatively very long time.

https://doi.org/10.24355/dbbs.084-202011201423-0

BerkeleyDB-FH-Java

Configuration Index	1	2	3	4	5
#Selected Features	72	62	52	35	1
#Deselected Features	47	25	29	13	6
#Undecided Features	0	32	38	71	112
#Valid Configurations	1	>5499	>5665	>6649	>5755
Modifying Feature Model t in ms	2	2	4	4	2
Modifying Assets t in ms	23372	16533	52893	16216	8407
Total t in ms	23374	16535	52897	16220	8409

Table 6.9: Measured time results of `BerkeleyDB-FH-Java`

`BerkeleyDB-FH-Java` is the biggest example SPL we chose and as such, deriving a subset SPL from it takes much longer than from the smaller SPLs. The shortest time required was 8.4 seconds (configuration 5), while the longest time was 52.9 seconds (configuration 3). We calculated the average time of all five configurations for each SPL and divided it by the number of files in each SPL. This data is shown in table Table 6.10.

	Avg t in ms	#.java Files	Avg t in ms per .java file
Automated_Teller_Machine-Antenna	1115.8	24	46.5
Elevator-Antenna-v1.4	1778.8	12	148.23
Elevator-FeatureHouse-v1.1	1383.2	43	32.17
GPL-FH-Java	1913	57	33.56
BerkeleyDB-FH-Java	23487	621	37.82

Table 6.10: Diagram showing time needed for derivation of subset SPLs

Regarding the long runtime of operations on the file system, we theorize that a multi-threaded implementation of the deletion of feature folders for FeatureHouse and the modifications of `.java` files for Antenna may significantly improve performance. The long runtime of the SPL `BerkeleyDB-FH-Java` is explained by the number of source files that need to be modified, since the average time per .java file is not exceptional.

To answer `RQ2`(*How much time is spent on the derivation of subset SPLs?*): Our prototype implementation of the derivation of subset SPLs using partial configurations delivers results within at most 52.9 seconds for a large SPL with 99 concrete features and 621 java files. We consider this a reasonable amount of time, and consequently the implemented protoypical approach to be usable for research.

https://doi.org/10.24355/dbbs.084-202011201423-0

7. Related Work

The difficulties of managing large software product lines and developing solutions for these problems have been a research topic for a long time. In this chapter, we present a selection of related publications and discuss similarities and differences to our work.

Feature Model Slicing

Acher et al. [ACLF11] introduced feature model slicing, which aims to remove features from feature models while preserving implicit dependencies between remaining features. We also developed a method to reduce the size of a feature model, but there are stark differences between our approaches that can be illustrated with an example. When removing feature B with slicing, the propositional formula $(A \implies B) \wedge (B \implies (C \wedge D))$ results in $A \implies (C \wedge D)$ [KST+16] [ACLF11]. The dependencies between the variables A, C, and D are kept intact. Our approach on the other hand, eliminates unwanted features by setting them to $false$ and simplifying the propositional formula. With this approach, removing feature B from the constraint $(A \implies B) \wedge (B \implies (C \wedge D))$ returns $\neg A$ as a constraint in the subset SPL's feature model.

Acher et al. introduced an algorithm for feature model slicing in 2011[ACLF11]. Krieter et al. concluded that this algorithm does not scale for large feature models with thousands of features and proposed a new algorithm for feature model slicing with a better runtime for large feature models and the removal of more than 30% of features [KST+16] [KSST].

Variant-Preserving Refactoring

Schulze et al. [STKS12] discuss the limitations of refactoring to improve code in feature-oriented programming. They conclude that the techniques usually used for refactoring are not sufficient for feature-oriented SPLs. This is because regular refactoring techniques only have to preserve the same behavior for one product,

https://doi.org/10.24355/dbbs.084-202011201423-0

while refactoring in a feature-oriented SPL has to preserve the behavior for all products that can be generated from that SPL. Schulze et al. define a refactoring as variant-preserving, if after refactoring each valid configuration remains valid and each product generated from a valid configuration that is compilable has the same external behavior [STKS12]. To apply this concept, they also developed a tool for the removal of code clones.

Our work is similar in the way that we create a new software product line from an existing SPL in which all products generated from valid configurations must produce the same behavior. The difference is that our approach specifically reduces the set of valid variants in the subset SPL.

Preprocessors

Meinicke et al. [MTS+16] integrated preprocessors in FeatureIDE, which made the derivation of subset SPLs with Antenna possible in the first place. They implemented support for the development of variable software using preprocessors with tools such as visualization (through feature models) and feature traceability (through existing FeatureIDE views and color highlighting of features).

Deriving subset SPLs from preprocessor projects can also be used as a tool to make code more legible. Subset SPLs may be used as a view if one selects features to exclude from the preprocessor project. This way unwanted and distracting code or even entire files can be removed and developers can focus on the task at hand.

https://doi.org/10.24355/dbbs.084-202011201423-0

8. Conclusion and Future Work

Large softwar eproduct lines are typically complex to analyze and, thus, automated support is required. The goal of this thesis was to support the configuration and development for software product lines, by developing a procedure to derive a reduced version of an existing SPL. In order to do this, we introduced the concept of subset SPLs and developed algorithms which can be used to generate a subset SPL based on a configuration.

We realized these concepts by implementing them in FeatureIDE, a framework for feature-oriented software development. We chose the two composers Antenna and FeatureHouse and extended them with the capability to remove features from both a feature model and the FeatureIDE project's assets.

Our evaluation strongly indicates that the implementation we developed does indeed produce correct results. We compared a total of 15452 .java files generated from subset SPLs and the original SPLs and our procedure computes only correct results. The longest runtime we measured for the derivation of subset SPLs was 52 seconds in a test case with 621 .java files. We believe that this runtime could be improved in the future by implementing work on each .java file in a multi-threaded environment.

Currently, the derivation of subset SPLs is a one-way street; a completely standalone SPL is created with no direct connection to the old one. In some instances, developers may want to reuse code that has been developed with a subset SPL and integrate it in the original one. In the future, a technique could be developed to automatically merge subset SPLs with the original SPL again. This would allow users to easily integrate any work done on a subset SPL back into the original.

While we expect various benefits from working on a subset SPL, we can not yet quantify the benefits. An interesting topic for research could be a study examining the advantages of working on a subset SPL. We believe that subset SPLs can simplify reading, understanding, and subsequently working on large software product lines by excluding unwanted code. However, exactly how much time and utility can be gained from our approach must be studied in practice.

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

A. Appendix

Propositional formula of ATM Feature Model

{(Automated_Teller_Machine), (Bank_Statement ∨ ¬ Print_Statement), (Language ∨ ¬ English), (Language ∨ ¬ German), (Language ∨ ¬ French), (English ∨ German ∨ French ∨ ¬ Language), (PIN ∨ ¬ Change_PIN), (Authentication ∨ ¬ PIN), (Authentication ∨ ¬ Password), (Authentication ∨ ¬ Facial_Recognition), (Authentication ∨ ¬ Iris_Scan), (PIN ∨ Password ∨ Facial_Recognition ∨ Iris_Scan ∨ ¬ Authentication), (¬ PIN ∨ ¬ Password), (¬ PIN ∨ ¬ Facial_Recognition), (¬ PIN ∨ ¬ Iris_Scan), (¬ Password ∨ ¬ Facial_Recognition), (¬ Password ∨ ¬ Iris_Scan), (¬ Facial_Recognition ∨ ¬ Iris_Scan), (Currency ∨ ¬ EUR), (Currency ∨ ¬ GBP), (EUR ∨ GBP ∨ ¬ Currency), (¬ EUR ∨ ¬ GBP), (Automated_Teller_Machine ∨ ¬ Language), (Automated_Teller_Machine ∨ ¬ Currency), (Automated_Teller_Machine ∨ ¬ Bank_Statement), (Automated_Teller_Machine ∨ ¬ Printer), (Automated_Teller_Machine ∨ ¬ Authentication), (Language ∨ ¬ Automated_Teller_Machine), (Currency ∨ ¬ Automated_Teller_Machine), (Authentication ∨ ¬ Automated_Teller_Machine), (¬ Print_Statement ∨ Printer)}

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

Bibliography

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing feature models. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 424–427, Lawrence, KS, USA, November 2011. IEEE. (cited on Page 35)

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *2009 IEEE 31st International Conference on Software Engineering*, pages 221–231, Vancouver, BC, Canada, 2009. IEEE. (cited on Page 14, 25, and 28)

[AVF17] Guilherme Assis, Gustavo Vale, and Eduardo Figueiredo. Feature oriented programming in Groovy. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development - FOSD 2017*, pages 21–30, Vancouver, BC, Canada, 2017. ACM Press. (cited on Page 25)

[BAS15] Sandy Beidu, Joanne M. Atlee, and Pourya Shaker. Incremental and Commutative Composition of State-Machine Models of Features. In *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, pages 13–18, Florence, Italy, May 2015. IEEE. (cited on Page 25)

[BM01] I.D. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 281–290, Stuttgart, Germany, 2001. IEEE Comput. Soc. (cited on Page 22)

[Bos01] Jan Bosch, editor. *Generative and component-based software engineering: third international conference, GCSE 2001, Erfurt, Germany, September 10-13, 2001: proceedings*. Number 2186 in Lecture notes in computer science. Springer, Berlin ; New York, 2001. Meeting Name: GCSE 2001. (cited on Page 28)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, September 2010. (cited on Page 1 and 7)

https://doi.org/10.24355/dbbs.084-202011201423-0

[EBN02] M.D. Ernst, G.J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002. (cited on Page 22)

[HFACA13] Ruben Heradio, David Fernandez-Amoros, Jose A Cerrada, and Ismael Abad. A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering*, 23(08):1177–1204, 2013. Publisher: World Scientific. (cited on Page 7)

[KSST] Sebastian Krieter, Reimar Schröter, Gunter Saake, and Thomas Thüm. An Efficient Algorithm for Feature-Model Slicing. page 7. (cited on Page 35)

[KST+16] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*, pages 60–64, Beijing, China, 2016. ACM Press. (cited on Page 35)

[MKR+] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. page 24. (cited on Page 22)

[MTS+16] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. FeatureIDE: taming the preprocessor wilderness. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 629–632, Austin, Texas, 2016. ACM Press. (cited on Page 36)

[MTS+17a] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer International Publishing, Cham, 2017. (cited on Page 17, 22, 25, and 28)

[MTS+17b] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer International Publishing, Cham, 2017. (cited on Page 22)

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering: foundations, principles, and techniques.* Springer, New York, NY, 1st ed edition, 2005. (cited on Page 7)

[SKLA12] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. Comparing program comprehension of physically and virtually separated concerns. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development - FOSD '12*, pages 17–24, Dresden, Germany, 2012. ACM Press. (cited on Page 25)

https://doi.org/10.24355/dbbs.084-202011201423-0

[STKS12] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, pages 73–81, Leipzig, Germany, 2012. ACM Press. (cited on Page 35 and 36)

https://doi.org/10.24355/dbbs.084-202011201423-0

https://doi.org/10.24355/dbbs.084-202011201423-0

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

Braunschweig, den 26. Juli 2020

https://doi.org/10.24355/dbbs.084-202011201423-0