Information and Software Technology xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption

Norbert Siegmund [a,*], Marko Rosenmüller [a], Christian Kästner [b], Paolo G. Giarrusso [b], Sven Apel [c], Sergiy S. Kolesnikov [c]

[a] Department of Computer Science, Otto-von-Guericke University Magdeburg, Magdeburg, Germany
[b] Department of Computer Science and Mathematics, Philipps University Marburg, Germany
[c] Department of Informatics and Mathematics, University of Passau, Germany

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Non-functional properties
Prediction
Measurement
Software product lines
SPL Conqueror

ABSTRACT

Context: A software product line is a family of related software products, typically created from a set of common assets. Users select features to derive a product that fulfills their needs. Users often expect a product to have specific non-functional properties, such as a small footprint or a bounded response time. Because a product line may have an exponential number of products with respect to its features, it is usually not feasible to generate and measure non-functional properties for each possible product.
Objective: Our overall goal is to derive optimal products with respect to non-functional requirements by showing customers which features must be selected.
Method: We propose an approach to *predict* a product's non-functional properties based on the product's feature selection. We aggregate the influence of each selected feature on a non-functional property to predict a product's properties. We generate and measure a small set of products and, by comparing measurements, we approximate each feature's influence on the non-functional property in question. As a research method, we conducted controlled experiments and evaluated prediction accuracy for the non-functional properties *footprint* and *main-memory consumption*. But, in principle, our approach is applicable for all quantifiable non-functional properties.
Results: With nine software product lines, we demonstrate that our approach predicts the footprint with an average accuracy of 94%, and an accuracy of over 99% on average if feature interactions are known. In a further series of experiments, we predicted main memory consumption of six customizable programs and achieved an accuracy of 89% on average.
Conclusion: Our experiments suggest that, with only few measurements, it is possible to accurately predict non-functional properties of products of a product line. Furthermore, we show how already little domain knowledge can improve predictions and discuss trade-offs between accuracy and required number of measurements. With this technique, we provide a basis for many reasoning and product-derivation approaches.

© 2012 Elsevier B.V. All rights reserved.

1. 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 customize a product by means of a selection of features that satisfies their functional requirements [3]. With many contemporary implementation mechanisms, one can automatically *generate* products based on feature selections. For example, we can map features to preprocessor definitions (i.e., `#define` statements) to specify which features are selected. The preprocessor removes all code units that belong to not selected features, that is, undefined preprocessor identifiers. Another way to customize a product's behavior is to use program parameters. For example, users can specify command-line parameters or configuration files to customize a product for their needs (e.g., customizing log levels of a web server via a configuration file). Although, in these examples, variability is evaluated at runtime, we consider also such products in our work and use uniformly product-line terminology.

In addition to functional requirements, users are also interested in *non-functional properties* of a product, such as performance,

* Corresponding author. Address: University of Magdeburg, P.O. Box 4120, 39016 Magdeburg, Germany.
E-mail address: nsiegmun@ovgu.de (N. Siegmund).
URL: http://wwwiti.cs.uni-magdeburg.de/~nsiegmun/ (N. Siegmund).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.07.020

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

footprint, and reliability. Non-functional properties are especially important in the domain of embedded and real-time 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 much money in mass production or increase user acceptance.[1] 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 quantifiable property (e.g., the fastest product). This, however, means that we have to measure many products, until we find a feature combination that satisfies certain non-functional requirements.

To search for the feature combination that is optimal with regard to a certain non-functional property, we may have to generate and measure *all* products. Even small SPLs with less than a hundred features can already have millions of products, and industrial-size SPLs can contain thousands of features [6–8]. Generating products for all feature selections is thus not feasible in practice. Consequently, we investigate alternatives based on heuristics that estimate non-functional properties without generating and measuring all individual products.

Our goal is to *predict* non-functional properties of customizable products. We do so by measuring the influence of each feature on a non-functional property. We compute a small but suitable set of products, by analyzing the relationships between features documented in a feature model [2]. Next, we compile and measure these products and quantify the influence of each feature from deltas between these products. Finally, we predict a product's non-functional properties by adding the quantified influences of all selected features.

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" will affect footprint and other non-functional properties of other features). Additionally, there may be a complex mapping from features to implementation assets that can lead to false approximations. To take such feature interactions into account, we develop a model in which we can document and incorporate known feature interactions. We measure the influence of these interactions to improve the accuracy of our predictions.

Our approach is independent of a particular implementation mechanism (i.e., we treat an SPL as a black box) and can be applied to different quantifiable non-functional properties. To evaluate our approach, we conducted two experiments, in which we compare predicted against measured non-functional properties.

For the first experiment, we choose *footprint* (binary size of a generated product) and selected nine SPLs of different sizes, languages, varying implementation techniques, from different domains (e.g., operating systems, database engines, end-user applications), and from different developers (both academic and industrial). In the second experiment, we choose the property *main-memory consumption* and selected six customizable programs from different domains (e.g., web server, compiler, database engines) and with varying customization techniques (#ifdef, command-line parameters, configuration files). With a linear number of measurements (i.e., without considering feature interactions), our predictions have an accuracy of 78.7% on average for all SPLs for footprint and an accuracy of 86.4%, on average, for main-memory consumption. By exploiting domain knowledge about feature interactions, predictions of footprint improve to 99.8% for all SPLs and measuring all pair-wise interactions for main-memory consumption, accuracy improves to 89%, on average.

This is a revised and extended version of a previous conference paper [9]. Compared to this earlier paper, we provide more evidence of the generality of our approach, in particular, by means of additional study of a second non-functional property. In addition to predicting only footprint in prior work, we predict *main-memory consumption* in six different sample programs. Moreover, a subset of four programs can be customized via program parameters which extend the scope of our approach to a broader range of application scenarios.

2. Problem statement

Non-functional properties are diverse, and it is not obvious how we can interpret and handle measurements of these properties. We concentrate on properties that can be quantified (i.e., that are measurable). The theory of measurement defines different levels (nominal, ordinal, interval, and ratio) of how measured values can be interpreted [10]. Our approach relies on *interval and ratio-scale*-based measures, because the values of two measurements reflect differences of the according property.[2]

To measure the vast majority of non-functional properties, we have to actually generate and execute a product [11]. A key idea of our approach is to identify the *influence of an individual feature* on the product's non-functional properties, because it is not clear which feature of a product contributes in which quantity to a product's properties. Even worse, a feature's influence on non-functional properties 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 how a feature influences a non-functional property is usually not possible [12].

Problems of approximating a feature's influence on 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 (i.e., a piece of code that is required only when features *A* and *B* are present). In the second case, multiple features share a certain piece of code. That is, this code is present only once in a product no matter how many features require it.

To illustrate the problems of feature interactions, we use a simple example that already exhibits measurement problems. In Fig. 1, we show the C++ implementation of a linked list with two features: *PrintList* and *PrintElement*. Features are implemented with conditional compilation. To measure say footprint (measured as the binary size of the compiled product), 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. We can alter our example to 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 Fig. 1) to use the external logging library. The library has a considerably larger binary size

[1] Discussions among researchers and industry representatives at conferences (like software product line conference) and Dagstuhl seminars emphasize the importance of this problem.

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

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

```
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 };
```

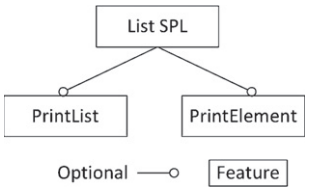


Fig. 1. C++ code of a list implementation with two features: *PrintList* and *PrintElement*. We show the feature model in the upper right corner.

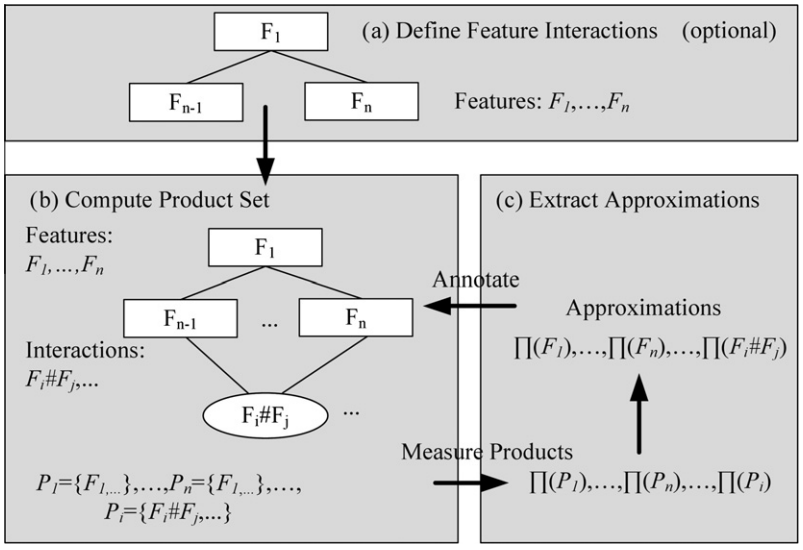


Fig. 2. The process of computing approximations of non-functional properties for features and feature interactions.

than the features itself. When approximating a feature's influence on footprint, the predominant part 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 incorrectly. The reason is that both features share the same library, which is included only once in the product, but was measured twice (once for each feature).

3. Non-functional properties of features

In this section, we present our approach to approximate the influence of a feature and feature interactions on a non-functional property. First, we describe the general concept of our approach. Second, we explain algorithms necessary to extract the approximations of each feature's influence on a non-functional property from a (minimal) set of products. We start with a description of our notation that we use to express configurations, measurements, and approximations of a feature's influence on a property. For illustration, we use the property footprint in the examples.

We describe a product P as a set of selected features $F_1, \ldots, F_n$: $P = \{F_1, \ldots, F_n\}$. $\Pi(\{F_1, \ldots, F_n\})$ (or short: $\Pi(P)$) refers to the specific non-functional property currently being considered. Furthermore, we represent the approximation of the influence of feature F_i on a non-functional property with $\Pi(F_i)$. Finally, we use operator # to indicate an interaction between features. For example, we denote the interaction between features F_1 and F_2 as $F_1\#F_2$ and we treat $F_1\#F_2$ like another feature.

As an underlying data structure and prediction model, we use *feature models* (as shown in Fig. 1) [2]. A feature model specifies all valid feature combinations and thus all configurations that can be generated.It has a hierarchical structure beginning with a root node, which represents the domain concept. Features can be mandatory (required in all products when the parent feature is selected) or optional (the user can decide to select this feature). Furthermore, we can specify an or relationship between features, which denotes that when the parent feature is selected we have to select at least one child feature of the or relationship. Similarly, we specify alternative groups with the only difference that we have to select exactly one child feature. We explain in the next section how these relationships affect the approximation of a feature's influence on non-functional properties.

3.1. Approximation process

The general idea of approximating a feature's influence on a non-functional property is to measure the delta between two

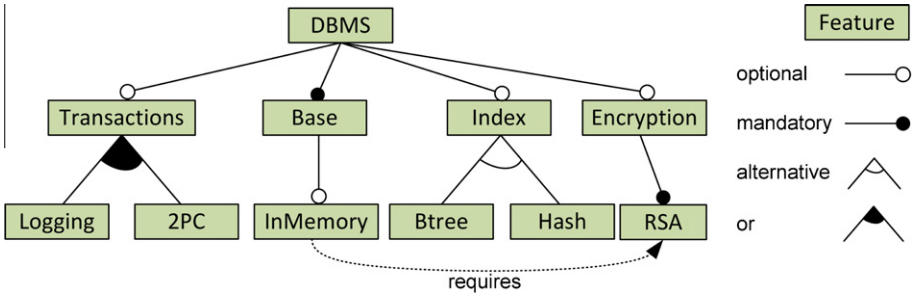


Fig. 3. Sample DBMS product line. The root denotes the concept. 2PC: two phase commit protocol.

Table 1
Set of products to approximate the influence of each feature. All measured values are in KB.

Feature	Feature product	Π	Delta product	Π
Base	P_1 = {Base}	420	$\emptyset$	0
Encryption	P_2 = {Base, Encryption, RSA}	730	P_1 = {Base}	420
RSA	P_2 = {Base, Encryption, RSA}	730	P_2 = {Base, Encryption, RSA}	730
Index	P_4 = {Base, Index, Hash}	570	P_1 = {Base}	420
Btree	P_3 = {Base, Index, Btree}	740	P_1 = {Base}	420
Hash	P_4 = {Base, Index, Hash}	570	P_3 = {Base, Index, Btree}	740
InMemory	P_5 = {Base, InMemory, Encryption, RSA}	610	P_2 = {Base, Encryption, RSA}	730
Transactions	P_6 = {Base, Transactions, Logging, 2PC}	995	P_1 = {Base}	420
Logging	P_6 = {Base, Transactions, Logging, 2PC}	995	P_7 = {Base, Transactions, 2PC}	885
2PC	P_6 = {Base, Transactions, Logging, 2PC}	995	P_8 = {Base, Transactions, Logging}	845

products that differ only in the presence or absence of this feature. We interpret the delta of two products as the approximation of the added (or removed) feature's influence.

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

$$\Pi(PrintList) = \Pi(\{base, PrintList\}) - \Pi(\{base\})$$

In Fig. 2, we illustrate the approach of approximating a feature's influence on non-functional properties for SPLs and customizable programs that support an automated product generation. Using a feature model, we determine a small, but suitable set of products (Fig. 2b). 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 (Fig. 2c). This means, to enable for each feature the computation of the delta between two products, we would need $2n$ measurements altogether for n features. By using the values of previous measurements, we can reduce the number of required measurements to $n + 1$.

Furthermore, as an optional initial step (Fig. 2a), we allow stakeholders to define feature interactions in the corresponding feature model [13].[4] 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 measured product and *predicted* non-functional properties of the same product. For example, if we know the existence of the feature interaction *PrintList#PrintElement*, we might predict $\Pi(\{PrintList, PrintElement\})$ = 220 KB, whereas the measured footprint is 200 KB. Hence, we would assign the difference between interaction and prediction (−20 KB) to the interaction *PrintList#PrintElement* as its influence on footprint.

3.2. 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 Fig. 3, we show a feature model of a database management (DBMS) SPL. We use this SPL as a running example throughout the remaining paper. As we explained previously, we need two products per feature to measure the delta of these products. One product in which the feature is present, called feature product, and another product, called *delta product*, in which the feature is missing. We summarize all products in Table 1 that we need to approximate each feature's influence on footprint and describe step by step how we determine the feature's influence depending on the relationship in the feature model.

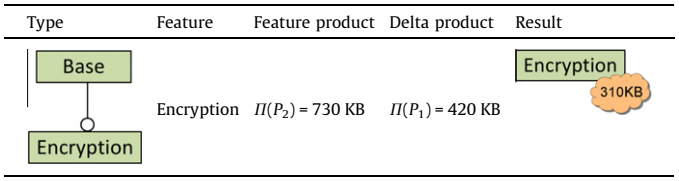
Note that there are SPLs that always require to select some features to derive a valid product (e.g., we may always must choose between alternative features). In such a case, we cannot determine a product that differs only in these features, because we always have to select them. To overcome this problem, our first task is to measure the influence of an *initial feature set* on a non-functional property. This initial feature set acts as the root feature for all features that have no parent feature. Since there is no product with fewer features, the delta product is the empty set for which the non-functional property is zero. Hence, we interpret the influence of the initial feature set on a non-functional property as the measured value of the corresponding product. For example, feature *base* must always be selected in our sample SPL (see Fig. 3):

Type	Feature	Feature product	Delta product	Result
Initial	Base	$\Pi(P_1)$ = 420 KB	$\Pi(\emptyset)$ = 0 KB	Base 420KB

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

[4] In a parallel line of research, we developed an approach that automatically detects feature interactions for the property performance [14].

(1) *Optional.* In an optional relationship, it is not required to select the child feature. Hence, we generate a product that contains only the parent feature.[5] Additionally, we need a second product with the optional child feature. In our sample SPL, feature *Encryption* is an optional feature. Since it has no parent feature, the initial feature set is considered as the root feature and acts as the parent feature. Based on the computed product set of Table 1, we measure the following products:



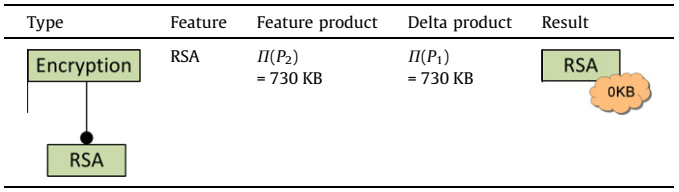
With these measurements we compute the influence of feature *Encryption*:

$$\begin{aligned}\Pi(Encryption) &= \Pi(P_2) - \Pi(P_1)\\ &= 730\ \text{KB} - 420\ \text{KB}\\ &= 310\ \text{KB}\end{aligned}$$

where P_1 represents the product that we measure for *Encryptions*' parent feature (which is the initial product set in this case).

(2) *Mandatory.* A mandatory relationship enforces that whenever the parent feature is selected, we must also select its child feature. As a consequence, we cannot measure the parent feature's influence on a property without measuring the influence also of the child feature. Hence, we set the value of the child feature to zero and the value of the parent to the sum of both influences. When a stakeholder selects the parent feature during product configuration, we show already the aggregated value of both features. This way, it is easy to see the implications of a feature selection for a stakeholder, because she usually selects features starting from the root node.

In our example, when selecting feature *Encryption*, we must also select its child feature *RSA*. Hence, we set the value of feature *RSA* to zero and the value of feature *Encryption* to the sum of both features.

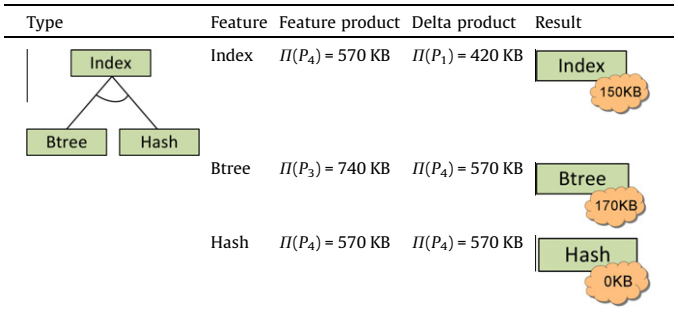


We compute the value for a mandatory relationship as follows:

$$\begin{aligned}\Pi(RSA) &= \Pi(P_2) - \Pi(P_2)\\ &= 730\ \text{KB} - 730\ \text{KB}\\ &= 0\ \text{KB}\end{aligned}$$

(3) *Alternative.* In an alternative relationship, we cannot select the parent feature of the relationship individually, but measure its value always in combination with its child features. Here, there is no unambiguous feature product for the parent feature, because we can measure this feature in multiple ways depending on which alternative feature we additionally selected. As a design decision, we use the product as the feature product of the parent feature that has the minimum measured binary size. This way, we can assign a minimum value to the parent feature that increases the footprint of a product by at least this respective value. If we would choose another feature product, we have to assign a negative footprint to at least one child feature (which has the minimum measured footprint). This may be unintuitive during configuration. 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. After having assigned a value to the parent feature, we use its feature product as the delta product for all of its child features.

The features *Btree* and *Hash* are alternative features with feature *Index* as their parent feature (see Fig. 3). To approximate their influences, we need the following three measurements:



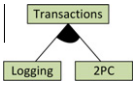
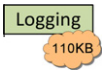
In Table 1, we see that we require a generated product per feature in the alternative group. In our example, we require two products $P_3 = \{Base, Index, Btree\}$ and $P_4 = \{Base, Index, Hash\}$. To compute the value of feature *Index*, as always, we need the measured product of the parent (or initial feature set): $P_1 = \{Base\}$. We use the following equations:

$$\begin{aligned}\Pi(Index) &= \text{Min}(\Pi(P_3), \Pi(P_4)) - \Pi(P_1)\\ &= \text{Min}(740\ \text{KB}, 570\ \text{KB}) - 420\ \text{KB}\\ &= 570\ \text{KB} - 420\ \text{KB}\\ &= 150\ \text{KB}\\ \Pi(Btree) &= \Pi(P_3) - \Pi(P_4)\\ &= 740\ \text{KB} - 570\ \text{KB}\\ &= 170\ \text{KB}\\ \Pi(Hash) &= \Pi(P_4) - \Pi(P_4)\\ &= 570\ \text{KB} - 570\ \text{KB}\\ &= 0\ \text{KB}\end{aligned}$$

(4) *OR.* In contrast to an alternative relationship, in an OR 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. For example, if we do not know the influence of feature *Transactions*, we would aggregate its influence twice for a product's prediction that contains its two child features *Logging* and *2PC*. Hence, we have to determine the approximation of the parent feature (e.g., feature *Transactions*) in an OR relationship rather than using the minimal measured product. To retrieve the value of the parent feature (*Transactions*), we need an additional measurement for the OR relationship in comparison to the alternative relationship. In this measurement, we create a product that contains two child features of the OR group (e.g., P_6 = {P6Base, Transactions, Logging, 2PC} in Table 1).

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

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

Type	Feature	Feature product	Delta product	Result
	Transactions	$\Pi(P_6)$ = 995 KB	$\Pi(P_1)$ = 420 KB	
	Logging	$\Pi(P_6)$ = 995 KB	$\Pi(P_7)$ = 885 KB	
	2PC	$\Pi(P_6)$ = 995 KB	$\Pi(P_8)$ = 845 KB	

With this additional measurement, we are able to compute the influence of feature *Transactions* and the remaining features of its group using the following equations:

$$\begin{aligned}\Pi(Logging) &= \Pi(P_6) - \Pi(P_7)\\ &= 995\ \text{KB} - 885\ \text{KB}\\ &= 110\ \text{KB}\\ \Pi(2PC) &= \Pi(P_6) - \Pi(P_8)\\ &= 995\ \text{KB} - 845\ \text{KB}\\ &= 150\ \text{KB}\\ \Pi(Transactions) &= \Pi(P_8) - \Pi(Logging) - \Pi(P_1)\\ &= 845\ \text{KB} - 110\ \text{KB} - 420\\ &= 315\ \text{KB}\end{aligned}$$

(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 restricts only the number of features and we already measure a product with a minimal number of features. In contrast, the requires constraint prohibits the measurement of a single feature. For example, we cannot measure feature *InMemory* without feature *RSA*. So, our approach is to measure first the product that includes the target of the requires constraint (i.e., feature *RSA* with product P_2 is the target of feature *InMemory*). Then, we measure the product that includes both features $P_5 = \{Base, InMemory, Encryption, RSA\}$.

Type	Feature	Feature product	Delta product	Result
	InMemory	$\Pi(P_5)$ = 610 KB	$\Pi(P_2)$ = 730 KB	

The problem of a cross-tree constraint is that we can have an overlapping set of features. That is, we subtract the influence of all parent features from the current feature and additionally subtract the influence of the required feature and its parent feature. When these parent features overlap (e.g., in the case of *Base*), it leads to the situation that we may subtract the influence of a feature twice. In our example, *Base* is such a feature, because it acts as the root feature (it is member of the initial feature set). Thus, we must identify the set of overlapping features $S = F_1, \ldots, F_n$ first and to omit subtracting them twice. With the following step, we determine the approximation of feature *InMemory* where $S = Base$:

$$\begin{aligned}\Pi(InMemory) &= \Pi(P_5) - \Pi(P_1) - \Pi(P_2) + \Pi(S)\\ &= 610\ \text{KB} - 420\ \text{KB} - 730\ \text{KB} + 420\ \text{KB}\\ &= -120\ \text{KB}\end{aligned}$$

We visualize the result of our computations in Fig. 4. We are aware of 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, because in each product either all features of a cycle are present or none. Therefore, the solution in this case is to transform the feature model into an alternative representation using atomic feature sets [15]. This way, these cycle features are composed as a single atomic feature and we can use our approach again.

3.3. Computing the product set for measurement

Measurements can be time consuming and expensive. This is the reason why we aim at further scaling the number of necessary measurements from $2n$ to $n + 1$ by reusing already executed measurements. To reach this goal, we use the hierarchical structure of feature models that allows us to reuse products already defined for the parent feature.

Since a feature model has a hierarchical form, every feature has a parent feature or the parent is the concept node. In the last case, we use the initial feature set as a 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 (a) contains the current feature, (b) has the minimal number of features, and (c) is valid. For example, when reaching feature *Encryption* of our sample SPL, we add product $P_2 = \{Base, Encryption, RSA\}$ to the product set. The delta product of this feature is the product of either the parent feature or the initial feature set. Hence, each newly determined product can use the previously defined product (e.g., the one for the parent) to compute the delta of its non-functional properties. An exception for this rule is the or relationship in which we have to measure an additional product to determine the influence of the parent feature of the or group (as we explained before). Furthermore, we have to measure an additional product per defined feature interaction, which we explain in Section 3.4.

Although the feature model of Fig. 4 has ten features,[6] we need to measure only eight configurations, because we reuse already measured configurations and we save a measurement due to the mandatory feature *RSA* and another measurement because of the alternative constraint between *Btree* and *Hash*. Feature groups (i.e., alternative and or groups) require the selection of (at least) one additional feature, but not the measurement of additional products. Hence, we use the same configuration for feature *Index* and *Btree*. Feature *InMemory* represents an interesting case, because it defines a requires constraint to feature *RSA*. Hence, the delta configuration for this feature must already include the required feature *RSA* and its parent feature *Encryption*, because we need two configurations that differ only in feature *InMemory*.

3.4. Measuring feature interactions

As we explained previously, feature interactions may affect the approximation of a feature's non-functional properties. That is, we approximate different values for a feature depending on the selection of other features, which can cause inaccurate predictions. For example, to approximate the influence of feature *Encryption* with $\Pi(Encryption) = 310$ KB, we used the following two products: $P_2 = \{Base, Encryption, RSA\}$ and $P_1 = \{Base\}$. However, we may also use products $P_9 = \{Base, Encryption, RSA, Transactions, Logging\}$ and $P_9 = \{Base, Transactions, Logging\}$ and compute a different delta: $\Pi(Encryption) = 350$ KB. Since the only differing feature is *Encryption*,[7] there is a feature interaction that influences the measured non-functional property.

If feature interactions are not known or should not be taken into account, *a pure feature-wise measurement approach* is used, that is,

[6] The concept node does not represent a feature.

[7] Of course, also feature *RSA* is a differing feature, but as a mandatory feature, we always measure its influence together with feature *Encryption*.

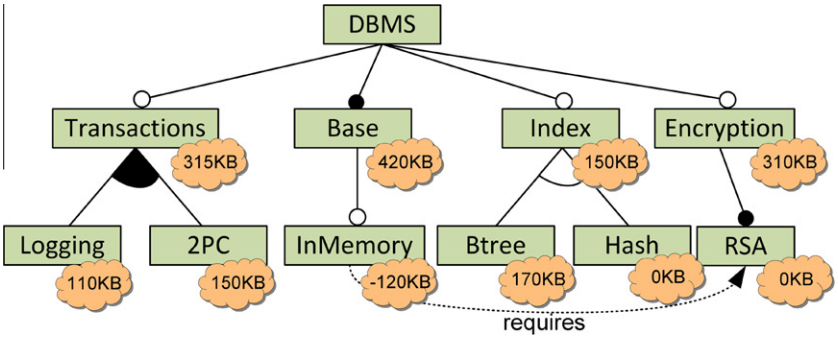


Fig. 4. Feature model after approximating the influence of each feature on footprint.

we ignore feature interactions and only estimate non-functional properties of individual features. Unfortunately, this is sometimes not sufficient, for example, if there is a complex mapping between features and implementation assets. In this case, a feature's approximation would take into account non-functional properties of several implementation assets that however are also related to other features. Hence, a single approximation for a feature is not sufficient.

Fortunately, it is often an easy task to identify such feature interactions for some non-functional properties, such as footprint. We can use three different sources to identify feature interactions by: (a) using the mapping between domain features and implementation assets, (b) analyzing the source code (e.g., searching for nested #ifdefs as a common indication of simple implementation interactions), and (c) using domain knowledge. In our evaluation (Section 4), 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 asked a domain expert for the Linux kernel to identify feature interactions. In the case that 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. To approximate the influence of a feature interaction, we choose a suitable product, measure it, and compare this value against what we predict for this product; the difference of measurement and prediction is the influence of the interaction. We summarize the different approaches next.

Feature-wise measurement (*FW*). Feature-wise measurement means that we do not consider feature interactions. We only approximate the influence of features and nothing more. That is, 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 assets, such as in feature-oriented programming [16]. Still, in other cases, the results of the measurement are useful if we are interested in more or less 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 (*IW*). For the interaction-wise approach, we measure not only each feature in the feature model, but also all *known* feature interactions. For each interaction, we create a product that contains the features that interact. 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 accuracy. Especially, when an SPL contains a large number of features, domain knowledge can help to identify which of them interact. This approach results in a solid prediction base.

Pair-wise measurement (*PW*). With pair-wise measurement, we automatically detect all pair-wise feature interactions (i.e., an interaction in which exactly two features participate). That is, we measure $n(n-1)/2 + n$ products. The approach results in a substantially increased product set to measure, compared to the feature-wise measurement: $O(n)$ vs. $O(n^2)$. Note that there are also *k-wise measurements* possible to measure feature interactions in which exactly k features participate (kth order feature interaction). However, they usually require a large number of additional measurements, and it is only reasonable to measure them, when we know from domain knowledge that there are such interactions.

3.5. Tool support: SPL Conqueror

We developed the tool *SPL Conqueror*[8] to manage and automate the process of determining and measuring products and to approximate a feature's influence on non-functional properties [17]. The application of SPL Conqueror provides two major benefits compared to a manual approach. First, SPL Conqueror 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, it predicts a product's non-functional properties almost instantly.

SPL Conqueror maintains a feature model of the given SPL or customizable program. We use SPL Conqueror to determine valid configurations that have to be measured. To support any programming language and composition technique, we abstract from specific implementation techniques and consider a customizable program or SPL as a black box. All customizable programs have in common that they need to know the configuration in a special format and they have to be executed with this given configuration. The measurement process has three steps: (1) generate a configuration in the application-specific format, (2) trigger the generation or execution of the product, and (3) execute a user-defined measurement program, which executes the application, measures its non-functional properties, and writes the results in a XML format that can be read by SPL Conqueror.

Each step must be defined in SPL Conqueror such that the whole measurement process can be automated. SPL Conqueror needs to know in which format a configuration must be generated. For example for preprocessor-based customization, we generate a `flags.h` file, which contains preprocessor statements (e.g., `#define HAVE_ENCRYPTION` to compile Berkeley DB with encryption support). The remaining task (one time per SPL) is to manually include this `flags.h` file in the compilation process (e.g., in the

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

makefile). We support a wide array of customization techniques, but further techniques can be included if necessary:

- **Preprocessor**-based customization is supported via an automated generation of a user-defined header file, which includes the definition of preprocessor flags corresponding to selected features.
- **FeatureHouse** is a language-independent composition tool based on feature-oriented programming [18]. It stores configurations in an expression file, in which the selected features are listed.
- **AHEAD** is a composition tool suite for programs and other artifacts based on feature-oriented programming [19]. The configuration mechanism is similar to FeatureHouse.
- **FeatureC++** generates C++ programs based on feature-oriented programming [20] and uses also expression files with a slightly changed syntax.
- **Configuration files** are used in many programs, such as the Apache web server and the Rar compression library. To use this method with SPL Conqueror, a user specifies the name and path of the configuration file as well as how a selected customization option is defined by the corresponding program. Basically, we define the value for these key-value pairs in the feature model and generate the according configuration file.
- **Command-line options** represent a common way to customize a program. In this case, triggering the generation is the process of executing a program with the generated set of command-line parameters, which are also derived as key-value pairs from the feature model. With this technique, we measure only runtime properties.

4. Evaluation

Since our approach only predicts non-functional properties and cannot provide precise results, we evaluated accuracy of our approximations with two series of experiments. The first series of experiments address measurement and prediction of the property footprint (binary size of a program) and the second series of experiment concentrates on the main memory consumption. We use the goal-question-metric approach to evaluation goals and research questions [21].

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. We provide online the raw material of measurements for each program. We refer the interested reader to our Web site for more detailed information and for downloading our tool: http://fosd.de/SPLConqueror.

4.1. Experiment overview

We *analyze* the prediction of a product's *footprint* and *main-memory consumption for the purpose of* evaluation *with respect to* accuracy *from the point of view* of the vendor/customer in the context of SPLs and customizable programs. Since our approach is customer-centered, we calculate a fault rate of our prediction as the relative difference between predicted and actual property: $\frac{|actual-predicted|}{actual} * 100$. As discussed in Section 3.4, we developed three approaches to quantify the influence of a feature on a non-functional property. These approaches differ in accuracy and measurement effort. To rate these alternatives, we define the following research questions:

- **Q1:** What is the average fault rate of the feature-wise measurement approach?
- **Q2:** What is the average fault rate of the interaction-wise measurement approach (only for footprint experiments, see below)?
- **Q3:** What is the average fault rate of the pair-wise measurement approach?
- **Q4:** How do the three approaches scale in terms of number of measurements compared to a brute-force approach?

In the following, we describe the experimental design that we use to answer the research questions in both experimental series.

Experimental design. The experiment is divided into two steps: (i) creating the prediction model and (ii) predicting a sample product's footprint and main-memory consumption. In the first step, we build a prediction model, i.e., a feature model with approximations. Since we have three approaches, we build three different predictions models according to the description given in Section 3.1. That is, we measure the feature product and the delta product and compute the approximations according our formula given in Section 3.2. We report the measurement effort for creating these models in terms of number of measurements. For the interaction-wise prediction model, we measure the influence of known interactions. To determine interactions (when possible), we analyze the source code of the used SPLs to gain the knowledge which interactions exists. With a self-written tool, we detect nested `#ifdef` statements in SPLs based on preprocessors (cfg. Table 3). For compositional approaches, we search for the existence of documented structure interaction modules (e.g., derivatives [22]). For the pair-wise approach, we specify between each pair of features a feature interaction in our model and measure its influence: comparing measurement and prediction of a product with this interaction.

The second step is the main step of evaluating accuracies of predictions. We measure products for the properties footprint and main-memory consumption and compare these measurements against our predictions for the three different approaches. For large SPLs, in which the measurement of all products is not feasible in reasonable time, we choose 100 random products. For all other SPLs, we measure all products. 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.

Variables. The experiment has a single independent variable: configuration (shown in Table 2). A configuration is the set of selected features. Hence, it specifies the functionality of a program and subsequently affects the program's non-functional properties. Furthermore, a configuration is the basis for our prediction model. That is, we compute the prediction for a non-functional property by adding the influences of all features participating in the configuration.

The dependent variable fault rate describes the difference between the predicted and measured property of a program. To compute footprint, we need the two intermediate dependent variables measurement and prediction, whereas the configuration influences both variables. To set the fault rate into perspective for the footprint property, we provide the highest and lowest measured footprint in Table 3. Note that 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). In this work, when discussing about accuracy, we refer to the dependent variable accuracy, which we compute according to the formula given in Table 2.

Analysis procedure. We analyze the fault rate visually using *box plots* [23] and Quantile–Quantile (Q–Q) plots. A box plot plots the median as a line within the box and the quartiles as lines as the

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

Table 2
Description of experiment variables. Indep: independent; dep: dependent; *F*: number of features.

Name	Type	Class	Scale type	Unit	Range
Configuration	Indep.	Flags, etc.	Nominal	N/A	2^F
Fault rate	Dep.	$\frac{\lvert measurement - prediction \rvert}{measurement} * 100$	Ratio	%	$0–\infty$
Accuracy	Dep.	$100 - Fault\ rate$	Ratio	%	$-\infty$ to 100

Table 3
Overview of the SPLs used in the evaluation of footprint prediction.

Product line	Domain	Lang.	Techn.	Feat.	Products	LOC	Size in KB	
							Min[a]	Max[a]
LinkedList	Component	Java	Comp.	18	492	2595	4.4	10.5
Prevayler	Database	Java	CC	5	24	4030	87	169
ZipMe	Compression	Java	CC	8	104	4874	79	99
PKJab	Messenger	Java	Comp.	11	72	5016	39	161
SensorNetwork	Simulation	C++	Comp.	26	3240	7303	19	875
Violet	UML editor	Java	Comp.	100	10^{20}	19,379	6.3	185
Berkeley DB	Database	C	CC	8	256	209,682	1800	2740
SQLite	Database	C	CC	85	10^{23}	305,191	166	200
Linux kernel[b]	OS	C	CC	25	3×10^7	13,005,842	11,245	13,829

[a] 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.
[b] We use only a subset of 25 features of the Linux kernel selected by a domain expert. CC: conditional compilation, Comp.: composition approach.

boundary of the box, so that 50% of all measurements are inside the box. Whiskers describe the distribution of the remaining measurements (see Table 4).

A Q–Q plot is often used to compare two data distributions by plotting their quantiles against each other. That is, a point (x_i, y_i) on the plot refers to the *i*th data point of first distribution (*x*-coordinate) and to the *i*th data point of the second distribution (*y*-coordinate). If both distributions are similar then *x* is equal to *y* and the point lies on the diagonal line $y = x$. We use this plot to compare for the same configurations predicted*versus* measured properties. For a perfect prediction, all dots would lie on the diagonal line. We visualize each configuration as a dot on the plot.

In addition to the visual analysis, we compute the average fault rate (arithmetic mean) per SPL and per measurement technique (i.e., feature-wise, interaction-wise, and pair-wise). To this end, we compute for each sample product *P* the fault rate, sum them up, and divide the result by the number of measurements:

$$AvgFaultRate = \frac{1}{n}\sum_{i=1}^{n}\frac{|measurement(P_i) - prediction(P_i)|}{measurement(P_i)} * 100 \quad (1)$$

Furthermore, we compute the standard deviation in percent of all measurements per SPL and per measurement approach to quantify the scattering of predictions around the average fault rate.

4.2. Predicting footprint

First, we conducted an experiment to predict footprint (binary size) of a compiled product. We selected footprint for several reasons:

- Although it may appear trivial, footprint is quite difficult to predict. As for performance, feature interactions can have an immense effect: Cross-cutting 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 without measurement bias, which is important for a large-scale evaluation with multiple SPLs 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 measurements (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 still provide insight into the benefits of the interaction-wise approach on different SPLs.

Experimental material. As experimental units for our footprint prediction, we selected nine existing SPLs with very different characteristics to cover a broad spectrum of scenarios. In Table 3, we provide an overview of the SPLs: We selected SPLs of different sizes (2500 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). Although very different SPLs are used, the main technical commonality is that, in all SPLs, we can automatically generate and compile products for a given feature selection.

Features are either explicitly given by an already existing feature model (i.e., LinkedList, Prevaylor, ZipMe, PKJab, SensorNetwork, Violet) or derived from documentation. For SQLite and Berkeley, we analyze the documentation to identify features. The document specifies preprocessor flags to turn functions on and off. We extracted this information and created a corresponding feature model. The configuration is given as preprocessor flags to generate the according program.

From Linux, due to the huge configuration space, we considered only a subset of 25 features, selected as representative by a domain expert. The domain expert selected the following features, which cover both modular features, such as drivers, as well as cross-cutting features: DEBUG_BUGVERBOSE, INLINE_SPIN_LOCK, OPTIMIZE_INLINING,

Table 4
Fault rates in percent of footprint predictions of all SPLs using the approaches (Appr.): feature-wise (FW), interaction-wise (IW), pair-wise (PW), brute force (BF). Mean: mean fault rate of predictions, Std: standard deviation of predictions.

		Effort		Fault Rate (in %)	
Program	Appr.	# Measurements		Distribution	Mean±Std
LinkedList	FW	11	2 %		0.9± 0.9
	IW	13	3 %		0.7± 0.7
	PW	88	18 %		0.2± 0.2
	BF	492	100 %	−6 −4 −2 0 2 4 6	–
Prevayler	FW	5	21 %		0.1± 0.1
	IW	7	29 %		0± 0
	PW	17	70 %		0± 0
	BF	24	100 %	−6 −4 −2 0 2 4 6	–
ZipMe	FW	8	8 %		0.6± 0.6
	IW	10	10 %		0.2± 0.3
	PW	21	20 %		0.3± 0.5
	BF	104	100 %	−6 −4 −2 0 2 4 6	–
PKJab	FW	8	11 %		0± 0
	IW	8	11 %		0± 0
	PW	36	50 %		0± 0
	BF	72	100 %	−6 −4 −2 0 2 4 6	–
SNW	FW	26	1 %		0.5± 1
	IW	34	1 %		0.3± 0.5
	PW	252	8 %		0.2± 0.6
	BF	3 240	100 %	−6 −4 −2 0 2 4 6	–
Violet	FW	80	0 %		1 86.7± 34.4
	IW	2115	0 %		0± 0
	PW	5229	0 %		7 22.5± 362
	BF	10^{20}	100 %	−6 −4 −2 0 2 4 6	–
Berkeley	FW	9	4 %		1.9± 2.2
	IW	15	6 %		0.5± 0.8
	PW	33	13 %		0± 0
	BF	256	100 %	−6 −4 −2 0 2 4 6	–
SQLite	FW	85	0 %		0± 0
	IW	146	0 %		0± 0
	PW	3306	0 %		0.1± 0
	BF	10^{23}	100 %	−6 −4 −2 0 2 4 6	–
Linux	FW	25	0 %		0.4± 0.3
	IW	207	0 %		0.4± 0.3
	PW	326	0 %		0.3± 0.2
	BF	$3{\cdot}10^{7}$	100 %	−6 −4 −2 0 2 4 6	–

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. Among the 25 features were some features that we knew would interact by changing the footprint (as the evaluated non-functional property) of other features (e.g., OPTIMIZE_INLINING and CC_OPTIMIZE_FOR_SIZE both apply global optimizations).

Experiment procedure. We compiled all C-based programs with GCC and with -O2 optimization, which performs all compiler optimizations that do not involve a size-speed trade-off. Since footprint measurements are not influenced by the used hardware and we kept the same compiler for all measurements, we could parallelize the footprint measurements on three systems.

Deviations occurred in the experiment for SQLite. It was not possible to measure all variants that are valid with respect to the feature model. In these cases, we run into compilation errors, because of undocumented dependencies between features (compilation flags). However, we could perform all feature-wise measurements to approximate for each feature its influence on footprint. The effect of these errors is that we could neither measure each pair-wise interaction and nor each known interaction. That is, the FW prediction model is complete, but the other prediction models lack some interactions. Hence, our predictions might be more accurate if we could determine the influence of these

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

interactions. However, considering the huge configuration space of SQLite, the few failed configurations are neglectable.

4.2.1. Results

In Table 4, we summarize the results of our footprint measurements and predictions for all SPLs. To put the results into perspective, we additionally show the effort of a brute-force approach. Referring to our research question Q1 (what is the average fault rate of the feature-wise approach), our predictions are usually quite accurate even for the feature-wise approach. The fault rate is 21.3%, on average, for all SPLs and 5.5% without Violet; an accuracy, on average, of 78.3% and 94.5% respectively.

Predictions based on more measurements are even better. For Q2 (fault rate of interaction-wise approach), we identify a fault rate of 0.18%, on average, (i.e., an accuracy of 99.8% on average). However, we identified an exception of this rule for the Violet SPL, which we discuss below. That is, for pair-wise measurements (Q3) the fault rate raises to 80.5%, on average, over all SPLs, but is at 0.23% without Violet. Nevertheless, even predictions based on feature-wise measurements usually only exhibit a fault rate of a few percent, which can be reduced to less than one percent with more measurements (by defining feature interactions).

To answer Q4, we show in Table 4 the absolute number of measurements we performed to infer approximations of a feature's footprint and its percentage, compared to the number of all possible products (brute force). In summary, we needed to measure only a small subset of all products. Especially the feature-wise approach scales linearly with the number of features, and not with the number of products. Although the interaction-wise and pair-wise approach requires a higher number of measurements, the relative number is below one percent for large SPLs, such as Violet, Berkeley DB, Linux. Due to parallelization, footprint measurements can be completed in a feasible amount of time.

4.2.2. Discussion

Let us have a closer look at Berkeley DB, Violet, and the Linux kernel, because their results show interesting points for further investigations. Berkeley DB is 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 for this configuration. In Fig. 5, we show the results of our different approaches and emphasize the different prediction patterns of the three approaches. Although we only measured nine 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 footprint that is too low, because we did not measure these feature interactions that include additional 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.

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 six 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 0.5%. Thus, by measuring only 15 products of Berkeley DB, we can almost predict the footprint of all 256 products with a high accuracy (99.5% on average). 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 illustrated in Fig. 5.

For Violet, we observed the largest fault rates (cf. Table 4). The reason is a complex mapping between (some) features and implementation assets. That is, an individual feature may map to multiple implementation assets and a single implementation asset may be required by multiple features. Hence, when measuring such a feature, the corresponding product contains several implementation assets 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 assets is inaccurate, because we consider the footprint of the implementation assets multiple times. 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 in order to automatically define appropriate feature interactions. Hence, with additional measurements, we can easily reduce the fault rate to under 1% (see Table 4, Violet_IW).

For Linux, we expected large fault rates regarding the 100 randomly generated products, because all Linux features affect the size of other features. We were surprised that we still achieved a quite precise prediction even with the feature-wise approach, partially, because the features had a weaker effect than expected. We slightly improved the accuracy with the interaction-wise approach,

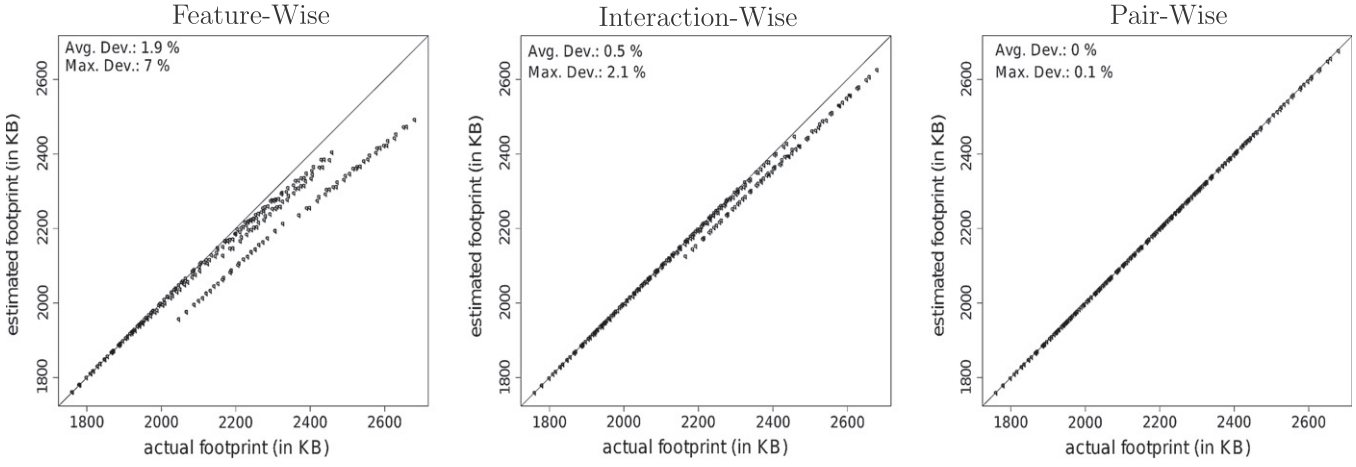


Fig. 5. Q–Q plot: Measured and predicted footprint in KB of Berkeley DB (compiled as static link library) using different approaches.

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

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_OPTIMIZE_FOR_SIZE*).

In summary, the first part of the evaluation shows that, at least for footprint, our prediction model provides good approximations of the actual non-functional property with few measurements. Next, we evaluate if our observations hold for a further non-functional property: main-memory consumption.

4.3. Predicting main-memory consumption

In the second series of experiment, we predict main-memory consumption (peak memory) of customizable programs. We selected main-memory consumption for the following reasons:

- Main-memory consumption is a property that emerges at runtime. That is, it is not a static property, such as footprint, but a dynamic one, which can substantially vary depending on which features and how many features are selected. We expect that predicting main-memory consumption is a challenging task, because we expect many feature interactions.
- Measurement bias for main-memory consumption occurs, but is usually very low. We wanted to explore how accurate predictions are when measurement bias occurs. That is, we evaluate if our approach is still feasible for non-functional properties in which measurements contain noise and may be subject to confounding influences.
- Finally, we chose main-memory consumption, because we are able to automate the measurement procedure for main-memory consumption performing a large number of measurements in a reasonable time.

Experimental material. To evaluate prediction of main-memory consumption, we initially selected seven existing sample systems (see Table 5). We use fresh sample programs, because we measure a different characteristic compared to footprint, which requires different measurement techniques. For example, footprint is not interesting for products that are customized via program parameters, because their sizes stay unchanged. Instead, we require a benchmark to measure the runtime behavior, which again is hard to define for a complex system (e.g., a single benchmark to measure the full Linux kernel). Since benchmarks are often used and important in the database domain, we selected Berkeley DB and SQLite. These systems represent SPLs for which products are generated using conditional compilation. Note that feature models of both SPLs differ to the models we used for footprint prediction, because we included features that are likely to change memory consumption (e.g., different page sizes) and excluded features that are not executed by the benchmark we used. As additional programs, we selected Curl, LLVM, x264, RAR, and Wget; programs that users can customize via program parameters. We selected these programs to demonstrate that our approach can be applied to both black-box SPLs and customizable programs (i.e., for which neither source code nor domain knowledge is available). We deliberately include many different domains, such as video encryption, compilers, and data transfer. Furthermore, these programs are well documented so that we could easily create feature models for them. Finally, all sample systems are industrial-strength real-world applications.

Table 5
Overview of sample programs used to predict main-memory consumption. CC:=conditional compilation; CP: command-line parameter. Vid. enc.: video encoding.

Program	Domain	Lang.	Techn.	Feat.	Products	LOC
Curl	Data transfer	C	CP	13	768	52,341
LLVM	Compiler	C	CP	11	1024	47,549
x264	vid. enc.	C	CP	16	1152	45,743
Wget	Data transfer	C	CP	16	5120	34,880
Berkeley DB	Database	C	CC	18	2560	209,682
SQLite	Database	C	CC	39	3,932,160	305,191
RAR	Compression	C++	CP	38	500,000	N/A

Experiment Procedure. In contrast to footprint, we face two additional challenges. First, when measuring a runtime property, we have to execute a benchmark application, which we discuss shortly. Second, we face measurement bias; that is, measuring the same product several times may result in different values. To overcome measurement bias, we measure each product three to ten times depending on the SPLs. We use this number of repetitions, because of two reasons. First, for all sample programs (with the exception of RAR), the standard deviation of measuring a single product is less than 1% of the arithmetic mean, which is sufficient for our studies. Second, increasing the number of measurements per product would substantially increase the time needed for this evaluation. Therefore, we decided to include more programs in the evaluation and reduce the number of repetitions per measurement instead of reducing the standard deviation to, say 0.1%, and measuring a single product hundred times. From these measurements, we compute the arithmetic mean and use it for our subsequent computations.

We use standard benchmarks (if available) for all sample programs, because self-developed benchmarks would bias the outcome of the measurements. Furthermore, standard benchmarks are created to simulate a common workload that is used in praxis, which is our intended goal. To measure the maximum required memory when performing a benchmark, we select the Linux standard program *time*. We used the following benchmarks:

- We use Oracle's standard benchmark for Berkeley DB. Similarly, we execute a benchmark script provided by SQLite to run a typical workload.
- LLVM is a modular compiler infrastructure. For our benchmarks, we use the *opt-tool* that provides different compile-time optimizations. We measure the main-memory LLVM needs to compile its standard test suite in several configurations (e.g., inline functions and combine redundant instructions).
- x264 is a command-line tool to encode video streams into H.264 and MPEG-4 AVC format. We measure the main-memory needed to encode the trailer of the cartoon *Sintel* (735 MB), often used as a standard benchmark for video-encoding projects.
- Curl and Wget are applications to transfer data over the Internet. As we found no standard benchmark, we download Apache's user manual which contains static HTML pages, CSS files and pictures. Due to the manual's folder structure, we can use several features of both programs (e.g., recursive download) to measure significant effects on memory consumption.

We measured main-memory consumption with the following systems, but measure all individual configurations of each sample program on the same system: AMD Athlon64 2.2 GHz, 2 GB RAM, Debian GNU/Linux 7; AMD Athlon64 Dual Core @2.0 GHz, 2 GB RAM, Debian GNU/Linux 7; Intel Core2 Quad @2.4 GHz, 8 GB RAM, Debian GNU/Linux 7.

Deviations. Early during our measurements of RAR, we identified huge measurement biases. Measurements of the same configuration deviated by 50–100 percent. This behavior was present with the Windows and Linux version of RAR. A possible reason may be that the algorithms in RAR are not deterministic. As a result, we were unable to approximate the influence of a single feature and thus discarded this program. For all other customizable programs, the measurement bias is less than one percent.

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

4.3.1. Results

In Table 6, we summarize the fault rates of predicting main-memory consumption for the sample programs. We observe that our approach of predicting memory consumption is feasible for many programs. The feature-wise approach has an average fault rate of 13.6% for all systems (Q1). For Wget and x264 it exhibits a high fault rate. Feature interactions have a crucial influence on the memory consumption. This is different to the footprint study in which, for most SPLs, fault rate is below 2%. This suggests that some non-functional properties are affected stronger by feature interactions than others, since we could largely exclude bias as a cause.

Using the pair-wise approach (Q3), we see that fault rate usually decreases (again with the exception of Wget and LLVM, which we discuss in the following); the average fault rate is 11% for all programs and 0.9% without Wget and LLVM. This means that our assumption of measuring the influence of each pair of features on a property improves accuracy of predictions. However, we have to keep in mind that this improvement requires additional measurements.

Referring to our research question Q4, we depict the number of required measurements for the two approaches in Table 6. The feature-wise approach usually requires to measure one percent of all products. For customizable programs with a large number of products (e.g., SQLite), we save even more measurements in relation to measuring all individual products. This demonstrates the scalability of our approach. Also for pair-wise measurements, we need to measure less than 10% of all products. Furthermore, for SQLite, we need to measure only 317 products which is 0.008% of all possible products.

4.3.2. Discussion

We observed a high fault rate for Wget and LLVM. A closer look at the distribution of the fault rates shows that the median of all fault rates is closer to zero than in the feature-wise approach (see boxplots in Table 6). That is, more predictions are accurate, but a few outlier predictions exhibit a very high fault rate (i.e., over 75%), which leads to a higher arithmetic mean fault rate. This high fault rate is similar to our observations for predicting footprint of Violet's products. In Violet, we could improve prediction accuracy significantly when using domain knowledge (i.e., the mapping from features to implementations). Unfortunately, we do not have the necessary knowledge for these two programs to show how this would affect accuracy.

We believe higher-order feature interactions are very plausible for LLVM; we hypothesize that they can be explained as follows. Each LLVM feature toggles a different optimization phase during compilation; each optimization might act differently on a code fragment, depending on how and whether previous optimizations have transformed it. For instance, function inlining enables other optimizations to operate on longer code fragments; depending on the size of code fragments and on the possibilities for optimizations, further optimization might trigger. To evaluate our assumptions, we used the documentation to manually define feature interactions. Overall, we measured 129 products of LLVM, which is 12.6% percent of all products. Although we are no domain experts, our predictions significantly improved to an average fault rate of 11%, which is an improvement of 22% compared to the pair-wise approach and 14% compared to the feature-wise approach.

We discuss feature interactions and general observations in Section 4.5.

4.4. Summary

We summarize the evaluation of predicting footprint and main-memory consumption in Table 7. We answer the research questions in terms of average fault rates (for Q1–Q3) and measurement effort relative to a brute-force approach (Q4). We can see that, for many sample programs, our approach provides a high accuracy of

Table 6
Fault rates in percent of predicting main-memory consumption of all SPLs using the approaches (Appr.): feature-wise (FW), pair-wise (PW), brute force (BF). Mean: arithmetic mean fault rate of predictions, Std: standard deviation of prediction fault rates.

		Effort		Fault Rate (in %)	
Program	Appr.	# Measurements		Distribution	Mean±Std
Curl	FW	11	1 %		0.4± 0.6
	PW	56	7 %		0.1± 0.2
	BF	768	100 %	-30 -10 0 10 20 30	–
LLVM	FW	11	1 %		28.8± 26.3
	PW	56	6 %		43.1± 59.5
	BF	1 024	100 %	-30 -10 0 10 20 30	–
x264	FW	12	1 %		27.9± 47.8
	PW	66	6 %		1.9± 5
	BF	1 152	100 %	-30 -10 0 10 20 30	–
Wget	FW	14	0 %		18.6± 24.9
	PW	91	2 %		20.6± 50.6
	BF	5 120	100 %	-30 -10 0 10 20 30	–
BerkeleyDB	FW	15	1 %		1.4± 1.3
	PW	98	4 %		1.4± 1.3
	BF	2 560	100 %	-30 -10 0 10 20 30	–
SQLite	FW	26	0 %		4.4± 3.8
	PW	317	0 %		0.3± 1.8
	BF	3 932 160	100 %	-30 -10 0 10 20 30	–

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

Table 7
Overview of research questions and experiment results. Effort means measurement effort. Q1–Q4 refer to the research questions given in the experiment description.

Experiment	Fault rate (%)			Q4: effort (%)		
	Q1	Q2	Q3	FW	IW	PW
Footprint	21.3	0.1	80	5.0	6.6	20
Without violet	5.5	0.2	0.2	5.8	7.5	22
Main memory	13.6	N/A	11	0.6	N/A	4.1
without LLVM & Wget	8.5	N/A	0.9	0.7	N/A	4.2

predictions. We also observe that feature interactions play a crucial role for the accuracy. This is the cause why our predictions for Violet, LLVM, and Wget are inaccurate; for that reason we provide the fault rates with and without these programs.

Measurement effort is relatively high for programs with a limited customizability compared to a brute-force approach, but absolutely small in terms of number of measurements. For example, although we measure 70% of all products of Prevayler for the pair-wise approach, these are only 17 measurements in total. The scalability is demonstrated when observing number of measurements for large SPLs, such as SQLite. Here, each approach requires less than 1% of all configurations and has a fault rate for footprint less than 1%.

Table 7 shows that more measurements provide more accurate predictions, with some exceptions. These exceptions, however, could be easily fixed with domain knowledge. A further result is that one of nine programs for footprint and two of six programs for main-memory consumption exhibit a high fault rate. Although the number of these programs is too small to know how often such exceptions occur, it indicates that in three-quarters of all programs a simple feature-wise technique including a pair-wise measurement is sufficient and, for the rest, a more sophisticated feature-interaction-detection approach or domain knowledge is needed to improve prediction accuracy.

4.5. Discussion

Next to the results of the evaluation, we discuss three important observations: using benchmarks, handling feature interactions, and number of measurements.

4.5.1. Benchmarks

For our second series of experiments, we had to execute benchmarks to measure the runtime behavior of a program. We measured main-memory consumption with standard benchmarks. So, we can predict memory consumption for other configurations only with respect to this particular workload. It is not our goal to predict non-functional properties completely independently of the workload. Instead, we provide an end-user solution in which customers perform only few measurements with their workload within their own environment. We argue that our predictions are more accurate regarding a product's properties for a specific application scenario than a synthetic benchmark, which uses a standardized workload and not one actually used.

4.5.2. Feature interactions

The results have shown that the existence of feature interactions that influence a non-functional property can cause high fault rates. We proposed two approaches to address this problem. First, we use domain knowledge to manually specify which features we expect to interact. Second, we incorporate all potential pair-wise interactions between pairs of features by measuring additional products. In most cases, both approaches yield more accurate predictions, because interactions with a large influence on a property are identified. However, when higher-order interactions (i.e., interactions between more than two features) exist and we are not aware of them, our predictions will be less accurate.

In a parallel line of research, we proposed an approach to automatically detect such non-functional feature interactions for performance [14]. The idea is to find which features interact at all and then search combinations of these interacting features that cause an observable feature interaction. Since finding such a combination requires a high effort, we propose three heuristics to find a sweet spot between measurement effort and accuracy of predictions. However, this is a different approach and goes beyond the scope of this paper.

4.5.3. 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 domain knowledge is available, we suggest always using it as it can significantly improve accuracy with only few additional measurements. Additionally, complex mappings or unknown feature interactions can cause large fault rates making predictions less accurate.

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

At this point, the measurements are often already sufficiently accurate to use them during product derivation – our initial goal – 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, if domain knowledge is not available, pair-wise measurements are a good strategy to increase accuracy of predictions at cost of an increased effort for measurements (from $O(n)$ to $O(n^2)$). We recommend it only 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 the trade-off between measurement effort and prediction fault rate in Fig. 6. 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 8000 features of the Linux kernel [7] (we considered only 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 1000 computers). In contrast, the feature-wise approach requires the measurement of only about 8000 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.

4.6. Threats to validity

4.6.1. Construct validity

A common threat to validity is that the experiment objects are not clearly defined. In our experiment, we want to know and compare the accuracy of the different measurement approaches. For this purpose, we defined Eq. (1), in which we specify how the fault rate is calculated. Using this definition, we can compare and rate the accuracy of the different approaches.

An incorrect choice of benchmarks represents another threat to construct validity. To minimize this threat, we use standard benchmarks delivered by vendors or used in the respective community if possible. Our aim was to not develop our own benchmark to avoid an uncommon application behavior and, therefore, flaw the exper-

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020

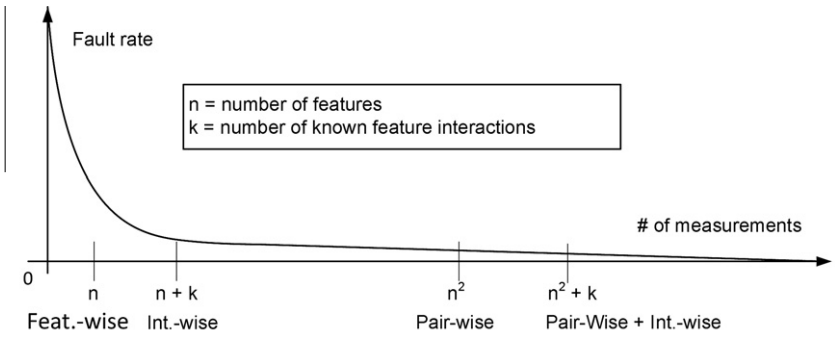


Fig. 6. Conceptual relation between number of measurements and fault rate of predictions.

iment design. In case of Wget, we could not find any standardized benchmark, which leaves room for a validity threat. We chose to download the Apache manual as a benchmark, because there is a large spectrum of common use cases (e.g., large files, many small files, nested folders, and pictures).

4.6.2. Conclusion validity

The reliability of measures strongly affects the conclusion validity of experiments. In our first experiment, we use footprint, defined as binary size of the generated program. For this measure, we can accurately determine the true size of programs using OS functions either by aggregating the size of all class files of a Java program or by determining the size of the executable in a case of a C/C++ program. Regarding main-memory consumption, we use the GNU/Linux tool *time* and depend therefore on its reliability. It measures the maximum resident set size of the process during its lifetime in KB. Since we measure only peak memory and not the average consumption over time, our measurements are not affected by page swapping of the OS or other influential factors that can affect validity of measurement. Furthermore, we repeated measurements and identified that the differences of several runs with the same configurations are below one percent.

4.6.3. Internal validity

For SPLs with many features and for the main-memory evaluation, we only sampled 100 products to compare prediction and measured property, because we cannot possibly generate and measure all products (exponential with the number of features) in reasonable time – this is exactly the motivation for our approach. We are aware of our evaluation leaving room for outliers, but we believe that 100 samples provide a reasonable number.

When measuring main-memory consumption, we have to deal with measurement bias. We repeated each measurement three to ten times and use the average of these measurements to compute a feature' influence and for our evaluation. With the exception of RAR, we observed only small deviations when measuring the same product multiple times in our six customizable programs (below one percent), which indicates that the measurement procedure is reliable.

In our footprint evaluation, we observed high fault rates for Violet. These fault rates were caused by complex mappings between features and implementation units. As we have shown, when this mapping is known, we can easily compute which configurations have to be additionally measured to achieve accurate predictions. Hence, the real problem is when we have a complex mapping between features and implementation units and this mapping is not known to us. So, the question is this the rule or exception in real-world applications? We believe that unknown complex mappings are the exception. Considering SPLs and customizable programs implemented with preprocessor, we almost always have a direct mapping between features and preprocessor flags, because we have to know these flags in order to generate the desired program. Since most of the customizable real-world programs today are based on preprocessors, this is already a strong argument. Furthermore, new programming paradigms, such as aspect-oriented and feature-oriented programming aim at a direct one-to-one mapping between features and implementation units. The general idea is to model features at the domain level and implement a feature accordingly [24]. Although Violet is implemented with feature-oriented programming, the mapping problem is an improper modeling or implementation of the functionality. Finally, complex mappings also occur for component-based software development. Here, however, we always need the mapping between features and components to be able to assemble this components together [25]. Hence, we believe that there is in most cases either the mapping known to be able to generate a program or the mapping is one-to-one.

4.6.4. External validity

Although we use a large variety of different SPLs, we are aware that the results of our evaluations are not automatically transferable to all other SPLs and all kinds of customizable programs. We selected real-world SPLs and customizable programs from different domains, having different sizes, and using varying implementation techniques. Our used SPLs have feature models with a typical structure and number of constraints (according to the criteria in [26]). We did not evaluate SPLs with an unusual, possibly degenerated feature model, which might influence the computation of the product set (cf. Section 3.3). Thus, we cannot generalize our results to such product lines.

We use non-functional properties in which measurements bias are below one percent. We cannot yet judge our approach for properties that exhibit different behaviors for the same workload within the same environment. That is, if the measurement of the same product yields different results also the predicted product can have heavily changing values of a non-functional property. We do not address this issue in this article and leave it for future work.

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

5. Related work

Many product-derivation approaches for SPLs have been proposed in the past [27–29]. However, most do not allow a user to specify non-functional constraints or to derive a product with de-

sired 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 and others presented a technique based on CSP solvers to find an optimal product [30]. 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 and others [31,32] 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.

A recent approach by Roos-Frantz and others focus also on quality attributes in SPLs [33]. In their work, they provide means to model quality attributes directly in a variability model, so that users can perform reasoning techniques to identify specification anomalies and to find configurations that satisfy given quality constraints. The verification is realized with a constraint programming solver within their tool FaMa-OVM.

In contrast to our approach, the approaches of Benavides and others, White and others, and Roos-Frantz and others do not consider the measurement and computation of a feature's non-functional properties. Hence, our approach can provide concrete quality data that is needed to parameterize their models. A combination of these techniques is feasible.

Only a few approaches apply measurements of non-functional properties to SPLs. Zubrow and Chastek proposed measures for SPLs that evaluate the development effort for an SPL [34]. Lopez-Herrejon and Apel express the complexity of an SPL in terms of variation points with a dedicated metric [35]. Apel and Beyer analyze the cohesion of features and the relationship to other ones at the level of source code [36]. Cohesion indicators may be used to enrich domain knowledge such that we can document possible feature interactions in our model.

An approach close to our work is the measurement of the binary size of an aspect-oriented SPL [37]. 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 [38]. The project focuses on techniques for tracing and adapting non-functional properties in component-based systems. The approach requires a dedicated 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 a wide variety of properties and address the exponential number of products that occur during the derivation process. Furthermore, we propose an implementation-independent 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. [12] 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 correlation 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 qualitative statement 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.

Feature Interactions. There is a large body of research on automated detection of feature interactions (e.g., see Nhlabatsi and others [39] and Calder and others [40] for surveys). Some approaches detect feature interactions using specifications of features [41–44]. In a parallel line of research, we propose to use heuristics to find feature-interactions automatically [14]. However, how feature interactions can be detected is not the focus of this paper; we only present two approaches that can be easily applied and build natural extensions to the pure feature-wise measurement.

6. Conclusion

To customize programs and derive products of a software product line, customers select features according to our requirements. However, it is often not known how a feature selection affects non-functional properties of the resulting program. We presented an approach to predict non-functional properties of customized and derived programs based on a feature selection without generating and measuring them. To this end, we approximate the influence of each feature on a non-functional property. The key idea is to produce two 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: feature-wise measurement, interaction-wise measurement, and pair-wise measurement. These approaches vary from linear to quadratic complexity in terms of the number of features in an SPL.

In a first series of experiments, 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 achieves a high accuracy with a small number of measurements.

In a second series of experiments, we demonstrated the generality of our approach with respect to other non-functional properties. We compared the predicted with actual maximum main-memory consumption in six different sample programs and SPLs. Although memory consumption is subject to measurement bias, we achieve an accuracy of 89% for pair-wise measurement.

In future work, we plan to generalize our approach such that we can predict non-functional properties of products with varying workloads.

Acknowledgments

We thank Martin Kuhlemann, Thomas Thüm, and Tillmann Rendel for helpful comments on previous drafts of the paper. Especially, we thank Janet Siegmund for advice on statistical evaluations. Norbert Siegmund's work is supported by the German Ministry of Education and Science (BMBF), No. 01IM10002B. The work of Marko Rosenmüller, Sven Apel, and Sergiy S. Kolesnikov is supported by the German Research Foundation (DFG), Project Nos. #SA 465/34-1 #AP 206/2, and #AP 206/4, #LE 912/13. Christian Kästner's and Paolo Giarrusso's work is supported by ERC Grant #203099.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
[2] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
[3] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[4] T. Henzinger, J. Sifakis, The discipline of embedded systems design, Computer 40 (10) (2007) 32–40.
[5] T. Henzinger, Two challenges in embedded systems design: predictability and robustness, Philosophical Transactions 366 (1881) (2008) 3727–3736.
[6] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, S. Ferber, Introducing PLA at Bosch gasoline systems: experiences and practices, in: Proceedings of the International Software Product Line Conference (SPLC), Springer, 2004, pp. 34–50.
[7] R. Lotufo, S. She, T. Berger, A. Wasowski, K. Czarnecki, Evolution of the Linux kernel variability model, in: Proceedings of the International Software Product Line Conference (SPLC), Springer, 2010, pp. 136–150.
[8] P. Toft, D. Coleman, J. Ohta, A cooperative model for cross-divisional product development for a software product line, in: Proceedings of the International Software Product Line Conference (SPLC), Kluwer Academic Publishers, 2000, pp. 111–132.
[9] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, S. Kolesnikov, Scalable prediction of non-functional properties in software product lines, in: Proceedings of International Software Product Line Conference (SPLC), IEEE, 2011, pp. 160–169.
[10] S.S. Stevens, On the theory of scales of measurement, Sciences 103 (2684) (1946) 677–680.
[11] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, G. Saake, Measuring non-functional properties in software product lines for product derivation, in: Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2008, pp. 187–194.
[12] J. Sincero, W. Schroder-Preikschat, O. Spinczyk, Approaching non-functional properties of software product lines: learning from products, in: Proceedings of Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2010, pp. 147–155.
[13] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, G. Saake, Integrated product line model for semi-automated product derivation using non-functional properties, in: Proceedings of Workshop on Variability Modelling of Software-intensive Systems (VaMoS), 2008, pp. 25–31.
[14] N. Siegmund, S.S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, G. Saake, Predicting performance via automated feature-interaction detection, in: Proceedings of International Conference on Software Engineering (ICSE), IEEE, 2012, pp.167–177.
[15] S. Segura, Automated analysis of feature models using atomic sets, in: Proceedings of the International Software Product Line Conference (SPLC), Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 201–207.
[16] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology (JOT) 8 (5) (2009) 49–84.
[17] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, G. Saake, SPL Conqueror: toward optimization of non-functional properties in software product lines, Software Quality Journal (2011) 1–31.
[18] S. Apel, C. Kästner, C. Lengauer, FeatureHouse: language-independent, automated software composition, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2009, pp. 221–231.
[19] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering (TSE) 30 (6) (2004) 355–371.
[20] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), Springer, 2005, pp. 125–140.
[21] V.R. Basili, Software modeling and measurement: The goal/question/metric paradigm, Tech. Rep. CS-TR-2956 (UMIACS-TR-92-96), 1992.
[22] J. Liu, D. Batory, C. Lengauer, Feature-oriented refactoring of legacy applications, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2006, pp. 112–121.
[23] T. Anderson, J. Finn, The New Statistical Analysis of Data, Springer, 1996.
[24] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology 8 (5) (2009) 49–84.
[25] G. Pour, Component-based software development approach: New opportunities and challenges, in: Technology of Object-Oriented Languages (TOOLS), 1998, pp. 376–383.
[26] T. Thüm, D. Batory, C. Kästner, Reasoning about edits to feature models, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2009, pp. 254–264.
[27] D. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of the International Software Product Line Conference (SPLC), Springer, 2005, pp. 7–20.
[28] M. Antkiewicz, K. Czarnecki, FeaturePlugin: feature modeling plug-in for eclipse, in: Eclipse '04: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Eechange, ACM Press, 2004, pp. 67–72.
[29] K. Czarnecki, S. Helsen, U.W. Eisenecker, Staged configuration using feature models, in: Proceedings of the International Software Product Line Conference (SPLC), Springer, 2004, pp. 266–283.
[30] D. Benavides, A. Ruiz-Cortés, P. Trinidad, Automated reasoning on feature models, in: Proceedings of Interactional Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2005, pp. 491–503.
[31] J. White, D.C. Schmidt, E. Wuchner, A. Nechypurenko, Automating product-line variant selection for mobile devices, in: Proceedings of the International Software Product Line Conference (SPLC), IEEE, 2007, pp. 129–140.
[32] J. White, B. Dougherty, D.C. Schmidt, Selecting highly optimal architectural feature sets with filtered Cartesian flattening, Journal of Systems and Software 82 (8) (2009) 1268–1284.
[33] F. Roos-Frantz, D. Benavides, A. Ruiz-Cortés, A. Heuer, K. Lauenroth, Quality-aware analysis in product line engineering with the orthogonal variability model, Software Quality Journal (2011) 1–47. http://dx.doi.org/10.1007/s11219-011-9156-5.
[34] Dave Zubrow, Gary Chastek, Measures for software product lines, Tech. Rep. CMU/SEI-2003-TN-031, Software Engineering Institute, 2003.
[35] R. Lopez-Herrejon, S. Apel, Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies, in: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), Springer, 2007, pp. 422–437.
[36] S. Apel, D. Beyer, Feature cohesion in software product lines: an exploratory study, in: Proceeding of the International Conference on Software Engineering (ICSE), ACM, 2011, pp. 421–430.
[37] F. Hunleth, R. Cytron, Footprint and feature management using aspect-oriented programming techniques, in: Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES), ACM, 2002, pp. 38–45.
[38] S. Göbel, C. Pohl, S. Röttger, S. Zschaler, The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects, in: Proceedings of the International Conference on Aspect-oriented software development (AOSD), ACM, 2004, pp. 74–82.
[39] A. Nhlabatsi, R. Laney, B. Nuseibeh, Feature interaction: the security threat from within software systems, Progress in Informatics 5 (2008) 75–89.
[40] M. Calder, M. Kolberg, E.H. Magill, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks and ISDN Systems 41 (2003) 115–141.
[41] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2011, pp. 372–375.
[42] H. Li, S. Krishnamurthi, K. Fisler, Verifying cross-cutting features as open systems, SIGSOFT Software Engineering Notes 27 (6) (2002) 89–98.
[43] K. Lauenroth, K. Pohl, S. Toehning, Model checking of domain artifacts in product line engineering, in: Proceedings of International Conference on Automated Software Engineering (ASE), IEEE, 2009, pp. 269–280.
[44] H. Post, C. Sinz, Configuration lifting: verification meets software configuration, in: International Conference on Automated Software Engineering (ASE), IEEE, 2008, pp. 347–350.

Please cite this article in press as: N. Siegmund et al., Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Inform. Softw. Technol. (2012), http://dx.doi.org/10.1016/j.infsof.2012.07.020