Scalable Prediction of Non-functional Properties in Software Product Lines

Norbert Siegmund, Marko Rosenmüller Christian Kästner, Paolo G. Giarrusso Sven Apel, Sergiy S. Kolesnikov
University of Magdeburg, Germany Phillips University Marburg, Germany University of Passau, Germany

***Abstract*—A software product line is a family of related software products, typically, generated from a set of common assets. Users can select features to derive a product that fulfills their needs. Often, users expect a product to have specific non-functional properties, such as a small footprint or a minimum response time. Because a product line can contain millions of products, it is usually not feasible to generate and measure non-functional properties for each possible product of a product line. Hence, we propose an approach to *predict* a product's non-functional properties, based on the product's feature selection. To this end, we generate and measure a small set of products, and by comparing the measurements, we approximate each feature's non-functional properties. By aggregating the approximations of selected features, we predict the product's properties. Our technique is independent of the implementation approach and language. We show how already little domain knowledge can improve predictions and discuss trade-offs regarding accuracy and the required number of measurements. Although our approach is in general applicable for quantifiable non-functional properties, we evaluate it for the non-functional property *footprint*. With nine case studies, we demonstrate that our approach usually predicts the footprint with an accuracy of 98 % and an accuracy of over 99 % if feature interactions are known.**

I. Introduction

A *software product line (SPL)* is a family of related software products sharing a common set of assets [1]. Differences and commonalities between products are typically described in terms of features [2]. Users can specify a product as a selection of features that satisfies their functional requirements [3]. With many contemporary implementation mechanisms, it is possible to automatically *generate* products based on feature selections.

In addition to functional requirements, users are also interested in *non-functional properties* of a product, such as performance, footprint, and reliability. Non-functional properties are especially important in the domain of embedded systems [4], [5]. Reducing the resource consumption of a software product can enable the use of cheaper hardware devices or extend battery live, which can save a significant amount of money in mass production. In a product-line setting, a stakeholder may wish to enforce *constraints* on non-functional properties (e.g., the footprint may not exceed the capacity of an embedded device) or select the product that is best according to some property (e.g., the fastest product). This, however, means we have to measure many products, until we find a feature combination that satisfies certain non-functional requirements.

To determining the feature set that *optimizes* a certain non-functional property, we may have to measure *all* products. Even small SPLs with less than hundred features can already have millions of products, and industrial-size SPLs can contain thousands of features [6], [7], [8]. So, generating products for all feature selections is simply not feasible in practice. Consequently, we investigate alternatives based on heuristics that approximate non-functional properties without generating and measuring all products.

We want to *predict* non-functional properties of an SPL's products, without actually generating a product, by aggregating the non-functional properties of selected features. Hence, we have to measure the non-functional properties *per feature*. To this end, we compute a small but suitable set of products, by analyzing the relationships between features documented in a feature model [2]. We actually compile and measure these products and approximate values per feature from deltas between these products. That is, we try to *predict* what influence the selection of a single feature will have on the non-functional property of the final product.

Of course, the predictions will not be exact, because the selection of one feature may influence non-functional properties of other features (for example, a feature "global compiler optimization" and will affect non-functional properties of most other features). Additionally, there may be a complex mapping from features to implementation units that can lead to false approximations. To account for feature interaction, we have a model in which we can define known feature interactions. We measure the influence of these defined interactions to improve the accuracy of our predictions.

We will show that for a product line with n features, already $n + 1$ measurements can lead to acceptable predictions of footprint, which can be improved further with more measurements. Especially when using domain knowledge about feature interactions, we can significantly improve accuracy with only few additional measurements. Even, measuring n^2 products, which usually yields very accurate results in our experience, still requires significant less effort than measuring all possible products (which is 2^n in the worst case).

We argue that our approach is general and can be applied to different quantifiable non-functional properties. To evaluate our approach, we choose a non-functional properties for which a measurement is reproducible and is not under subject of measurement bias. Hence, we use the non-functional property *footprint* in our evaluation. We predict the footprint of products in nine case studies. To demonstrate that our prediction model is actually useful in practice, we cover very different aspects in a broad spectrum of scenarios: we selected SPLs of different sizes (2 500 to 13 million lines of code, 5 to 100 features), languages (C, C++, and Java), varying implementation techniques, from different domains (e.g., operating systems, database engines, end-user applications), and from different developers (both academic and industrial). With a linear number of measurements (which means, without considering feature interactions), our footprint predictions have an accuracy of 99 % of the actual measurements. However, if complex mappings between features and implementation units exist, the approach is not sufficient. By exploiting domain knowledge about feature interactions, predictions improve to 99.7 % for all SPLs.

II. Problem Statement

Non-functional properties are diverse, and it is not obvious how we can interpret and handle the measured values. We concentrate on properties that can be quantified (i.e., that are measurable). The theory of measurement provides different levels (e.g., nominal, ordinal, interval, and ratio) of how measured values can be interpreted [9]. In our approach, we support *interval and ratio-scale*-based measures, because the values of two measurements reflect differences of the according property.[1]

To measure the vast majority of non-functional properties, we have to actually generate and execute a product [10]. Our idea is to identify the *influence of an individual feature* on the product's non-functional properties. However, it is not clear which feature of a product contributes in which quantity to a product's properties. Even worse, non-functional properties of a feature may depend on the presence of other features, such that correlations between measured values and corresponding features are ambiguous. Hence, determining an *exact value* of a non-functional property per feature is not always possible [11].

Problems of approximating a feature's non-functional properties are mostly caused by interactions between features. Two types of feature combinations can cause a feature interaction: (a) features *A and B* are present in a product and (b) features *A or B*. For footprint this means, that we have to include additional code in a product in the first case, whereas a set of features share the same code in the second case.

To illustrate the problems of feature interactions, we use a simple example that already exhibits measurement problems. In Figure 1, we show the C++ implementation of a linked list with two features: *PrintList* and *PrintElement*. Features are implemented with conditional compilation. To measure a non-functional property (e.g., footprint measured as the binary size of the compiled product) per feature, we first measure each individual feature. Hence, we measure the footprint of Lines 5 and 6 as well as Line 11 for feature *PrintList*. We would not measure Lines 8 and 9, because these lines are compiled only for a product that contains both features *PrintList* and *PrintElement*. Hence, if we would predict the footprint of a product that includes both features, the prediction would be inaccurate. To predict the footprint correctly, we would have to measure the influence of the feature interaction (Lines 8-9).

As another example, consider a set of features that use the same resource. A shared resource may be an external library or otherwise shared code. We can easily extend the list SPL example to show the problems of shared resources. We may use an external library to log the elements of a list instead of printing them. To this end, we change the call to *cout* and method *print* (in Lines 6 and 15 of Figure 1) to use the external logging library. The library has a considerably larger binary size than the features itself. When approximating a feature's non-functional properties, the predominant part of the footprint would stem from the logging library. Because we measure the size of the library for both features, we would predict the size of a product with both features incorrect. The reason is that both features share the same library, which is only included once in the product, but was measured twice (once for each feature).

```
1  class List {
2    int numberOfElements;
3    Element* head;
4  #ifdef PrintList
5    void printList() {
6      cout << numberOfElements;
7  #ifdef PrintElement
8      if(this->head != NULL)
9        printElement(this->head);
10 #endif
11   }
12 #endif
13 #ifdef PrintElement
14   void printElement(Element* node){
15     node->print();
16     if(node->hasNext())
17       this->printElement(node->getNext());}
18 #endif
19 };
```

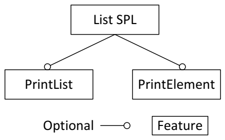


Fig. 1. C++ code of a list SPL with two features: *PrintList* and *PrintElement*. We show the feature model in the upper right corner.

III. Non-functional Properties of Features

In this section, we present our approach to approximate the influence of a feature (or a feature interaction) in a product with respect to non-functional properties. First, we describe the general concept of our approach. Second, we explain algorithms necessary to extract the approximations of each feature's non-functional properties from a set of products.

We define $P\{F_1,...,F_n\}$ as the product P consisting of the features $F_1,...,F_n$, *V(P)* as the measured non-functional property of product *P*, and V_F as the approximation of the non-functional property of feature *F*. Furthermore, we specify the index of a product *P* to indicate for which feature the product is generated and measured. For example, product $P_{F_1}\{F_1\}$ describes the product to measure feature F_1.

For simplicity, we omit mandatory features in P{..} and only show the features important for the approximation. In addition, *parent*(F_1) is the direct parent feature of feature F_1 in the feature model.

A. Process

The general idea of approximating a feature's non-functional properties is to measure the delta between two products that differ only in the presence or absence of this feature. We

[1] For some non-functional properties, we may consider only ratio-scale-based measures, because we may need to reason about approximations.

interpret the delta of two products as the approximation of the added (or removed) feature's non-functional properties.[2]

Let us assume we measure two products of the list SPL that differ only in the presence of feature *PrintList*: $P_{base}\{base\}$ and $P_{PrintList}\{base{,}PrintList\}$.[3] We can approximate the influence of feature *PrintList* as the delta between products P_{base} and $P_{PrintList}$: $V_{PrintList}$ = $V(P_{PrintList})$ - $V(P_{base})$.

In Figure 2, we illustrate the approach of approximating a feature's non-functional properties for SPLs that support an automated product generation. Using a feature model, we determine a small, but suitable set of products. In an *automated process*, we generate and measure each product of this set. Based on these measurements, we compute the delta between two products and use this value as the approximation for a feature. This means, to enable for each feature the computation of the delta between two products, we would need *2*n* measurements altogether. By using the values of previous measurements, we can reduce the number of required measurements to *n+1*.

Furthermore, we allow stakeholders to define feature interactions in the feature model [12]. Still in an automated process, for each feature interaction we add a single product to the set of products and measure its influence. That is, we *scale* the number of measurements to improve the quality of the prediction. We compute the actual influence of a feature interaction by using the delta between non-functional properties of the actually measured product and *predicted* non-functional properties of the same product. For example, if we define a feature interaction I_1 between features *PrintList* and *PrintElement*, we might predict $\text{V}_{predicted}(P_{F_1,F_2})$ = 220 KB, whereas the actual footprint is $\text{V}_{actual}(P_{F_1,F_2})$ = 200 KB. Hence, we would assign -20 KB to the interaction I_1 as the approximation for footprint.

We developed the tool *SPL Conquerer*[4] to implement the process of determining and measuring products and to compute a feature's non-functional properties. The application of SPL Conquerer provides two major benefits. First, SPL Conquerer realizes an automated measurement and approximation process that does not require any user interaction (e.g., the measurement process can run over night without monitoring). Second, based on the results of the automated measurement and approximation process, SPL Conquerer can predict a product's non-functional properties almost instantly.

B. Measuring Feature Interactions

As we explained previously, feature interactions may affect the approximation of a feature's non-functional properties. That is, a feature has different non-functional properties depending on the selection of other features and thus can cause inaccurate predictions. If feature interactions are not known or should not be taken into account, *a pure feature-wise measurement approach* is used. That is, we generate a product per feature and interpret deltas between products as the approximations of a feature's non-functional property. Unfortunately, this is sometimes not sufficient, for example, if there is a complex mapping between features and implementation units. In this case, a feature's approximation would take into account non-functional properties of several implementation units that however are also related to other features. Hence, a single approximation for a feature is not sufficient.

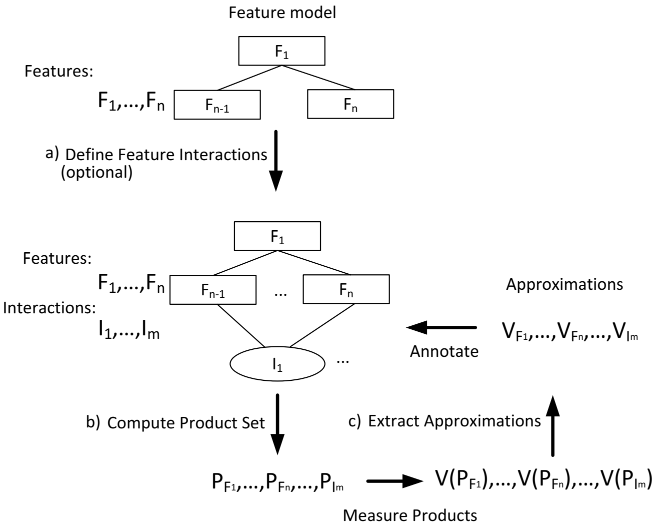


Fig. 2. The process to compute approximations of non-functional properties for features and feature interactions.

Fortunately, it is often an easy task to identify such feature interactions. We can use three different sources to identify feature interactions by: (a) using the mapping between domain features and implementation units, (b) analyzing the source code (e.g., searching for nested #ifdefs as a common indication of simple implementation interactions), and (c) using expert knowledge. In our evaluation (cf. Section IV), we use our knowledge of the mapping from domain features to implementation units for the Violet SPL, we analyze the source code of Berkeley DB, Prevayler, and SQLite, and we ask a domain expert for the Linux kernel to identify feature interactions. In the case no domain knowledge is available, it can be worthwhile to simply assume the existence of a feature interaction between each pair of features in an SPL. A similar approach was used to generate test cases for SPLs [13]. In the following, we give a short description about the different approaches including their complexity with respect to the number of measurements.

Feature-wise measurement: We generate a product for each feature. The complexity is *O(n)*, in which *n* is the number of features of an SPL. This approach should be used for very large SPLs and is most accurate if there is a one-to-one mapping between features and implementation units as in feature-oriented programming [14]. Still, in other cases, the results of the measurement are useful if we require only rough predictions for a product's non-functional properties (e.g., if a stakeholder is only interested in a qualitative comparison of non-functional properties from a set of desired products).

Interaction-wise measurement: In this approach, we measure not only each feature in the feature model, but also all *known* feature interactions. *SPL Conqueror* creates a product

[2] Such an approach requires interval or ratio scales [9].

[3] Feature *base* represents the code that is present in every product.

[4] http://fosd.de/SPLConqueror

that contains the features that belong to an interaction. This way, we have to measure *O(n+k)* products, in which *n* is the number of features and *k* is the number of defined interactions. If *k=0*, the interaction-wise approach is identical to the feature-wise approach. Measuring all interactions improves the accuracy in the prediction of a product's non-functional properties. Especially, when an SPL contains a large number of interactions and domain knowledge helps to identify many of them, this approach results in a solid prediction base.

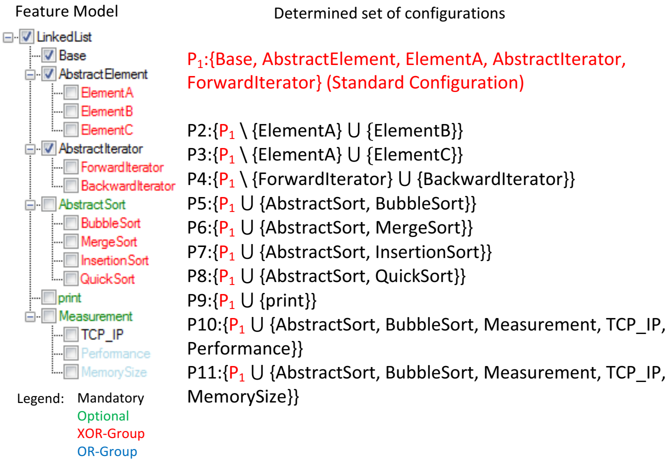
Pair-wise measurement: To automatically detect all first-order feature interactions (i.e., an interaction in which exactly two features participate), we use a pair-wise measurement approach. That is, we measure $((n*(n-1))/2)+n$ products. This results in a substantially increased product set to measure compared to the feature-wise measurement. Furthermore, there are also *n-wise measurements* possible to measure feature interactions in which exactly *n* features participate (n-th order feature interaction). However, they usually require a large number of additional measurements.

C. Computing the Product Set for Measurement

Our approach uses a specific set of measured products from which we can extract an approximation for each feature. An important goal is to minimize the number of products in the product set. That is, we aim at measuring only a linear number of products with respect to the number of features in an SPL. To reach this goal, we utilize the hierarchical structure of feature models that allows us to reuse products already defined for the parent feature. Hence, we need only *n+1* products instead of *2*n* products. In detail, every feature model has a root feature (which can be empty) and each feature in a feature model has either a parent feature or the parent is the root feature. Beginning with the root feature, we traverse the feature tree and add for each feature *a single* product to the product set that contains the current feature as well as the feature set of the parent's product. Hence, each newly determined product can use the previously defined product to compute the delta in its non-functional properties, which is the approximation of the non-functional properties of the differing feature.

Unfortunately, there are also constraints that may make a measurement for the given feature ambiguous, because we have to add sometimes multiple features at the same time due to constraints. Hence, the constraints in a feature model affect the product set as we will explain in Section III-D.

Beside constraints of a feature model, there are two other factors that affect the product set. First, to measure the influence of a feature interaction, we have to measure a product that contains the features that cause the interaction. To this end, we add an additional product to the product set with all features part of the interaction relationship. Second, we need an initial product (or feature set) to have a starting point – an *initial value* – from which we can compute the deltas of non-functional properties. This *initial feature set* is necessary if the individual selection of the root feature results not in a valid product. Then, we handle the set of features in the initial product as the "root feature" and traverse the feature tree starting from the features of the initial feature set. Hence, from this *initial value*, we add or subtract the approximations of each feature in a product's feature set to predict non-functional properties of the product.



P1:{Base, AbstractElement, ElementA, AbstractIterator, ForwardIterator} (Standard Configuration)

P2:{P1 \ {ElementA} U {ElementB}}
P3:{P1 \ {ElementA} U {ElementC}}
P4:{P1 \ {ForwardIterator} U {BackwardIterator}}
P5:{P1 U {AbstractSort, BubbleSort}}
P6:{P1 U {AbstractSort, MergeSort}}
P7:{P1 U {AbstractSort, InsertionSort}}
P8:{P1 U {AbstractSort, QuickSort}}
P9:{P1 U {print}}
P10:{P1 U {AbstractSort, BubbleSort, Measurement, TCP_IP, Performance}}
P11:{P1 U {AbstractSort, BubbleSort, Measurement, TCP_IP, MemorySize}}

Fig. 3. Computing the set of products to measure all features.

In Figure 3, we show an example of the computation of the product set of an extended example of a list SPL. On the left side, we show a feature model of a linked list SPL.[5] In this example, we define the initial feature set as product $P_1\{$Base, AbstractElement, ElementA, AbstractIterator, ForwardIterator$\}$. This product represents the initial feature set with the minimal number of features from which we add further features. For example, we add feature *print* to the initial feature set in $P_9\{$SC,print$\}$ to measure the approximation of this feature. As you can see, there are less products than features. This is caused by the fact that we cannot create individual products for mandatory features (e.g., features *Measurement* and *TCP_IP* must always be selected in combination).

D. Approximating Non-functional Properties per Feature

In the following, we describe our approach to compute approximations of a feature's non-functional properties for the most common relationships between features [2], [3]. In Figure 4, we present on the left side a relationship of a feature model (e.g., the mandatory relationship at the top). In the middle, we show a graphical representation of the product that has to be measured. We distinguish products with different colors and hatchings. Finally, on the right side, we state the feature selection of the corresponding products. Next, we define the equations how we extract approximations of non-functional properties for each feature.

(1) Mandatory: A mandatory relationship enforces that whenever the parent feature is selected (feature *A* in Figure 4), we must also select its child feature (feature *B*). As a consequence, we cannot measure feature *B* without *A* and measure the actual value of feature *B* always together with feature *A*. Hence, we decided to set the value of feature *B* to zero and the value of feature *A* to the sum of both features.

[5] SPL Conqueror visualizes mandatory features with black names, optional with green names, alternative features with red names, and OR-group features with light blue names.

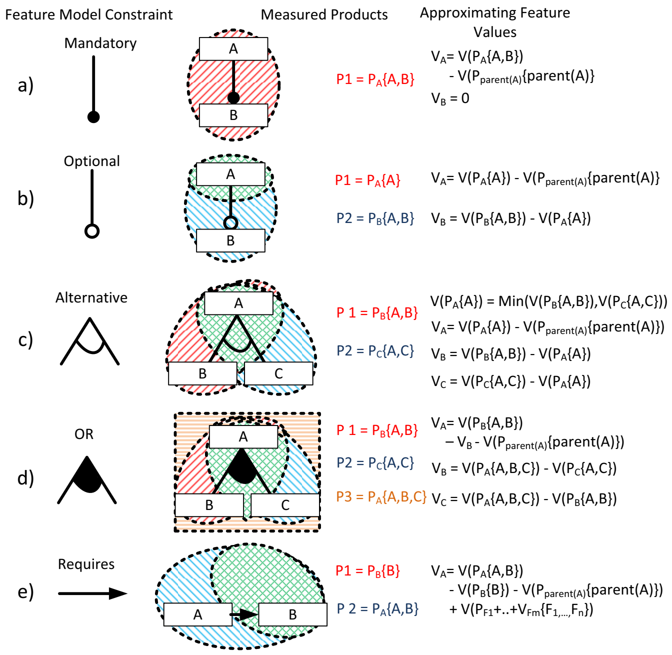

Feature Model Constraint
Measured Products
Approximating Feature Values
Mandatory
a)
A
B
P1 = P_A{A,B}
V_A= V(P_A{A,B}) - V($P_{parent(A)}${parent(A)}
V_B = 0
Optional
b)
A
B
P1 = P_A{A}
V_A= V(P_A{A}) - V($P_{parent(A)}${parent(A)}
P2 = P_B{A,B}
V_B = V(P_B{A,B}) - V(P_A{A})
Alternative
c)
A
B
C
P 1 = P_B{A,B}
P2 = P_C{A,C}
V(P_A{A}) = Min(V(P_B{A,B}),V(P_C{A,C}))
V_A= V(P_A{A}) - V($P_{parent(A)}${parent(A)})
V_B = V(P_B{A,B}) - V(P_A{A})
V_C = V(P_C{A,C}) - V(P_A{A})
OR
d)
A
B
C
P 1 = P_B{A,B}
P2 = P_C{A,C}
P3 = P_A{A,B,C}
V_A= V(P_B{A,B}) – V_B - V($P_{parent(A)}${parent(A)})
V_B = V(P_A{A,B,C}) - V(P_C{A,C})
V_C = V(P_A{A,B,C}) - V(P_B{A,B})
Requires
e)
A
B
P1 = P_B{B}
P 2 = P_A{A,B}
V_A= V(P_A{A,B}) - V(P_B{B}) - V($P_{parent(A)}${parent(A)}) + V(P_{F1}+..+V_{Fm}{$F_{1,...,}F_n$})

Fig. 4. Extracting feature's non-functional properties from automatically generated products depending on feature model constraints.

Thus, when a stakeholder selects feature A during product configuration, we show already the aggregated value of feature A and B. This way, it is easier to see the implications of a feature selection for a stakeholder, because she usually selects features starting from the root node. We compute the value for a mandatory relationship as follows (Figure 4a)):

$$\begin{aligned} V_A &= V(P_A\{A,B\}) - V(P_{parent(A)}\{parent(A)\}) \\ V_B &= 0 \end{aligned}$$

It is important to note that $V(P_{parent(A)}\{parent(A)\})$ is always available, because we begin with the initial feature set and an initial value and go with each step deeper into the feature model hierarchy. Hence, either we have already measured $V(P_{parent(A)}(parent(A)))$ or $P_{parent(A)}(parent(A))$ is the initial product.

(2) Optional: In an optional relationship, it is not required to select the child feature. Hence, we generate a product containing only the parent feature of the relationship: $P_A\{A\}$.[6] Additionally, we need a second product with feature B: $P_B\{A,B\}$. In this case, the computation of V_A and V_B is easy:

$$\begin{aligned} V_A &= V(P_A\{A\}) - V(P_{parent(A)}\{parent(A)\}) \\ V_B &= V(P_B\{A,B\}) - V(P_A\{A\}) \end{aligned}$$

(3) Alternative: An alternative relationship is more difficult, because we cannot select the parent feature of the relationship alone, but measure its value always in combination with different child features (e.g, $P_B\{A,B\}$ and $P_C\{A,C\}$). As a design decision, we set the approximation for feature A depending on the non-functional property either as the minimum or as the maximum value of all measured products of the alternative relationship. Additionally, we set the feature in the alternative relationship with the minimal or maximal measured value to zero. It depends on the property and the initial value if we either subtract (using the maximum) or add (using the minimum) a feature's non-functional properties. The reason behind this decision is that we cannot determine the value of the parent feature and would have to set this value to zero. We argue that users can make earlier decisions of selecting the parent of an alternative group if they retrieve the information that at least a certain value has to be added instead of zero. However, our design decision is not necessarily required and does not affect other computations. In Figure 4c, we can see that we require a generated product per feature in the alternative relationship. In our example, we require two products $P_B\{A,B\}$ and $P_C\{A,C\}$. To compute the value of feature A, as always, we need the measured product of the parent of feature A: $P_{parent(A)}\{parent(A)\}$. Then, we use the following equations:

$$\begin{aligned} V(P\{A\}) &= Min(V(P_B\{A,B\}), V(P_C\{A,C\})) \\ V_A &= V(P_A\{A\}) - V(P_{parent(A)}\{parent(A)\}) \\ V_B &= V(P_B\{A,B\}) - V(P_A\{A\}) \\ V_C &= V(P_C\{A,C\}) - V(P_A\{A\}) \end{aligned}$$

(4) OR: In contrast to the alternative relationship, we can select multiple child features. This raises the problem that we need to determine the influence of the parent feature of the relationship. If we do not compute V_A, we would aggregate the approximation of feature A multiple times for a product's prediction that contains multiple child features. Hence, we have to determine the approximation of feature A in an OR relationship rather than using the minimal measured product. To retrieve the value of feature A, we need an additional measurement for the OR relationship. In this measurement, we create a product that contains two features of the OR group (P_A\{A,B,C\} in Figure 4d). With this new measurement, we are able to extract the value of feature A using the following equations:

$$\begin{aligned} V_A &= V(P_B\{A,B\}) - V_B - V(P_{parent(A)}\{parent(A)\}) \\ V_B &= V(P_A\{A,B,C\}) - V(P_C\{A,C\}) \\ V_C &= V(P_A\{A,B,C\}) - V(P_B\{A,B\}) \end{aligned}$$

(5) Requires: Finally, we also consider cross-tree constraints in the feature model. The excludes constraint does not change the computation of a feature's non-functional properties, because it only restricts the number of features and we already try to measure a product with minimal number of features. In contrast, the requires constraint prohibits the measurement of a single feature. Our approach is to measure first the product that includes the target of the requires constraint $P_B\{B\}$ (product 1 in Figure 4e)). Then, we measure the product that includes both features $P_A\{A,B\}$. The problem of a cross-tree constraint is that we can have an overlapping set of features. That is, we may have features in both products $P_B\{B\}$ and $P_A\{A,B\}$ simultaneously. Thus, we must identify the overlapping features first and retrieve their values for the product containing these

[6] Of course, we have to include all necessary features to derive a valid product, e.g., all parent features.

features: $P_{F_1,...,F_n}\{F_1,...,F_n\}$ in which $F_1,...,F_n$ represents all overlapping features. With the following equation, we determine the approximation of feature A:

$$\begin{aligned} V_A &= V(P_A\{A,B\}) \\ &- V(P_B\{B\}) - V(P_{parent(A)}\{parent(A)\}) \\ &+ V(P_{F_1,...,F_n}\{F_1,...,F_n\}) \end{aligned}$$

We are aware of the fact that there might be cycles in a feature model such that each feature of the cycle cannot be measured without any other feature of the cycle. Hence, approximations of individual features in a cycle cannot be computed. However, this is not necessary at all, because in each product either all features of a cycle are present or none. Therefore, the solution in this case is the definition of a feature interaction that contains the approximation of all features in a cycle. The creation of such feature interactions can be automated, which we did for the Violet product line (see Section IV).

IV. Evaluation

Since our approach only predicts non-functional properties and cannot provide precise results, we evaluate the accuracy of our approximations with a series of case studies. We developed the tool *SPL Conquerer* to automate measurement, computation of the product set, and approximation of a feature's non-functional properties. We demonstrate that our predictions are sufficiently accurate for many real-world scenarios, in which we want to constrain the configuration space or select a nearly-optimal product regarding some non-functional property. For brevity, we show only a selection of our evaluation in the paper. We refer the interested reader to our website for more and detailed information.[7]

We selected nine existing SPLs with very different characteristics to cover a broad spectrum of scenarios. In Table I, we give an overview of our case studies: We selected SPLs of different sizes (2 500 to 13 million lines of code, 5 to 100 features), implemented with different languages (C, C++, and Java) and different variability mechanisms (conditional compilation and feature-oriented programming), from different domains (e.g., operating systems, database engines, end-user applications), and from different developers (both academic and industrial). From Linux, due to the huge configuration space, we considered only a subset of 25 features, selected as representative by a domain expert.[8] Among the 25 features were some features that we knew would change the footprint (as the evaluated non-functional property) of other features (e.g., *OPTIMIZE_INLINING* and *CC_OTPIMIZE_FOR_SIZE* both apply global optimizations). Although very different SPLs are used, the main technical commonality is that, in all case studies, we can automatically generate and compile products for a given feature selection.

As an evaluation strategy, we measure a set of products from each SPL to approximate non-functional properties per feature (using each the feature-wise, the interaction-wise, and the pair-wise approach) and compare the resulting predictions with the actual properties of all products. Exceptions are Violet, SQLite, and Linux kernel, because the measurement of all products is not feasible in reasonable time. We choose to use 100 random products instead.[9] Since our approach is customer-centered, we calculate a fault rate of our prediction as the relative difference between predicted and actual property: $\frac{|actual - predicted|}{actual}$. To set the fault rate into perspective, we provide also the highest and lowest measured value in Table I.[10]

A. Accuracy of Predicting Footprint

We evaluate the accuracy of our approach by means of the non-functional property *footprint* (the binary size of the compiled product). We selected footprint for several reasons:

- Although it may appear trivial, footprint is actually quite difficult to predict. As for performance, feature interactions can have an immense effect: Crosscutting features can significantly influence the footprint of many other features. Interactions due to shared libraries, nested #ifdefs (code is only included when two or more features are selected), or possible compiler optimizations make footprint difficult to predict.
- We can measure footprint quickly and reliably, which is important for a large-scale evaluation with multiple case studies as ours. We can easily reproduce values, and we exclude noise and confounding influences, such as system load, which easily can bias benchmarks. In addition, since we need to automate a high number of measures (not only for products used to approximate values per feature, which a normal user of our approach would do, but, in addition, also for reference products to compare predicted and actual size), it comes in handy that measuring footprint is quick.
- Finally, since we are not domain experts for all SPLs, it is difficult to evaluate the influence of domain knowledge to recognize possible interactions. For footprint, many implementation approaches give us a chance of using heuristics to detect possible interactions (e.g., by searching for nested #ifdefs); hence, we can provide still insight into the benefits of the interaction-wise approach on different SPLs.

[7] http://fosd.de/SPLConqueror

[8] The domain expert selected the following features, which cover both modular features, such as drivers, as well as crosscutting features implemented using conditional compilation: DEBUG_BUGVERBOSE, INLINE_SPIN_LOCK, OPTIMIZE_INLINING, CC_OPTIMIZE_FOR_SIZE, MODULE_UNLOAD, FRAME_POINTER, MODULE_SRCVERSION, DNOTIFY, INOTIFY_USER, FIRMWARE_IN_KERNEL, SND_VERBOSE_PROCFS, POWER_SUPPLY_DEBUG, PCNET32, NF_CONNTRACK_IPV6, NLS_ISO8859_15, NO_HZ, NET_POLL_CONTROLLER, PRINTK_TIME, SATA_NV, SC520_WDT, KPROBES_SANITY_TEST, I2C_DEBUG_ALGO, CHR_DEV_SCH

[9] We created the random products as follows: For each feature, we randomly decide whether to include or not include it. If the resulting feature selection is not valid according to the feature model, we start over.

[10] Note, the fault rate requires careful interpretation, and a base product or a feature with overproportional influence on the property may distort the figure. We cannot provide a relative fault rate corresponding to some base or minimal product, because it is not clear what the base or minimal product is (we would need to measure all products in the first place). However, for comparison we provide the highest and lowest measured property in each SPL.

								Product size in KB	
Product line	Domain	Origin	Language	Implementation technique	Features	Products	LOC	Min*	Max*
LinkedList	Data structures	Academic	Java	Composition (Jak)	18	492	2 595	4.4	10.5
Prevayler	Database	Industrial	Java	Conditional compilation	5	24	4 030	87	169
ZipMe	Compression	Academic	Java	Conditional compilation	8	104	4 874	79	99
PKJab	Instant messenger	Academic	Java	Composition (FeatureHouse)	11	72	5 016	39	161
SensorNetwork	Simulation	Academic	C++	Composition (FeatureC++)	26	3240	7 303	19	875
Violet	UML editor	Academic	Java	Composition (FeatureHouse)	100	ca. 10^{20}	19 379	6.3	185
Berkeley DB	Database	Industrial	C	Conditional compilation	8	256	209 682	1 800	2 740
SQLite	Database	Industrial	C	Conditional compilation	85	ca. 10^{23}	305 191	166	200
Linux kernel$^+$	Operating system	Industrial	C	Conditional compilation	25	ca. 33 000 000	13 005 842	11 245	13 829

* Minimal and maximal size of large SPLs may not be exact, because we cannot measure all products. We list the smallest and largest measured value.
$^+$ We use only subset 25 features of the Linux kernel selected by a domain expert.

TABLE I
OVERVIEW OF THE SPLS USED IN OUR EVALUATION.

	#Measurements (in percent*)		
Product line	Feature-wise	Interact.-wise	Pair-wise
LinkedList	11 (2.2)	13 (2.6)	88 (17.8)
Prevayler	5 (6)	7 (29)	17 (70)
ZipMe	8 (8)	10 (10)	21 (20)
PKJab	8 (11)	8 (11)	36 (50)
SensorNetwork	26 (1)	34 (1)	252 (8)
Violet	80 (0)	2115 (0)	5229 (0)
Berkeley DB	9 (4)	15 (6)	33 (13)
SQLite	85 (0)	146 (0)	3306 (0)
Linux kernel	25 (0)	207 (0)	326 (0)

* Number of measurements in relation to the number of products in percent (cf. I).

TABLE II
NUMBER OF MEASUREMENTS FOR OUR DIFFERENT APPROACHES.

In Figure 5, we summarize the results of our footprint measurements and predictions for all SPLs using box plots.[11] In Figure 6, we additionally plot predicted values against measured results for one of our case studies, which illustrates the accuracy visually (for a perfect prediction, all dots would lie on the diagonal line).

Our predictions are usually accurate even for the feature-wise approach. Predictions based on more measurements are even better. However, we identified an exception of this rule for the Violet SPL, which we discuss next. Nevertheless, even predictions based on feature-wise measurements usually only exhibit a fault rate of a few percent, which can be reduced to under one percent with more measurements (by defining feature interactions). Moreover, we show in Table II the number of measurements we did to infer approximations of a feature's footprint. In brackets, we indicate the percentage of the number of measurements compared to the number of all possible products. We only need to measure a small fragment of all products.

Let us have a closer look at three case studies, Berkeley DB, Violet, and the Linux kernel, because their results show interesting points for further investigations. Berkeley DB is a an SPL that makes exhaustive use of nested #ifdefs. This means, it is often the case that a certain feature combination requires additional code which increases the footprint. In Figure 6, we show the results of our different approaches. Although we only measured 9 products of Berkeley DB for the feature-wise approach, we have an average fault-rate of about 1.9 % for all 256 products. We often predict a too low footprint, because we did not measure those feature interactions that include additional source code in a product (nested #ifdef). Hence, for larger products containing an increasing number of features that depend on each other, the fault-rate increases.

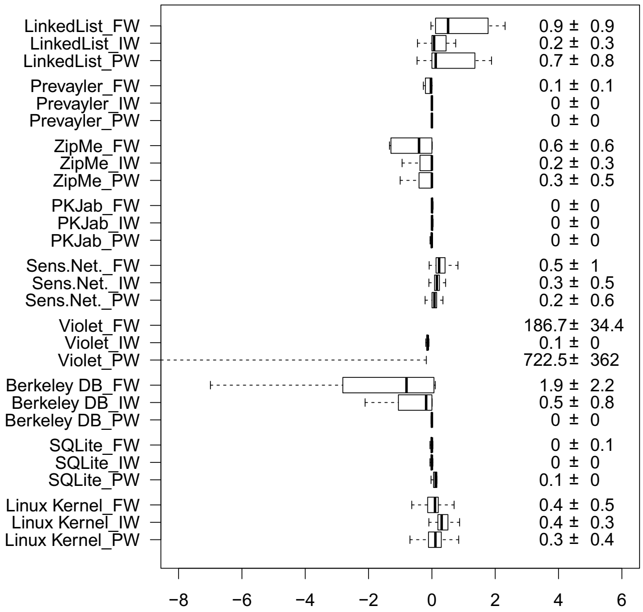


Fig. 5. Fault-rates in percent of all SPLs using the different approaches including the absolute average and standard deviation. To improve readability, we omit the plot for Violet_PW. FW: Feature-wise; IW: Interaction-wise; PW: Pair-wise

To improve the quality of the measurement, we analyzed the source code of Berkeley DB to identify such (syntactic) feature interactions. With a self-written tool, we identified 6 cases of nested #ifdefs. These #ifdefs cause feature interactions, for which we measured 6 additional products. Considering these known feature interactions the average fault-rate is reduced to

[11] Fig. 5 uses a *box plot* to describe data [15]. It plots the median as thick line and the quartiles as thin line, so that 50 % of all measurements are inside the box. Whiskers describe the remaining 50 % measurements.

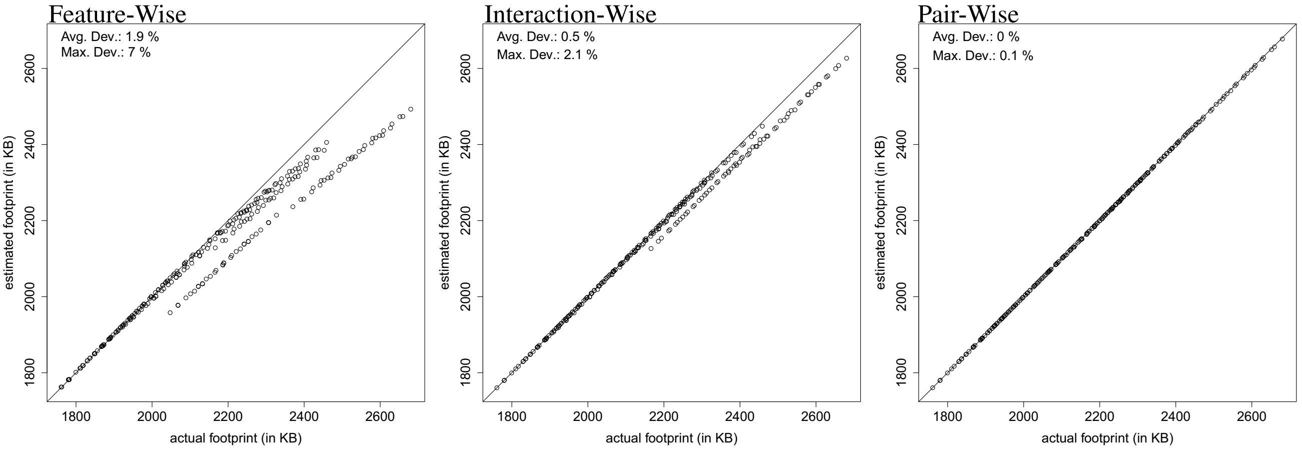


Fig. 6. Measured and predicted footprint in KB of Berkeley DB (compiled as static link library) using different approaches.

0.5 %. Thus, by measuring only 15 products of Berkeley DB, we can almost predict the footprint of all 256 products with a very high accuracy. Finally, we applied the pair-wise approach to Berkeley DB and measured 37 additional products. This eliminated faults almost entirely (maximum fault-rate of 0.1%) as visible in Figure 6.

With Violet, we identified the worst fault rates (cf. Figure 5). The reason is the complex mapping between some features and implementation units. That is, an individual feature maps to multiple code units and a single code unit may be required by multiple features. Hence, when measuring such a feature, the corresponding product contains several implementation units that are also present when measuring another feature's product. Therefore, predicting the footprint of a product that includes multiple features with an overlapping set of implementation units is inaccurate, because we consider the footprint of the implementation units multiple time. The pair-wise approach is even worse, because more than two features map to the same code. Furthermore, we did not use thresholds (e.g., limiting the size of a feature interaction to the sum of the size of the participating features) to limit the influence of interactions (as we would do for practical use) to gain insights in the nature of feature interactions. Hence, when predicting a product containing three features, we aggregate three times the approximation of pair-wise feature interactions, though only two times would be correct. If more features interact, the inaccuracy increases, which is an interesting insight about feature interactions. Fortunately, the mapping that causes the problems can easily be analyzed to automatically define appropriate feature interactions. Hence, with additional measurements, we can easily improve the accuracy to over 99 % (see Figure 5 Violet_IW).

Linux is another interesting case. Among the 25 features selected by a domain expert were some features that we knew would change the size of other features (e.g., *OPTIMIZE_INLINING* and *CC_OTPIMIZE_FOR_SIZE* both apply global optimizations). Hence, we expected large fault-rates regarding the 100 randomly generated products. We were surprised that we still achieved a precise prediction even with the feature-wise approach, partially, because the features had a weaker effect than expected. We slightly improved the accuracy of the interaction-wise approach, because we defined feature interactions between more than two features (i.e., we define an interaction between every feature and the features *OPTIMIZE_INLINING* and *CC_OTPIMIZE_FOR_SIZE*).

In general, our evaluation shows that, at least for footprint, our prediction model provides good approximations of the actual non-functional property. We discuss next how this could be used in practice, and how many measurements one should perform in the following.

B. Best Number of Measurements

The results have shown that the feature-wise approach is a good initial approximation of a product's non-functional properties. However, if there is domain knowledge available, we suggest to always use it as it can significantly improve accuracy with only few additional measurements. Additionally, complex mappings or unknown feature interactions can cause large fault-rates, which would require to always include the mapping in the feature model in terms of feature interactions.

If domain knowledge is not available, it is difficult to decide whether investment in more measurements is worth. For some non-functional properties, such as footprint, it might be feasible to extract information about interactions from the source code. Sometimes other sources of domain knowledge might be available.

At this point, the measurements are often already sufficiently accurate to use them during product derivation, for example, to rule out products that obviously do not fulfill the required constraints or to determine a set of possible candidates for the optimal product.

Finally, the pair-wise approach achieves often the highest accuracy, but increases the effort for measurements significantly ($O(n^2)$). We only recommend it, if there is either no domain knowledge available or to combine it with the interaction-wise approach when the number of features is acceptably small.

We illustrate our observations in Figure 7. In general, the accuracy increases (i.e., the fault-rate decreases) with additional measurements, but a stakeholder must be aware of the fact that too many measurements render the approach infeasible. For example, if we want to measure all 8.000 features of the Linux kernel [7] (we only considered 25 in our evaluation) with the pair-wise approach, we would need about 64 million measurements (which, extrapolating from our experiments, would take roughly 2 years to measure on a cluster of 1.000 computers). In contrast, the feature-wise approach requires the measurement of only about 8.000 products (which could be realistically done in one day using a cluster of 100 computers). For our approach, balancing between desired accuracy and investment in measurements is essential.

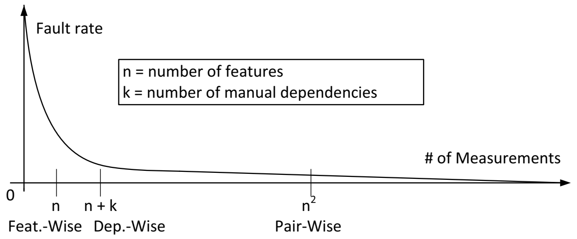


Fig. 7. Observed relation between number of measurements and fault rate of predictions.

C. Threats to Validity

There are some threats to internal and external validity. Regarding internal validity, we need to point out that, for SPLs with many features, we only sampled 100 products to compare prediction and actual property, because we cannot possibly generate and measure all products in reasonable time. We are aware of that this evaluation leaves room for outliers, but we did not find any outliers in our data by sampling.

Regarding external validity, although we use a large variety of different SPLs, we are aware of that the results of our evaluations are not automatically transferable to all other SPLs. Our used SPLs have feature models with a typical structure and number of constraints (according to the criteria in [16]). We did not evaluate SPLs with an unusual possibly degenerated feature model, which might influence the computation of the product set. Thus, we cannot generalize our results to such product lines.

Finally, of course we cannot generalize our evaluation to non-functional properties other than footprint. However, we argue that – similar to performance and many other non-functional – footprint is subject to many internal and external influences. These influences have a crucial impact on the applicability of our approach, which we could handle with footprint. In this paper, we want to convey that the approach of approximating non-functional properties per features is realistic at all.

V. Related Work

Many product-derivation approaches for SPLs have been proposed in the past [17], [18], [19], [20], [21]. However, they do not allow a user to specify non-functional constraints or to derive a product with desired non-functional properties. Research in this area focuses on reducing the complexity of the configuration process and supporting the user with tools during feature selection. Nevertheless, some approaches also allow a user to optimize the feature selection for a specific non-functional property. Benavides et al. presented a technique based on CSP solvers to find an optimal product [22], [23]. The solver evaluates values attached to features in the feature model and then computes an optimal configuration for a small number of features. Their studies show that with an increasing number of features the computation time grows exponentially.

White et al. [24], [25] enable users to define constraints on non-functional properties to derive a product with desired non-functional properties. They propose a solution based on filtered Cartesian flattening to approximate a nearly optimal product for even large scale feature models. In contrast to our approach, they do not consider the measurement and computation of a feature's non-functional properties. Hence, their proposed techniques can be combined with our computed feature values.

Only a few approaches apply measurements of non-functional properties to SPLs. Zubrow and Chastek proposed measures for SPL that evaluate the development effort for an SPL [26]. Lopez-Herrejon and Apel express the complexity of an SPL in terms of variation points with a dedicated metric [27]. An approach close to our work is the measurement of the binary size of an aspect-oriented SPL [28]. The authors compiled aspects in distinct files and measured the binary size. The footprint of different products can then be computed. Another related approach for optimizing non-functional properties was developed in the *COMQUAD* project [29]. The project focuses on techniques for tracing and adapting non-functional properties in component-based systems. The approach requires an own component model, which is an extension of *Enterprise JavaBeans* and *CORBA Components* and relies on AOP as implementation technique. In contrast to these approaches, we consider also other properties and address the exponential number of products that occur during the derivation process. Furthermore, we propose an implementation and language independent approach not restricted to aspects. Additionally, we maintain a product-line model to explicitly address feature interactions which is not supported by others.

Sincero et al. [30], [11] propose to estimate a product's non-functional properties based on a knowledge base consisting of measurements of already produced products. Their aim is to find a covariance between feature selection and measurement. This way, they can give information about how a feature influences a non-functional property during configuration. In contrast to our approach, they do not save actually measured feature's non-functional properties but a quantification of how a feature affects a property. When it comes to product derivation, they do not present an expected value for a product's properties, as we do, but can show with a slider whether a feature selection improves a property such as performance or not.

VI. Conclusion

We presented an approach to approximate a feature's non-functional properties for the prediction of non-functional properties of products in advance without actually generating and measuring them. The main idea is to produce a set of products that differ in a single feature such that we can interpret the delta in the products properties as the approximation of the corresponding feature. We propose three different approaches to measure approximations of features and feature interactions. These approaches scale from linear to a quadratic complexity in terms of the number of features in an SPL. In an evaluation, in which we compare predicted with actual footprints in different SPLs, we achieve an accuracy of 98 %, on average. Especially, the approach that measures *known interactions* between features achieve a high accuracy with a small number of measurements.

We believe that our approach is applicable for all quantifiable non-functional properties. In future work, we plan to demonstrate the generality of our approach for other non-functional properties, such as performance. Furthermore, we are working on ways to automatically identify feature interactions. This way, we may reduce our fault rate and improve the efficiency of determining a feature's non-functional properties with only few additional measurements.

Acknowledgments

We thank Martin Kuhlemann, Thomas Thüm, and Janet Feigenspan for helpful comments. Norbert Siegmund's word is supported by the German ministry of education and science (BMBF), Nb. 01IM10002B. The work of Marko Rosenmüller, Sven Apel, and Sergiy S. Kolesnikov is supported by the German Research Foundation (DFG), project numbers #SA 465/34-1 #AP 206/2-1, and #AP 206/4-1. Kästner's work is supported by ERC grant #203099

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[2] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," SEI, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] T. Henzinger and J. Sifakis, "The discipline of embedded systems design," *Computer*, vol. 40, no. 10, pp. 32 –40, 2007.

[5] T. A. Henzinger, "Two challenges in embedded systems design: Predictability and robustness," *Philos Transact A Math Phys Eng Sci.*, vol. 366, no. 1881, pp. 3727–3736, 2008.

[6] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing PLA at Bosch gasoline systems: Experiences and practices," in *Proc. Int'l Software Product Line Conference*. IEEE Computer Society, 2004, pp. 34–50.

[7] R. Lotufo, S. She, T. Berger, A. Wasowski, and K. Czarnecki, "Evolution of the Linux kernel variability model," in *Proc. Int'l Software Product Line Conference*. Springer-Verlag, 2010, pp. 136–150.

[8] P. Toft, D. Coleman, and J. Ohta, "A cooperative model for cross-divisional product development for a software product line," in *Proc. Int'l Software Product Line Conference*. Kluwer Academic Publishers, 2000, pp. 111–132.

[9] S. S. Stevens, "On the theory of scales of measurement," *Sciences*, vol. 103, no. 2684, pp. 677–680, 1946.

[10] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake, "Measuring non-functional properties in software product lines for product derivation," in *Proc. Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2008, pp. 187–194.

[11] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk, "Approaching non-functional properties of software product lines: Learning from products," in *Proc. Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2010, pp. 147–155.

[12] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, and G. Saake, "Integrated product line model for semi-automated product derivation using non-functional properties," in *Proc. Workshop on Variability Modelling of Software-intensive Systems*. University of Duisburg-Essen, 2008, pp. 25–31.

[13] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *Proc. Int'l Software Product Line Conference*. IEEE Computer Society, 2010, pp. 196–210.

[14] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[15] T. Anderson and J. Finn, *The New Statistical Analysis of Data*. Springer, 1996.

[16] T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *Proc. Int'l Conf. Software Engineering*. IEEE Computer Society, 2009, pp. 254–264.

[17] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Software Product Line Conference*, 2005, pp. 7–20.

[18] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature modeling plug-in for eclipse," in *Workshop on eclipse technology eXchange*. ACM Press, 2004, pp. 67–72.

[19] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models," in *Proc. Int'l Software Product Line Conference*, 2004, pp. 266–283.

[20] G. Botterweck, D. Nestor, A. Preußner, C. Cawley, and S. Thiel, "Towards supporting feature configuration by interactive visualization," in *Proc. Workshop on Visualisation in Software Product Line Engineering*. IEEE Computer Society, 2007, pp. 125–131.

[21] R. Rabiser, D. Dhungana, and P. Grünbacher, "Tool support for product derivation in large-scale product lines: A wizard-based approach," in *Proc. Workshop on Visualisation in Software Product Line Engineering*. IEEE Computer Society, 2007, pp. 119–124.

[22] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," in *Proc. of Int'l Conf. on Advanced Information Systems Engineering*. Springer-Verlag, 2005, pp. 491–503.

[23] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés, "FAMA: Tooling a framework for the automated analysis of feature models," in *Proc. Workshop on Variability Modelling of Software-intensive Systems*. Springer-Verlag, 2007, pp. 129–134.

[24] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko, "Automating product-line variant selection for mobile devices," in *Proc. Int'l Software Product Line Conference*. IEEE Computer Society, 2007, pp. 129–140.

[25] J. White, B. Dougherty, and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268–1284, 2009.

[26] D. Zubrow and G. Chastek, "Measures for Software Product Lines," SEI, Tech. Rep. CMU/SEI-2003-TN-031, 2003.

[27] R. Lopez-Herrejon and S. Apel, "Measuring and characterizing cross-cutting in aspect-based programs: Basic metrics and case studies," in *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2007, pp. 422–437.

[28] F. Hunleth and R. Cytron, "Footprint and feature management using aspect-oriented programming techniques," in *Proc. Joint Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*. ACM Press, 2002, pp. 38–45.

[29] R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler, "Towards pervasive treatment of non-functional properties at design and run-time," in *Proc. Int'l Conf. Software and Systems Engineering and their Applications*. CNAMCMSL, 2003.

[30] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat, "On the configuration of non-functional properties in software product lines," in *Proc. Software Product Line Conference, Doctoral Symposium*. Kindai Kagaku Sha Co. Ltd., 2007, pp. 167–173.