The Benefits of a Feature Model in Banking

Experience Report

Claudia Fritsch
KfW
Frankfurt am Main, Germany
claudia.fritsch@kfw.de

Richard Abt
KfW
Frankfurt am Main, Germany
richard.abt@kfw.de

Burkhardt Renz
Technische Hochschule Mittelhessen
Gießen, Germany
burkhardt.renz@mni.thm.de

ABSTRACT

This experience report describes the surprisingly beneficial introduction of feature modeling at KfW, a government promotional bank. On behalf of the government and based on promotional directives, KfW grants retail loans to small and medium enterprises, business founders, self-employed professionals, municipalities and private individuals. The promotional directives, called programs, define mandatory and optional properties of these loans. We have now successfully built a feature model from these properties.

Our feature model will be presented with its outstanding characteristic, which is an additional subtree containing the programs as features. Complete and correct cross-tree constraints will also allow us to analyze and scope the portfolio, reduce complexity, and speed-up time-to-market. This is the advent of product line development at KfW.

In order to standardize our portfolio, we have subsequently developed tools on top of the feature model, namely, a browser-based, multi-user configurator assisting non-technical-affine users in their product design, and a generator producing complete product documentation from the feature model and partial configurations. More applications are currently underway.

This is our story of applying Software Product Line Engineering in banking, a domain where it is unusual or even unknown. We share our ideas, analyses, progress, and findings where the results have been thrilling us for the past two years and will continue to do so.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; *Requirements analysis*; • **Human-centered computing** → Information visualization; • **Applied computing** → *Online banking*; • **Social and professional topics** → *Automation.*

KEYWORDS

Software Product Line Engineering, Feature Modeling, Partial Configuration, Document Generation, Retail Loans, Mass Customization, Experience Report

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414946

ACM Reference Format:
Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. The Benefits of a Feature Model in Banking: Experience Report. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3382025.3414946

1 INTRODUCTION

1.1 KfW

Founded in 1948 as a bank under public law and equipped with funds of the European Recovery Program (ERP, "Marshall Plan"), the Kreditanstalt für Wiederaufbau (KfW) has been giving loans to companies, private individuals and German municipalities for more than 70 years.

Today, KfW is one of the world's leading promotional banks, and one of Germany's largest banks. On behalf of the Federal Republic of Germany and the Federal States, KfW finances projects worldwide, supports economic and social progress in developing and transition countries, and promotes domestic investments in Germany.

Domestic promotion is off-the-shelf retail banking, and in this report, we only consider KfW retail banking. The main financial instruments are particularly favorable loans, reduced in price by KfW funds or federal sponsorship.

1.2 KfW Retail Programs and Loans

Each year, KfW grants hundreds of thousands of retail loans. Most of the properties that characterize such a loan are predefined by a promotional directive called *KfW retail program.*

A KfW retail program is usually requested by a German ministry and designed by a KfW credit domain expert. Each program aims at a certain promotional goal, such as environmental protection, renewables, energy efficient production, home ownership, education or founding. Currently, KfW offers 55 different retail programs.

To define a retail program, a KfW domain expert authors a *program information sheet,* which describes the program in all possible variants over 4 – 12 pages. Many program information sheets contain one or more links to other documents, containing further details, such as minimum technical requirements or state aid rules.

Usually, a *retail loan* is a financial service with a certain cash flow profile. KfW retail loans have many more properties permitting many more loan configurations, resulting in about 3,700 different loan products. This wide variety in retail loans is unique for German banks.

1.3 KfW Loan Life Cycle

Most KfW retail loans are brokered by on-lending banks, because KfW does not have any retail branches. If a potential borrower

wants to apply for a KfW loan, the software system of the on-lending bank must request a KfW loan under a certain KfW retail program by calling a KfW web service. The KfW banking software checks the loan application, and if it is valid, it will respond within seconds by issuing an immediate confirmation, which includes a loan agreement.

The KfW banking software creates a loan account in the KfW database, the most important parameters are:

- loan id
- borrower
- on-lending bank
- date of loan agreement
- loan amount
- interest rate
- intended purpose
- loan term (the number of years until the loan has to be fully repaid)
- grace period (the number of years where the borrower only pays interest)
- interest rate period (the number of years that the first interest rate remains fixed)
- repayment pattern

The borrower accepts the loan agreement by requesting the loan payment. After the grace period is over, the borrower starts to repay his debt. The borrower must then prove that he has used the payment to finance the intended purpose as provided in the loan agreement.

During the remaining loan term, the KfW banking software supervises the loan account and takes care of all ordinary and extraordinary circumstances, such as prolongation, change of repayment schedule, change of borrower, etc. After the borrower has fully repaid the loan, the loan account is closed.

1.4 Complexity of the Portfolio

For the past ten years each KfW retail program has been developed and maintained separately. Domain experts working in different divisions design, describe and maintain meticulously the retail programs in a manual process. During program development, they have to respect the demands of the ministries. In doing so they have introduced new properties, values, rules and dependencies, which have increased the complexity of the portfolio.

KfW loans are subsidized, either by a reduced interest rate, or by a grant, or both. Usually, the amount of the grant is defined as the minimum of a certain percentage of the loan amount and an absolute maximum value. The percentage and the maximum value depend on the program and on the quality of the subsidy standard that the borrower achieves through the investment measures.

For instance, a new, energy efficient house that consumes only 40 percent of the energy compared to a reference house defined by the German Energy Saving Regulation (EnEV), is eligible to a 20 % grant, at most €24,000. A new house that consumes 55 % of the energy is eligible to a 15 % grant, at most €18,000.

However, in programs where such a reference does not exist, the amount of the grant is computed *individually,* e.g., for deep geothermal energy the amount of the grant depends on the borehole depth. These exceptions seem to be inevitable and add complexity to the portfolio.

Portfolio complexity has been growing constantly. It complicates loan allocation and processing. The business logic that is implemented in the KfW banking software needs to be adapted to each program, sometimes even to program variants. Development costs have been rising and *time-to-market is several times longer* than it used to be.

We have observed that it would be impossible to simplify the banking software if the portfolio remained as complex as it was. Better time-to-market could only be achieved if the retail programs were better structured and easier to manage.

2 MOTIVATION

2.1 Who Are We?

We are a team of KfW domain experts and business analysts working at the interface between the credit and the IT departments. Our education and experiences cover banking, business administration, economics, mathematics, computer science and software engineering, particularly software product line development.

In 2017 we were assigned the task to "modularize" the KfW retail programs. The idea was: if programs were assembled from pre-defined reusable parts, the loan processing software could be assembled from pre-defined reusable components, and time-to-market would speed-up. We were supposed to find these reusable parts. Unfortunately, but not unexpectedly, that turned out to be a long-term project.

Although Product Line Engineering is usually applied in domains of tangible software-intensive systems, we believe that it is also applicable in our domain of financial services. This is so, because the KfW programs implement the KfW mission, share a common set of features, and are processed by a software-intensive system built from a common set of assets [6].

We also believe that Product Line Engineering should be applied to the very early stages of development, i.e., retail program development, documentation and information, before we consider applying it in software development.

2.2 Our Goals

Our primary goal is to speed-up time-to-market and to reduce development costs.

To do this, we must shorten the development time of retail programs and software. To shorten development time, we must standardize our products. To standardize our products, we must reduce the complexity of the portfolio.

Our journey to achieve these goals consists of the following steps:

- We have *explored and visualized* our loan portfolio in a feature model (see section 3).
- We have been *analyzing the complexity* of the portfolio using a configuration editor (see section 4).
- We have developed tools that *define standards* and *prevent growth of complexity* (see section 5).
- While new applications of the feature model are underway, our organization has begun the transition from developing single programs to *modeling the portfolio* (see section 6).

2.3 A First Attempt

In order to explore and visualize the complexity of our portfolio, we started to collect properties of our programs and put them in a table.

We figured that each program could be defined by a set of properties and their values. Properties include recipients and intended purpose, eligible investment locations, loan terms and grace periods, interest rate periods and repayment terms, fees and grants. Property 'recipients' can take values like private individual, entrepreneur or company.

We found these properties mainly in the problem space, but also in the solution space. In the problem space, program information sheets, FAQs, precedences and expert knowledge describe properties and values. In the solution space, a subschema of our database describes program properties and contains coded values.

We selected two retail programs and re-engineered their properties and values that distinguish these programs.

As a first attempt to visualize the properties, we created an Excel® sheet, where each column represented a program property, and each property could take several values. Each row was a possible combination of values of properties, and represented a configuration of one loan.

Each value of a property we added to the Excel® sheet almost doubled the number of rows. Soon, our Excel® sheet had 40 columns and eight thousand rows. We tried to discover commonalities, variabilities and exceptions, but there was so much redundancy in the rows that we did not see the wood for the trees.

As a means to visualize complexity the Excel® sheet had failed.

3 THE FEATURE MODEL AND ITS CHARACTERISTICS

We needed a redundancy-free model that displayed each property and each property value only once. Familiar with feature modeling in the automotive industry, our team came up with the idea of *perceiving and modeling program properties and their values as features.* This chapter recounts how we constructed our special feature model.

3.1 Designing the Feature Tree

From the set of properties and values we designed a hierarchy of features. Because a feature configuration would be a retail loan, we called the root "KfW retail loan". The properties fell into six categories, namely, recipient, investment, finance, sales, risk and promotion. Each category became one subtree under the root. We have structured the features in each subtree hierarchically. The more general features are closer to the root, and the more specific features are closer to the leaves. We encoded as many dependencies as possible as tree constraints by carefully modeling the group types of the features (and, or, alternative). At that time, we detailed the features as far as necessary to differentiate between programs.

Many programs share a subset of the same properties, but the properties differ in values. For example, every program defines price categories according to loan terms and interest rate periods, but the periods differ (program A: 5-5 / 10-10 / 20-10 years, program B: 10-10 / 20-10 / 20-20 / 30-10 / 30-20 years, etc.).

As a feature modeling tool we have used FeatureIDE [12]. It was easy to learn, easy to handle, extremely useful from the start, and it has been convenient, fast, aesthetic and reliable ever since.

After one week, the feature tree presented about 60 % of the features of the KfW programs systematically and redundancy-free.

3.2 Modeling Dependencies

The feature tree spanned an oversized product space. We had to add cross-tree constraints (CTCs) in order to restrict the possible configurations to permissible loan configurations. Because we wanted to visualize and analyze the portfolio, the CTCs had to be complete and correct.

Dependencies between features originate from the domain, i.e., from laws, requirements given by ministries, or preferences asked by customers. We do not have to consider any technical constraints.

We were looking for CTCs among properties, but didn't find many. Only a few dependencies among properties are true for all programs, e.g., "loan term ≥ interest rate period". Many dependencies are true for a few programs only. The CTCs we could define did not rule out all configurations we do not sell as loans, for two reasons:

(1) Some constraints *require numerical expressions,* like comparison of values, which cannot easily be expressed in a feature model.
(2) Many constraints *depend on KfW retail programs, program variants* or certain *groups of programs.*

We solved the first problem by dividing the continuum in discrete, mutually exclusive, seamless intervals: Instead of modeling "50,000 EUR < loan amount <= 100,000 EUR", we introduced a feature "B_50_100T" in the feature tree (where T indicates thousand), and another feature representing the currency "EUR". If B_50T_100T is selected (and EUR is selected), the loan amount will lie between 50 thousand and 100 thousand EUR.

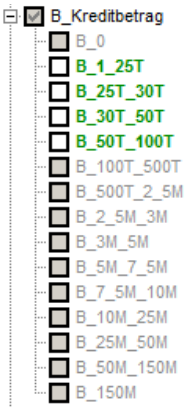

Figure 1

Figure 1 shows feature "loan amount" (B_Kreditbetrag). Each subfeature represents the left-open, right-closed interval of loan amounts defined by the feature name. If a certain configuration allows a maximum loan amount of 100 thousand EUR, the features B_1_25T, B_25T_30T, B_30T_50T and B_50T_100T will be undefined, and all other B_* features will be unselected. This is fine for us because we do not have to calculate figures from numeric features, and it allows us to use existing tooling. The interval boundaries stem from the program differences and might seem odd, but domain experts know these values well and have never questioned this mixed bag.

We solved the second problem by adding an exceptional subtree to our feature model, namely, the program subtree.

3.3 Creating the Program Subtree

We structured the set of KfW retail programs and organized the programs in a promotionally and logically worthwhile hierarchy:

- The root is called *program.*

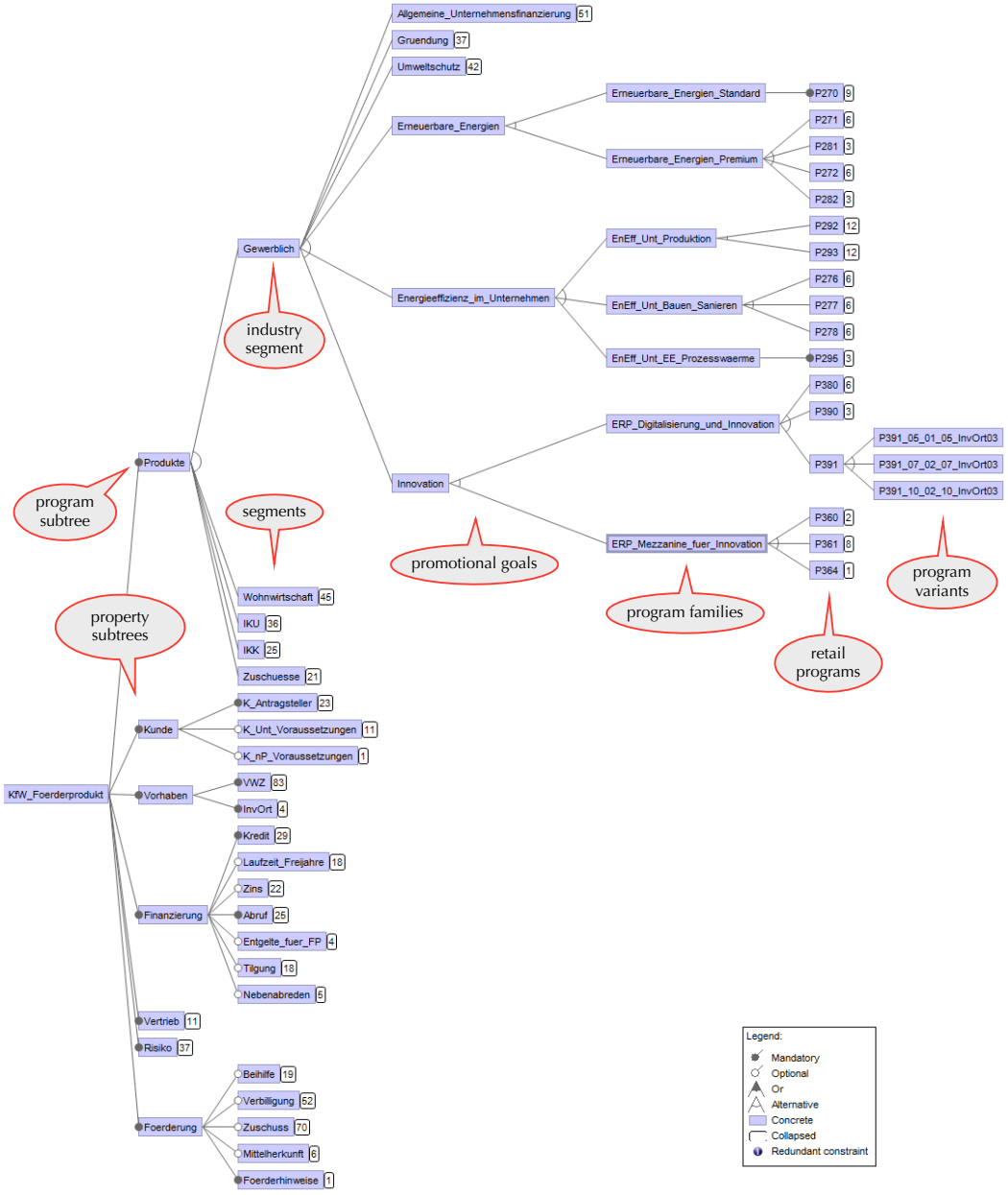

Figure 2: The feature tree with the program subtree

- Each KfW retail program falls in one of five *segments* (industry, housing, municipal companies, municipality, grants), so these five segments became children of the root.
- The industry segment is rather large and covers six promotional goals, while other segments cover only one or two. So a *promotional goals* level was introduced as children of segments.
- Many programs belong to a program family. To keep the tree balanced, every program was assigned a *program family* as parent, and these program families are children of promotional goals.
- Each KfW retail program is identified by a three-digit number. We prefixed the letter P because FeatureIDE did not accept feature names beginning with a number. Every program is a child of a program family.
- Each program has one or more program variants. A program variant is defined by a set of properties including loan term, grace period and interest rate period. *Program variants* are children of programs and are the leaves of the tree.

The resulting level five hierarchy consists of 5 segments, 13 promotional goals, 32 program families, 55 programs and 300 program

variants. All feature groups are of type "alternative" and all paths from the root to the leaves have the same length. We have put this program tree under the root "KfW retail loan". Figure 2 presents the feature tree, program subtree partly expanded, and other subtrees are collapsed.

Hence, *programs are also features,* their ancestors and children, too.

From hereon we distinguish between property features and program features. Any node of the program subtree is a program feature, while the nodes of all other subtrees are property features.

3.4 Defining Cross-Tree Constraints

Within the feature model, constraints restrict the possible combinations of features to permissible loan configurations.

As soon as the feature model contained the programs we could write cross-tree constraints in the form of

"P270 ⇒ not stateAid"

"renewables ⇒ quarterlyRepayment"

"industry ⇒ fixedInterestRate".

These CTCs connect program features with property features. Technically, the antecedent can contain features from any level of the program subtree. We always use the feature on the highest possible level to express that the implied properties are common to that group, which, by the way, keeps the constraints short.

El-Sharkawy et al. [4] report that the accurate definition of cross-tree constraints is not trivial to get right. In our case, three factors facilitate the definition of CTCs:

(1) Thanks to the program subtree we can express dependencies of properties on programs or groups of programs.
(2) FeatureIDE analyzes the feature model during editing (with the help of the integrated solver SAT4J) and informs instantly about possible errors (dead features, contradicting CTCs, redundant constraints).
(3) The configuration editor of FeatureIDE empowers feature modelers to test the CTCs they have written.

Step by step, we analyzed more programs and added features and constraints to our model. As of today, the model contains 55 active programs, 940 features, 700 thereof terminals, and 1,200 cross-tree constraints. 821 features appear in cross-tree constraints, yielding a cross-tree constraint ratio of 821 / 940 = 87.3 %, which is high, relates to the complexity of our portfolio, and is not atypical for real-world feature models [2]. FeatureIDE keeps running smoothly and with high performance.

3.5 Complete Configurations Define Loans, Partial Configurations Define Programs

Having defined the feature tree, its constraints and cross-tree constraints, FeatureIDE provides a configuration editor which is always in sync with the feature model. In the configuration editor we can configure loans. Each complete configuration defines a loan. FeatureIDE statistics count about 800,000 different loan configurations on our model.

Two virtually orthogonal aspects – programs and their properties – in one model offer (at least) two new possibilities in the configuration editor:

- By selecting a program feature the user can instantly see which properties are common to this program, not included in the program, or variable.
- By selecting a property feature the user can instantly see which programs include (or exclude) this property.

As an example, figure 3 shows on the left hand side the propagation of a selected program feature on the property subtrees, and on the right hand side the propagation of a selected property feature on the program subtree. The white filled and ticked checkbox denotes the manually selected feature. Grey filled checkboxes are unselected features. White filled checkboxes are open decisions, so-called undefined features.

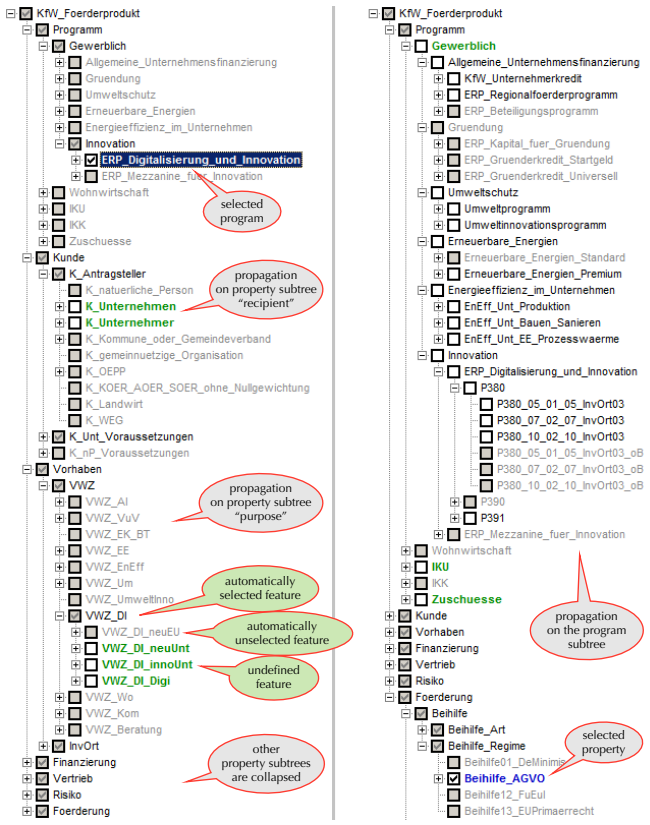

Figure 3: Left: Propagation of a selected program on property subtrees. Right: Propagation of a selected property on the program subtree.

We use these two techniques to

- test our feature model for correctness
- ensure completeness of the constraints.

Selecting a *program* in the configuration editor results in a *partial configuration* showing the common properties of this program as selected features, the non-properties as unselected features and the variation points of this program as undefined features. Partial configurations make sense because they define programs, and a program defines a certain class of loans.

Selecting a *property* unselects all programs that do not include this property and leaves all programs undefined that include this property. Continuing to select undefined features until the configuration is *complete* results in the configuration of a product, which is a valid KfW loan. The selected features represent its properties and their values. In the program subtree one terminal feature, i.e., a program variant, will be selected.

We have visualized the portfolio, and the CTCs are complete and correct. The feature model makes commonalities and variabilities visible, enabling us to compare the programs and to analyze the portfolio.

Our approach is new to KfW, it is still an experiment and our feature model is a prototype. We have a long way to go.

4 THE CONFIGURATOR AS AN ANALYSIS TOOL

The feature model visualizes portfolio complexity. This chapter covers how we use the configuration editor of FeatureIDE to analyze the complexity, and how we developed a configurator that runs in a web browser, in order to empower our colleagues, who do not want to install FeatureIDE, to analyze the portfolio.

4.1 Analyzing the Portfolio

For the first time we can see all features of all programs in one model and free of redundancy. The feature model enables us to configure loans and to experience how many configurations are possible. It exposes commonalities, variabilities and exceptions.

To analyze the portfolio we select a property feature in the configuration editor and observe the propagation on the program subtree. If the selection of a property feature results in

- all programs being selected, the feature is common to all programs and therefore a standard feature.
- many programs being selected, the feature is common to many programs.
- one program or very few programs being selected, the feature is an exception.

We have discovered dozens of features that made candidates for elimination or standardization. We put them on a list and asked domain experts where these exceptions came from and if they were dispensable. Our colleagues have been working on that list. Often it is not easy to come to a decision and put it into practice.

As a next step, we wanted to enable KfW domain experts outside of our team to do analyses. We also needed expert help to verify the feature model, especially the cross-tree constraints.

FeatureIDE is not a suitable tool for non-technical-affine people who

- shouldn't have to install a software development tool on their computers
- don't have access to our workspace and don't want to edit the feature model anyway
- don't want to risk changing the model involuntarily.

4.2 YAP: A Client-Server Product Configurator

We decided to implement a configurator on our feature model that would run in a web browser, so that our colleagues could configure a product without having to install and use FeatureIDE. We were very pleased about another highlight of FeatureIDE: its core functionality can be used standalone, without GUI [9]. Fortunately, this library is available for download at [3].

Our configurator should fulfill the following requirements:

(1) It should have the look-and-feel of FeatureIDE's configuration editor, and it should run in a web browser.
(2) A server component should store a given version of our feature model read-only.
(3) If the user selected a feature in the configurator, FeatureIDE should be called to propagate the selection on the model.
(4) The solution should allow many users to use the configurator simultaneously.

Closely cooperating with the FeatureIDE development team we developed a client-server-solution and called it YAP - Yet Another Product configurator:

A stateless *Feature Model Service* runs on the web server and provides a RESTful interface to the feature model and FeatureIDE. When a YAP client calls the Feature Model Service for the first time, it returns the feature model and the initial feature configuration, that is, the mandatory features are automatically selected, and all other features are undefined. YAP presents the initial feature configuration to the user. As soon as the user selects a feature, the client calls the Feature Model Service and passes the configuration including the user's selection. The Feature Model Service calls the FeatureIDE core, passes the configuration, requests propagation, receives a new configuration, and returns it to the client, which updates the presentation.

Figure 4 shows a screenshot of YAP. The left hand side displays the feature tree top-down, next to each feature a four-state checkbox indicating the state of the feature (manually selected, automatically selected, unselected or undefined). The more user-friendly feature names and tooltips that explain the features stem from feature attributes. The right hand side displays a tile for each program, the program name and number, the lowest and highest possible interest rate and an expansion panel of program variants. The programs and variants on the right hand side are always in sync with the user's configuration on the left hand side. All information seen in the YAP browser window is contained in the feature model, except for the interest rates which are calculated externally and read from a file. If the user enters a string in the search field on the upper right, the feature tree will shrink, only showing paths from the root to feature names that contain this string.

The immediate propagation of a selection surprises and inspires the YAP users. The fact that decision propagation prevents users from making contradictory decisions [10] keeps baffling our audiences. For the first time, domain experts can experience the complexity of the portfolio. As of today, YAP is still a prototype and is available for KfW employees only. They use it for analyses and as a source of information on KfW programs. For instance, if you want to find out which programs contain state aid, you can either ask an expert, browse 30 program information sheets, or click feature "state aid" in YAP and observe its propagation on the program subtree. YAP makes expert knowledge available to non-experts, and it makes commonalities and variabilities of our programs more visible, even for experts.

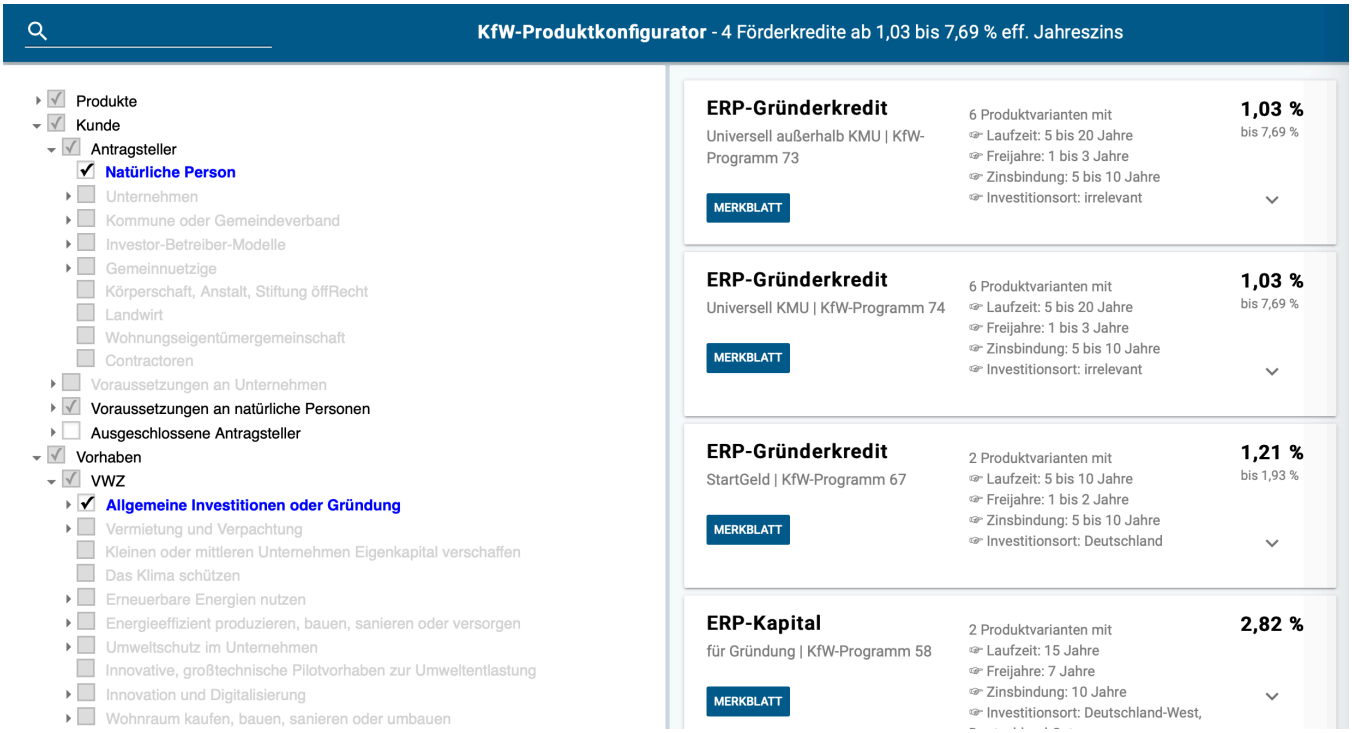

Figure 4: YAP, a configurator that runs in a web browser. The available programs and variants and their interest rates on the right hand side adjust to selected features on the left hand side. Recipient and intended purpose have been selected.

5 THE FEATURE MODEL AS A VEHICLE FOR STANDARDIZATION

We can analyze the portfolio and we are working on the reduction of complexity of the existing portfolio. But each time new programs are ordered and developed, new complexity can be introduced into the portfolio. We need to define a standard to prevent the growth of complexity.

Instead of "defining a standard" and ensuring that new developments comply, we provide tools that produce standard artifacts and relieve domain experts. Our number one standardization measure is to generate documents from the feature model and partial configurations. In this chapter we describe the development of our document generator and its introduction as a standardization tool.

5.1 Generating Program Information Sheets

Complexity stems from the variety of program-specific business logic. This business logic is reflected in the program information sheets. A program information sheet describes the properties of one KfW program: the promotional goal, who is eligible to apply, intended purposes, financial parameters, etc., plus possible dependencies between these properties, such as "if the intended purpose is *equipment*, the loan term will be limited to 5 years."

Program information sheets address people looking for a loan, on-lending banks, private individuals, borrowers and ministries.

Program information sheets are written and maintained manually as Microsoft Word® documents. Typically, a domain expert authors and maintains one information sheet. Information sheets share the KfW corporate design, and they should be uniform and consistent. Because different programs share common features, many information sheets contain information of the same kind. The authors have to keep layout, structure, names, descriptions and generic terms consistent across all information sheets manually. With ever growing complexity and steadily increasing demands to change the portfolio, this manual process is approaching its limits.

Features define KfW programs, and the description of a program consists of the descriptions of its features. Figure 5 shows how features map to a program information sheet. The order of the features corresponds to the structure of the document. High- or middle-level features correspond to headings. Low-level and terminal features correspond to text snippets in the paragraphs.

If we

- maintain all text snippets centrally, so that each text snippet exists only once, and
- generate program information sheets from our feature model and these text snippets,

we can enforce that

- program information sheets share the same layout, document structure and style,
- different program information sheets describe identical features exactly the same way,
- program information sheets include only modeled features,
- exceptions are excluded or at least recognized as such.

We could kill two birds with one stone:

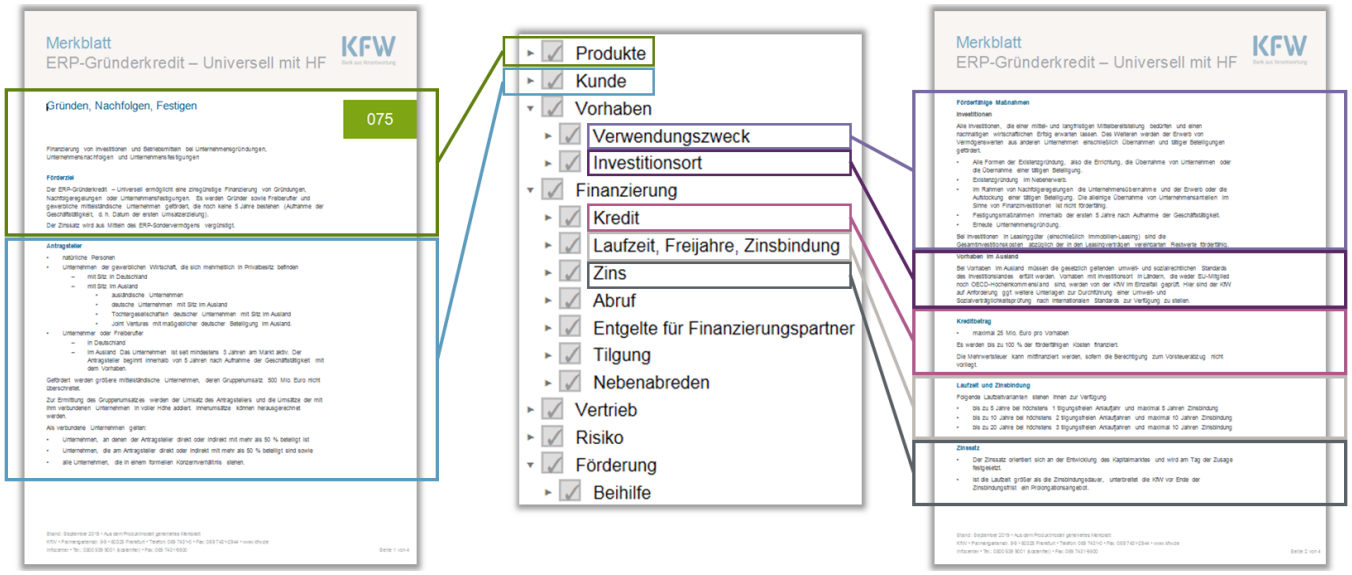

Figure 5: Mapping of features to a program information sheet

(1) Program information sheets would no longer have to be maintained and kept in sync manually.
(2) Generating program information sheets would be a means of standardization.

FeatureHouse, an Eclipse plug-in, can compose software from a feature model and a configuration, if suitable software artifacts are supplied with the features [1]. For each selected feature, FeatureHouse looks for a file with the same name as the feature. If such a file exists, FeatureHouse composes the software artifacts included in this file into a result file. This composition is called superimposition [12]. From the composer's point of view, it does not matter whether it composes software code or other text documents. It may as well compose program information sheets from text snippets attached to features in our model.

Normally, FeatureHouse requires a complete feature configuration as input to the composition of documents. It does process partial configurations, but ignores undefined features. Since the generation of program information sheets must consider undefined features, we needed another solution. Our document generator should fulfill the following requirements

(1) generate a properly formatted program information sheet from the feature model and the name of a program feature,
(2) generate the entire document, because no text snippet is common to *all* program information sheets,
(3) generate text from undefined features, too, because selecting a program leaves many decisions open,
(4) read short text snippets from the feature model instead from files, because we do not want to maintain hundreds of text files, one for each feature, especially since the descriptions of some features consist of only a few words.

We designed the solution as follows:

- We accept and embrace the fact that the generated document has the same structure as the feature tree, traversed in pre-order. This is a certain limitation, but it will keep the generator simple, and it makes a great standard.
- We analyze the nature of the partial configurations and develop appropriate capabilities of the composer.
- We store short text snippets in the feature model.
- We find it convenient to use Markdown [7] as mark-up and let Pandoc [11] transform Markdown into properly formatted Microsoft Word® documents.

Since release 3.5, FeatureIDE has supported feature attributes. These are additional properties of features, and they are stored in the feature model. We use attributes to store text snippets for features described by a short text, or to store the reference to a file that contains a longer text. Currently, our model contains 330 text snippets, 180 thereof in attributes, 150 in files. All text snippets are written in Markdown.

Selecting a program in the program subtree results in a partial configuration and leaves many features undefined, e.g., possible recipients, the size of the company and term variants. Our composer must process these undefined features. But how? We analyzed these partial configurations, saw that open decisions where always within small subtrees of the feature tree, and identified three types of open decisions that we called A, B and C:

A. Features of a subtree which need to be itemized.
B. Children of one feature define a range whose maximum and/or minimum value need to be generated.
C. Permissible combinations of features need to be identified and from these combinations a text needs to be composed.

Our composer basically concatenates the text snippets attached to selected features in the pre-order of the feature model, but also implements type A, B and C. That is, during document generation the composer resolves the decisions left open in the partial configuration. This includes repeatedly calling FeatureIDE and the SAT solver to determine permissible combinations of features. Our short

paper [5] elaborates on type A, B and C, and how the composer pieces together documents from partial configurations.

In a nutshell, the generator composes one Markdown document from the partial configuration and the text snippets. Pandoc transforms the Markdown document to a Microsoft Word® document. We provide a Word® docx template with KfW corporate design elements for Pandoc. Figure 6 shows the document generator architecture.

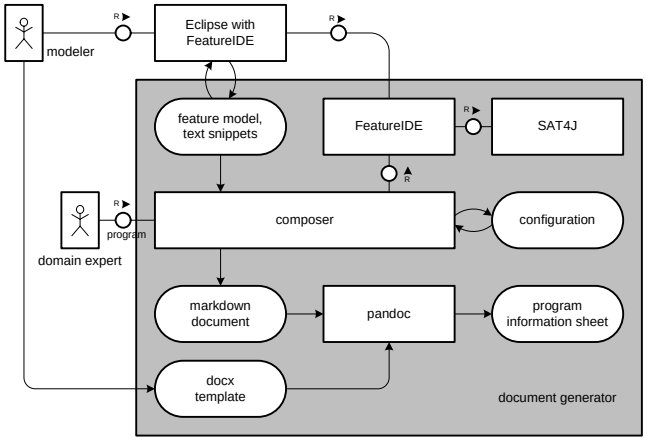

Figure 6: The document generator architecture (notation: FMC block diagram [8])

In order to establish YAP and the document generator as the KfW program standardization toolset, we have integrated the document generator in YAP. Each program tile provides a button Merkblatt (German for information sheet), which, when clicked, calls the document generator and passes the program number as argument. So domain experts and other colleagues can use the document generator whenever they want.

5.2 Standardization Toolset

By comparing the original program information sheets with the generated ones, we discovered that some original information sheets contain statements for which our feature model does not contain any features. Our team and the domain experts always work on these cases: Is a feature missing or is the statement superfluous? If a feature is missing, we will add it to the model, add its description, and generate the text. If the statement is superfluous, we will ask them to remove it from the information sheet, and thus, from the program. It can be tedious to remove long-established features. But we keep working on it, because it reduces complexity.

Newly developed retail programs should "follow the standard". Domain experts ask us, what "the standard" is. Instead of providing rules or guidelines, we answer: YAP and the document generator define the standard.

Using the document generator, we discover exceptions. Here is an example: In most KfW retail programs, the possible combinations of loan term, grace period and interest rate period are permitted for all loans that can be derived from this program. However, some programs exhibit an extra dependency: the permitted combinations differ with respect to the intended purpose. In these cases, the program information sheet must itemize the permitted combinations of loan term, grace period, interest rate period and intended purpose (4-feature-combination). In all other programs, the itemization does not include the intended purpose (3-feature-combination).

The document generator processes each configuration identically. It does not print a 4-feature-combination for some programs and a 3-feature-combination for others. We consider the 4-feature-combination to be an exception that should be abolished. We have proposed to remove the critical purpose from the affected programs, and to design a new program that promotes this purpose.

When a ministry orders *a new retail program,* we model this program experimentally in our feature model. Usually, the new program belongs to an existing promotional goal or program family, so we insert the new program as a descendant. If any ancestors of the new program feature appear in antecedents of cross-tree constraints, these constraints will define the standard for the new program. We add missing cross-tree constraints until they are complete. Then, we publish the feature model containing the new program as a separate instance of YAP.

We suggest that existing features be used, new features only be introduced if unavoidable, and only existing dependencies be used. Until now the difficulty was that neither KfW domain experts nor ministries knew all existing properties, values and dependencies. Now YAP and the document generator make them visible.

Afterwards, we generate the program information sheet. The document generator and the re-usable text snippets render the description of the new program and save domain experts from writing (or cloning) standard paragraphs. We add missing text snippets of new features such as the intended purpose. The document generator ensures that each information sheet describes common features in the exact same words, and that every information sheet has the same layout, structure, font, wording and style. If you generate, you will not need guidelines and you will not have to ensure that they are followed.

As a result, domain experts can develop new programs that are consistent and conform to the existing portfolio even if they don't know the existing properties and values. This is how YAP and the document generator prevent the growth of complexity.

6 CONCLUSIONS AND FUTURE WORK

Our feature model helps to standardize the programs, to reduce manual activity and to avoid redundancy. It will excel in applications that need dynamic and flawless configuration of products. Our ideas and plans include

- to shorten time-to-market for new products by giving ministries access to the feature model
- to improve KfW's program information by generating reader-specific documentation
- to improve access to KfW's programs by offering product configurators
- to support KfW's interest rates pricing and publishing system by providing price-setting features
- to support checking loan applications
- to scope the product line

The following subsections give some details.

6.1 Giving Ministries Access to Feature Model

A product line can only be efficiently maintained if customers and orderers are satisfied with standard features and dependencies. In our case, the requirements imposed by the ministries are crucial. In a next step we want to offer ministries who request KfW programs to use YAP and the document generator. We will try to convince them to accept the standard and to work with us to shorten time-to-market and to reduce development costs.

6.2 Generating Reader-Specific Documentation

Program information sheets are multi-purpose documents that have to satisfy many readers: bank advisors, potential borrowers, loan recipients and ministries. On the one hand, this all-in-one solution minimizes the number of manually maintained documents. On the other hand, it is suboptimal for any reader.

Firstly, if we generate documents, we can generate reader-specific documents for each program with little more effort from the same source, our feature model: A program description including expert knowledge for bank advisors, a one-page showcase for our web site, an individual loan information sheet from a complete configuration for loan recipients and a program definition for ministries. The documents will be consistent and uniform nonetheless.

Secondly, we can generate product descriptions from complete configurations. This way, customers will receive mass-customized retail loan documents that describe the loan they have configured even before they go to their bank and apply for a loan.

6.3 Providing New Access Paths to KfW Loans

6.3.1 Finding a KfW Loan – Status Quo. Because KfW loans are particularly favorable, they are very competitive and in demand. But portfolio complexity challenges customers, too.

If you are looking for a KfW loan, you may browse the KfW web site, use the program finder, and answer a few questions. The result is a list of 3 - 5 links to possible programs. Following each link, you will find a short, structured web page on each program with a link to the program information sheet. Then you will have to decide which program will suit you best. This will be difficult, because you have to read and understand the promotional directives in the information sheets and their appendices.

Or you go to your bank. Most of our loans are on-lent loans, brokered by credit unions, savings banks and private banks. Expert bank advisors for promotional loans know KfW programs very well and help their customers.

In order to find the program that will suit you best and to get an offer, you or your bank advisor need *expert knowledge.*

6.3.2 Suggesting a New Way. Each complete feature configuration defines a KfW loan. Until now, the configurator YAP is available within KfW only. We plan to publish YAP so that anybody can configure KfW loans. The advantages of YAP over the existing program finder are:

- You can choose the loan features you want in any order that you like. Start with the type of recipient you are, the type of investment you make or the amount of money you need.
- You can't make mistakes in the decision process, thanks to the underlying propagation mechanism.
- You can see the interest rate you will have to pay.
- You will always get a result – your best option – instead of several possibilities.

After having configured your loan, you can generate your individual loan information sheet on the spot. It will describe exactly the loan features you have selected. Furthermore, we plan to generate the exact list of data and documents you need to bring to your bank for a loan application.

6.3.3 Integrating KfW Programs in Promotion Portals and Manufacturer Web Sites. Today, private individuals do not have to go to their bank to find a favorable loan. They rather browse the internet where

- independent portals broker loans
- manufacturers of goods eligible for promotion (heatings, fuel cells, windows, etc.) broker KfW loans and grants as a powerful sales argument.

We have implemented a programming interface to our feature model: The *KfW open API* provides access to the RESTful interface of the Feature Model Service (see 4.2), so a third-party software can browse programs or properties, make selections and configure loans. At this time, KfW open API is still a prototype and only available for registered beta users. One of the leading international manufacturers of heating systems uses the KfW open API to check if a potential customer is eligible for a KfW grant. As soon as KfW open API fulfills all regulatory demands, we are going to make it available for everybody.

6.4 Scoping the Product Line

KfW retail programs have always been well-managed. Now we want to manage the portfolio, the properties and the dependencies. The organization needs to move towards modeling the portfolio rather than developing single programs. How can we do that?

As mentioned before, the document generator uncovers all the specialties of a retail program. We have observed that the complexity or extraordinariness of an information sheet correlates with the complexity of the software required to process loans derived from this program. The document generator also detects if a program deviates just too much from the others so that the information sheet cannot be generated in full with reasonable effort. From a product line point of view, such a program is out of scope.

The feature model and the document generator provide a means to demonstrate the deviations and to discuss them among domain experts, with the ministries and with the IT department. Implementation costs become obvious as soon as retail program design starts, long before software solutions are developed. The organization can make an informed decision whether new properties or dependencies should be included in the portfolio or not. This is the first step of the transition from single program development to product line development.

REFERENCES

[1] Sven Apel, Christian Kästner, and Jörg Liebig. 2020. *FeatureHouse: Language-Independent, Automated Software Composition.* Retrieved March 5, 2020 from http://www.fosd.net/fh

[2] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems

Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[3] Feature IDE development team. 2020. *FeatureIDE - An extensible framework for feature-oriented software development.* Retrieved March 5, 2020 from https://featureide.github.io

[4] Sascha El-Sharkawy, Saura Jyoti Dhar, Adam Krafczyk, Slawomir Duszynski, Tobias Beichter, and Klaus Schmid. 2018. Reverse engineering variability in an industrial product line: observations and lessons learned. In *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai (Eds.). ACM, 215–225. https://doi.org/10.1145/3233027.3233047

[5] Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. *Generating Documents from Partial Configurations.* Retrieved September 7, 2020 from https://esb-dev.github.io/mat/KfW_DocGen_abc.pdf

[6] Claudia Fritsch and Ralf Hahn. 2004. Product Line Potential Analysis. In *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3154)*, Robert L. Nord (Ed.). Springer, 228–237. https://doi.org/10.1007/978-3-540-28630-1_14

[7] John Gruber. 2020. *Markdown.* Retrieved March 5, 2020 from https://daringfireball.net/projects/markdown/

[8] Andreas Knöpfel, Bernhard Gröne, and Peter Tabeling. 2006. *Fundamental Modeling Concepts: Effective Communication of IT Systems.* Wiley, Chichester, UK.

[9] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017*, Maurice H. ter Beek, Walter Cazzola, Oscar Díaz, Marcello La Rosa, Roberto E. Lopez-Herrejon, Thomas Thüm, Javier Troya, Antonio Ruiz Cortés, and David Benavides (Eds.). ACM, 42–45. https://doi.org/10.1145/3109729.3109751

[10] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating configuration decisions with modal implication graphs. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 898–909. https://doi.org/10.1145/3180155.3180159

[11] John MacFarlane. 2019. *Pandoc.* Retrieved March 5, 2020 from https://pandoc.org

[12] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer. https://doi.org/10.1007/978-3-319-61443-4