Family-Based Performance Measurement

Norbert Siegmund Alexander von Rhein Sven Apel
University of Passau, Germany

Abstract

Most contemporary programs are customizable. They provide many features that give rise to millions of program variants. Determining which feature selection yields an optimal performance is challenging, because of the exponential number of variants. Predicting the performance of a variant based on previous measurements proved successful, but induces a trade-off between the measurement effort and prediction accuracy. We propose the alternative approach of *family-based performance measurement*, to reduce the number of measurements required for identifying feature interactions and for obtaining accurate predictions. The key idea is to create a variant simulator (by translating compile-time variability to run-time variability) that can simulate the behavior of all program variants. We use it to measure performance of individual methods, trace methods to features, and infer feature interactions based on the call graph. We evaluate our approach by means of five feature-oriented programs. On average, we achieve accuracy of 98 %, with only a single measurement per customizable program. Observations show that our approach opens avenues of future research in different domains, such an feature-interaction detection and testing.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement techniques; D.2.13 [*Reusable Software*]: Domain engineering

Keywords Family-based Analysis, FeatureHouse, Performance Prediction

1. Introduction

Customizability is a critical success factor in software engineering. A customizable program gives rise to a *family* of *program variants* that can be tailored to the requirements of individual stakeholders. The customization process is simple: Select the *features* that satisfy your requirements, map the selected features to their respective implementation artifacts, and generate the corresponding program variant based on the artifacts. However, despite this simplicity, it is often unclear what the *best* feature selection is to satisfy all requirements, including non-functional requirements.

Especially, performance is critical. Often, a user wants to know the influence of a feature on performance of the generated program variant. For example, in the data-management domain: a customer has a specific workload for which she wants to determine the best feature selection of a given customizable data-management system (i.e., with the fastest response time). Considering the huge space of possible variants (33 optional and independent features give rise to a unique variant for each human on the planet), a brute-force approach that measures all variants does not scale.

Recent solutions to this problem aim at *predicting* a variant's performance based on the feature selection [17, 34]. This requires a performance model of the customizable program, which either must be trained via a number of measurements [11, 15, 17, 32] or inferred based on code analyses [24] and architectural knowledge [5, 38]. Both approaches have their benefits. Measurement-based performance models produce accurate performance predictions, but at the cost of time-consuming measurements. With code analysis, one does not have to actually measure individual program variants, reducing prediction accuracy, because the environment and other side-effects cannot be considered. Clearly, there is a trade-off between measurement effort and prediction accuracy [33].

We focus on measurement-based performance modeling. Our goal is twofold. First, we collect precise information on the performance behavior of individual features and their interactions based on the customer's workload. Second, we aim at minimizing the measurement effort to produce a performance model. To reach our goals, we propose *family-based performance measurement*. The key idea is not to measure individual program variants, but to execute a *variant simulator* that subsumes the individual behavior of all program variants.

Technically, family-based performance measurement consists of three steps: First, it generates a variant simulator for a given customizable program. In this step, it tracks which methods belong to which feature. Second, it executes the variant simulator with a given user-defined benchmark, logs the method execution times, and creates a corresponding call graph. Third, it analyzes the call graph to determine how much time has been spent within feature code and which of the paths in the call graph are visited only for a specific feature combination (which indicates a feature interaction). Then, times for each feature and feature interaction are aggregated in a performance model, in the form of a *choice-calculus* expression [14].

Family-based performance measurement relies on tracing information that is obtained when creating the variant simulator, by translating compile-time variability (e.g., feature modules [3]) to run-time variability (conditional execution). However, it can be applied also to programs that are variable at run time, but, in this case, tracing information needs to be provided externally.

We argue that family-based performance measurement is a promising alternative to the state of the art:

- It executes and measures only one variant simulator per customizable program and workload.
- It determines which features are actually used by the respective workload.
- It identifies performance-critical features and feature interactions of any order. [1]

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2517208.2517209

[1] The *order* of a feature interaction specifies how many features interact (minus one) [28]. For example, an interaction between two features is first-order.

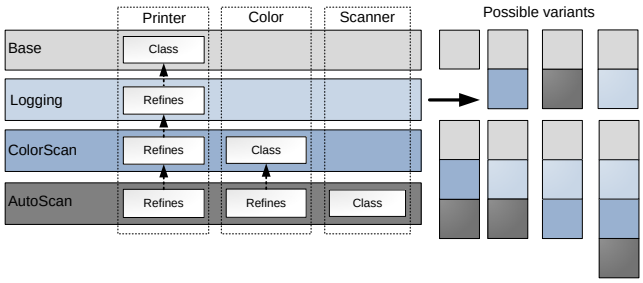

Figure 1. Decomposition of classes (vertical bars) with respect to features (horizontal layers) in the printer driver

Being a novel approach, it has some limitations (e.g., no multi-threading support), which we discuss in Section 5.2. Furthermore, we concentrate on feasibility in the evaluation. How accurate are predictions based on a variant simulator? What is the role of feature interactions, and how can they be detected, independently of their order? To answer these questions, we developed a tool chain for family-based performance measurement based on FEATUREHOUSE [3], in which features are implemented with statically composable feature modules. We evaluate our approach using five feature-oriented programs taken from a standard repository. Our experiments show that we achieve a prediction accuracy of 98 %, on average, while measuring only a small fraction of the variant space (we need only a single execution). We detected many feature interactions, up to the order of five, which cannot be found by current approaches that rely on pair-wise sampling [34].

2. Feature-oriented Programming

We demonstrate feasibility of our approach with *feature-oriented programming (FOP)* [6]. Programmers implement features in distinct, physically separated modules, called *feature modules*, whose composability facilitates customization. A feature module encapsulates source code otherwise scattered across different classes inside a single unit of composition. It can introduce new classes and extend existing classes (*class refinements*). A composer, such as FEATUREHOUSE [3], incrementally composes a set of given feature modules to obtain a final program variants.

In Figure 1, we show the layered design of a feature-oriented printer driver. Horizontal layers represent the feature modules of the driver, and vertical bars represent JAVA classes. Starting from a base implementation of a class, multiple refinements belonging to different features are applied. Refinements add new members to classes, such as methods and fields, or extend existing methods by overriding. For example, if a user selects feature *ColorScan*, class `Printer` is composed of the base implementation and a refinement to scan paper with colors.

In Figure 2, we show the code of the four feature modules of the printer driver. Technically, composition is implemented by superimposition [3]. Superimposition composes feature modules, based on nominal and structural similarity. It starts at the top of the hierarchical structure of the feature modules; if two program elements match in name and type they are merged; then, it proceeds recursively.

In our example, the composition starts with the feature modules *Base* and *Logging*. Both declare a class `Printer`. Since the names match, the content of the classes is superimposed, meaning that the composed class `Printer` contains the union of all their members. Then, superimposition proceeds with the individual class members. If two methods have the same signature in both feature modules, one method overrides the other method, in composition order (method `print` of feature *Logging* overrides method `print` of feature *Base*), a mechanism called *method refinement*. Keyword **original** refers to the overridden method. In our example, we extend the base implementation of `print` by adding a statement and calling the overridden

Feature: *Base*

```
1  class Printer {
2    public static void main(String[] args){
3      Printer p = new Printer();
4      p.print((Page)args[0]);}
5    public void print(Page p) {
6      ... // 2s
7    }
8    public Page scan() {
9      ... // 1s
10 }}
```

Feature: *Logging*

```
11 class Printer {
12   public void print(Page p) {
13     log("Execute printer job."); // 1s
14     original(p);}
15   public Page scan() {
16     ... // 0.5s
17     return original();}
18 }
```

Feature: *ColorScan*

```
19 class Printer {
20   public void print(Page p) {
21     ... // 5s
22     scan();
23     original(p);}
24   public Page scan() {
25     return original();}
26 }
```

Feature: *AutoScan*

```
1  class Printer {
2    // scans one page and prints it
3    public void print(Page p) {
4      ... // 1s
5      scan();
6      original(p);}
7  }
```

Figure 2. A feature-oriented printer driver (the class declarations of *Logging*, *ColorScan*, and *AutoScan* refine the corresponding declaration of *Base*)

Program variant: {*Base*, *Logging*, *ColorScan*, *AutoScan*}

```
1  class Printer {
2   public static void main(String[] args){
3     Printer p = new Printer();
4     p.print((Page)args[0]);}
5    public void print(Page p) {
6      ... // 1s by AutoScan
7      ... // 5s by ColorScan
8      log("Execute printer job."); // 1s by Logging
9      ... // 2s by Base
10     scan(); // by ColorScan
11     scan(); // by AutoScan
12   }
13   public Page scan() {
14     ... // 0s by ColorScan
15     ... // 0.5s by Logging
16     ... // 1s by Base
17 }}
```

Figure 3. Variant with the features *Base*, *Logging*, *Scan*, and *Copy*

method. The result of the composition of all features of our example is shown in Figure 3.

Note that, using the program variant of Figure 3, we can measure the performance of all features in combination (because the example does not include mutually exclusive features). But, this way, we cannot infer to what extent individual features contribute to the measured performance and how they interact. To quantify the influence of each feature, we would need to measure individual combinations, using a brute-force or sampling approach. As an alternative to this approach, we propose a family-based approach, which relies on a variant simulator.

Variant Simulator. A *variant simulator* subsumes the behavior of all variants of the customizable program. Technically, we generate

Variant simulator: {*Base*, *Logging*, *ColorScan*}

```
1  class Printer {
2    static boolean _HighColor_enabled;
3    @Feature(name="Base")
4    public static void main(String[] args){
5      Printer p = new Printer();
6      p.print((Page)args[0]);}
7    @Feature(name="Base")
8    public void print_role_Base(Page p) {
9      ...// 2s by Base
10   }
11   @Feature(name="Logging")
12   public void print_role_Logging(Page p) {
13     ...// 1s by Logging
14     print_role_Base(p);
15   }
16   @Feature(name="FeatureSwitch")
17   public void print_role_before_Logging(Page p) {
18     if (_Logging_enabled) {
19       print_role_Logging(p);
20     } else {
21       print_role_Base(p); }
22   }
23   @Feature(name="ColorScan")
24   public void print_role_ColorScan(Page p){
25     ...// 5s by Logging
26     print_role_before_Logging(p);
27   }
28   @Feature(name="FeatureSwitch")
29   public void print(Page p) {
30     if (_ColorScan_enabled) {
31       print_role_ColorScan(p);
32     } else {
33       print_role_before_Logging(p); }
34   }
35 }
```

Figure 4. Excerpt of the variant simulator for the features *Base*, *Logging*, and *ColorScan*

a variant simulator using *variability encoding* [2], which essentially translates compile-time variability to run-time variability. A variant simulator contains the code of all features as well as information on which feature combinations are valid. It invokes feature-specific behavior based on enabling and disabling guards around feature code. The guards are controlled by boolean variables that represent the presence and absence of individual features and that can be set at run time. For each method refinement, there is a new method introduced, called *feature switch*, that dispatches between the refined methods and the refining method. Finally, each method and each feature switch is annotated with information on their origin (i.e., the feature it belongs to).

Figure 4 shows the variant simulator of our example. Each method is annotated; the annotation in Line 3 states that this method belongs to feature *Base*; the annotation in Line 16 states that the method is a feature switch, included by variability encoding.

Note how the original methods have been renamed to implement method refinement. Each method refinement is translated to two new methods, one method that implements the actual refinement and one method (a feature switch) that dispatches between the refined method and the actual refinement. For example, `print_role_Logging` implements a method refinement (Line 12) and `print_role_before_Logging` a feature switch (Line 17). The base method does not need a feature switch (Line 8), and the last method refinement in chain receives the original name of the method (Line 29).

3. Family-based Measurement

Family-based measurement is inspired by related work on family-based analysis of software product lines [36], type checking [20, 23], static analysis [8, 27], model checking [4, 13], and deductive verification [37]. The idea is simple: Do not measure each program variant (or a subset thereof) individually, but execute a corresponding variant simulator to build a performance model. Clearly, there are several challenges that arise when using a variant simulator, such as how to handle alternative features and how to identify feature interactions. Hence, we begin with discussing requirements and goals of family-based measurement:

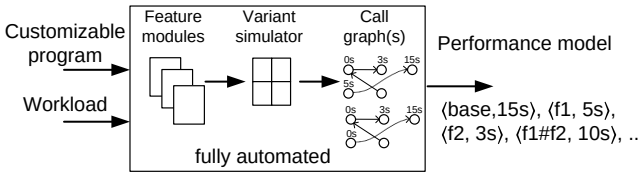

Figure 5. Process of family-based performance measurement

- When measuring performance, we need a specific workload or benchmark. For instance, to determine performance of a customizable database system, we need a representative workload that simulates the target application scenario of the fully-customized program variant. Such a workload may be standardized (e.g., TPC-H in the database domain) or given by the user to represent the intended use case.
- Beside the workload, we require a variability model (e.g., a feature model) to encode run-time variability properly. We reason only about the valid execution paths across features, and disregard invalid feature combinations. Furthermore, the variability model is used to optimize the performance model, for example, by stating which features are mandatory.
- The third requirement is measure all valid execution paths in the variant simulator. That is, we aim at full feature coverage according to the given workload.
- Finally, we want to fully automate the whole process, as illustrated in Figure 5. The inputs are the customizable program and the workload for which we generate the performance model.[2] Next, the variant simulator is generated and executed on the workload. From this measurement, we automatically extract execution times and feature interactions, and produce the corresponding performance model.

3.1 Performance Model

Our goal is to produce a performance model that captures the times that the variant simulator spent in the respective features' code. Clearly, the model should incorporate interactions among features, because feature interactions can have a substantial influence on performance [33]. The output is a variable performance model that computes, for any given feature selection, a performance value, which is used to predict the performance when actually executing the corresponding variant. We express the performance model using the *choice calculus* [14], in which each term holds a feature name and the time consumed by the feature when selected. The performance model Π for a customizable program p with the features $f_1, \ldots, f_n$, and a workload w is defined as follows:

$$\Pi_p^w = \sum_{i=1}^{n} \langle f_i,\, t_i^w \rangle \quad | \quad f_i \in p,\, t_i^w \in \mathbb{R} \tag{1}$$

For a workload w, we log the time t_i^w that a feature f_i contributes to the execution of p. A simple performance model of our running example is:

$$\langle Base,\, 5\text{s} \rangle + \langle Logging,\, 1.5\text{s} \rangle + \langle ColorScan,\, 5\text{s} \rangle$$

With this performance model, we can predict the execution time of all valid program variants by adding the corresponding performance values (considering feature *Base* is mandatory and *Logging* and *ColorScan* are optional):

[2] Note, in our scenario, we assume that the customizable program can be automatically generated.

Variant	Time in s
Base	5.0
Base, Logging	6.5
Base, ColorScan	10.0
Base, Logging, ColorScan	11.5

Note that we can have additional terms in the performance model representing feature interactions. A feature-interaction term represents the execution time that is consumed when the interacting features are present in a program variant, in addition to times of the individual features. If we identify a feature interaction between n unique features, we add a corresponding term to the performance model, denoted with operator # [7]:

$$\langle f_1 \# \ldots \# f_n,\ t_{1\#\ldots\#n} \rangle$$

For example, if the features *Logging* and *ColorScan* interact, we write *Logging#ColorScan*. Adding the feature-interaction term $\langle Logging\#ColorScan,\ 2\text{s} \rangle$ to the performance model of our example, would change the predicted performance of configuration {*Base, Logging, ColorScan*} from 11.5s to 13.5s, because we must add 2 seconds for program variants that contain both *Logging* and *ColorScan*.

3.2 Approach

The overall approach of family-based performance measurement consists of four steps:

1. Generate the variant simulator and weave code into the variant simulator to trace and measure feature code;
2. Execute the variant simulator to build a call graph that consists of methods with annotated feature names and measured execution times;
3. Identify feature interactions based on the call graph;
4. Aggregate performance values per feature to build a performance model;

In the remaining section, we explain the four steps in detail. For illustration, we use our running example, where *Base* is mandatory and *Logging*, *ColorScan*, and *AutoScan* are optional. Furthermore, we assume the times the features consume as specified in the comments of Figure 3.

3.2.1 Call Graph with Feature Annotations

The first step of family-based performance measurement is to measure the execution time of each individual feature. We need to extract the following information from the running variant simulator:

- Which methods are executed?
- To which features do these methods belong?
- What is their execution time?
- What is the sequence of method calls?[3]

To obtain this information, we weave code around each method when compiling the variant simulator, as shown in Algorithm 1. First, we measure the execution time of each method and log the time together with the name of the corresponding feature (or feature combination). To this end, when visiting a method, we automatically start a measurement (Line 9) and log the visit (Line 3). After returning from this method, we stop the measurement and log the result (Lines 11 and 12). Second, we build a call graph to trace the control flow, as illustrated in Figure 6. To this end, we log which feature calls which other feature, and we distinguish between a call due to a method refinement (via `original`) and a normal method call (Lines 4–7). By storing the name of the visited feature, we can keep track of that we visit all features (required by feature-coverage requirement), and we can identify feature interactions by analyzing the control flow across features.

By applying Algorithm 1 to the variant simulator of our running example, we obtain the call graph shown in Figure 6. Horizontal

[3] Currently, we do not consider multi threading.

Algorithm 1: Build call graph

```
   Data: CallGraph callGraph, Method parent
   Result: CallGraph callGraph
1  When executing Method method
2  begin
3  |  callGraph.add (method, method.featureName);
4  |  if method.isRefinement() then
5  |  |  callGraph.createRefineEdge(parent, method);
6  |  else
7  |  |  callGraph.createCallEdge(parent, method);
8  |  parent = method;
9  |  startTime = startMeasurement();
10 |  method.execute() ;   // continue method execution
11 |  measuredTime = endMeasurement()- startTime;
12 |  method.time = measuredTime;
13 |  parent = method.parent;
14 |  return callGraph;
15 end
```

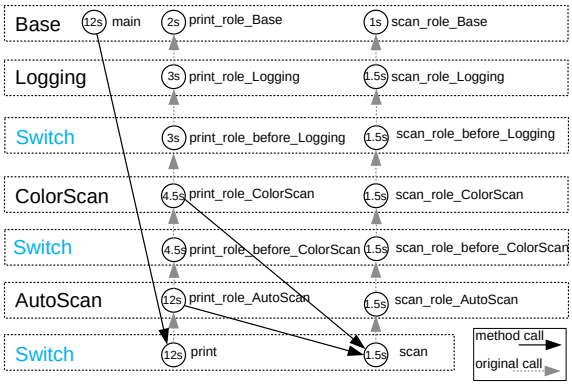

Figure 6. Call graph of our running example

layers denote features, nodes denote methods and method refinements, and edges denote method calls (both normal calls and calls due to **`original`**). For example, method `print_role_Base` contains the code contributed by feature *Base*; it is is refined three times, which results in three feature switches and three actual refinement implementations.

Although every node holds the time that the corresponding method has consumed, we do not know yet the actual time spent in a feature. This is because the measured time comprises also all execution times of methods that have been called from this method. For example, if we measure the execution time of method `print_role_ColorScan`, we measure 9.5s, which is only half the story, because we spent only 5s in this method, and the remaining 4.5s in `print_role_Logging`, `scan`, etc. Another important information that we do not have yet is on which features the execution of a method depends. To gain this information, we have to identify feature interactions, as we explain next.

3.2.2 Identifying Feature Interactions

Our approach to identify feature interactions is based on the observation that a call from one method to another always induces an interaction between the calling feature and the called feature. With "interaction", we mean that the called feature would never be executed, if the calling feature was not present.[4] For example, in Figure 6, there is an interaction between *Base* and *AutoScan*, as with-

[4] There are also features a code level, referred to as *derivatives* [28]. We treat derivatives similar to features. If the execution of interaction code leads to an alternative call sequence, we treat this similar to mutually exclusive features (see Section 3.3).

out *Base*, method `print_role_AutoScan` would not be executed. Likewise, there is an interaction between *AutoScan* and *ColorScan*, because `print_role_AutoScan` calls `scan`; *ColorScan*'s method refinement is executed only because *AutoScan* made this call.

In general, the execution of a method in a feature f_n may depend on a whole set $f_1, \ldots, f_{n-1}$ of other features. We define this set as the *prefix* $f_1\#...\#f_{n-1}$ of feature f_n in the corresponding interaction term. In our algorithm, prefixes grow when traversing the call graph.

In Algorithm 2, we show how to identify feature interactions. When entering a method, we determine whether this is due to a normal method call or a method refinement (Line 4). In the case of a normal call, we protocol a feature interaction between the calling feature and the called feature by adding the corresponding prefix to the call graph (Line 7). The prefixes are passed from the calling feature to the called feature and extended in each step (Lines 4–7).

Algorithm 2: Detect feature interactions

```
   Data: CallGraph callGraph, Method parent
   Result: CallGraph callGraph
1  // When entering Method method
2  begin
3      String currentFeature = method.getAnnotation();
4      if method.isRefinement() then
5          method.prefix = parent.prefix;
6      else
7          method.prefix
           = parent.prefix + currentFeature + "#";
8      callGraph.addPrefix(method, method.prefix);
9      ...
10     method.execute() ;   // continue method execution
11     ...
12     return callGraph;
13 end
```

There are two important facts to consider that are not obvious. First, an **`original`** call does not cause a feature interaction, because the execution of the method that has been refined does not depend on the feature that applied the refinement. Second, the execution of *all* refinements of a method depends on the feature that calls the method (via a normal method call).

In Figure 6, method `print` and all of its refinements depend on the presence of feature *Base*, which calls `print` from `main`. Based on this call, we identify four feature interactions: *Base#AutoScan*, *Base#ColorScan*, *Base#Logging*, and *Base#Base*. So, all features that refine method `print` obtain the prefix *Base#* to keep track of the interaction. Furthermore, all measured times will be included in the performance model with the prefix and the feature name. As the program execution continues, the prefix grows. For example, the two calls to method `scan` from *AutoScan* and *ColorScan* result in two sets of feature interactions:

{*Base#AutoScan#ColorScan*, *Base#AutoScan#Logging*, *Base#AutoScan#Base*}

{*Base#ColorScan#ColorScan*, *Base#ColorScan#Logging*, *Base#ColorScan#Base*}.

Simplifying Interaction Terms. The interaction terms created by Algorithm 2 are quite verbose, so we simplify them, according to a standard model of feature composition by Batory [7]. In particular, we define two rules for simplification. First, if a feature interacts with itself, then we can shrink the prefix accordingly, because the execution of the feature's methods depends on the presence of itself, which is always satisfied:

$$f_i\#f_i \longrightarrow f_i \qquad \text{(S-Ref)}$$

Second, mandatory features are always present, so a feature interaction between an optional and a mandatory feature depends only on the presence of the optional feature:

$$\frac{mandatory(f_i) \qquad prefix(f_i) = \emptyset}{f_i\#f_j \longrightarrow f_j} \qquad \text{(S-Mand)}$$

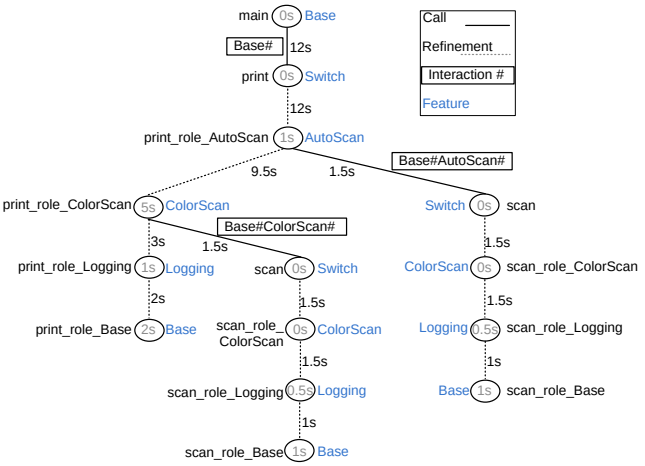

Figure 7. Reachability tree of our running example (with omitted feature switches). Times inside methods represent the time spent by the method itself. Times on edges represent actually measured times

Note that # is not commutative, and this rule does not apply if f_j is mandatory and f_i is optional, because, in this case, we execute a method of the mandatory feature f_j only if the optional feature f_i is present. Consequently, the time spent in this method influences performance only when both features are present.

When applying these simplifications to our running example, we obtain the following feature combinations in the performance model:

Base, *AutoScan*, *ColorScan*, *Logging*, *AutoScan#ColorScan*, *AutoScan#Logging*, *ColorScan#Logging*

Note that, even in this simple example, we identify three pair-wise feature interactions, indicating the potential of this approach to detect feature interactions. Next, we explain the remaining task to obtain a performance model: aggregating the execution times of individual methods.

3.2.3 Aggregate Execution Times

As said previously, when we log the time at the end of a method execution, we measured not only the time consumed by the method itself, but also the time spent in other methods it called. We have to subtract these times from the current measurement to compute the actual performance of this method.[5] For example, we measured an execution time of 9.5 seconds for method `print_role_ColorScan`, but we visited the method only for 5 seconds and spent the remaining time in the methods `print_role_Logging` (1 second), `print_role_Base` (2 seconds), and in four additional methods (1.5 seconds), as illustrated in Figure 7.[6]

Technically, we solve this problem by representing the call graph as a *reachability tree* similar to reachability trees in model checking [12]. Nodes in the tree represent executed methods and edges represent calls to other methods. Using this tree, we compute the actual time t_m spent in method *m* with the measured time T_m as follows:

$$t_m \quad = \quad T_m - \sum_{i=1}^{n} T_i \quad | \quad \forall i \in children(m) \qquad (2)$$

with *n* being the number of children of method *m* in the reachability tree.

[5] Since we use a call graph and not a dynamic call tree, we handle also recursive methods by representing it only a single time in the graph and aggregating the executing time of all executions.

[6] Note that we omitted some feature switches in Figure 7, for readability.

In Figure 7, we show the corresponding tree for our example. For each method, we know the corresponding feature, and after the execution of each method, we know the time spent. This time is annotated to the corresponding edges of the tree and subsumes the whole time spent below the current node. Using Equation 2, we compute for method `print_role_ColorScan` the time:

$$\begin{aligned} t_{print\_role\_ColorScan} &= T_{print\_role\_ColorScan} \\ &- \sum_{i=1}^{n} T_{print\_role\_ColorScan} \\ &= 9.5\text{s} - (3\text{s} + 1.5\text{s}) = 5\text{s} \end{aligned}$$

Next, we aggregate the method times t_i to build the performance model. For this task, we have to consider feature interactions. That is, it is not sufficient to take only the annotated feature names and sum the times up for each feature. If we do so, we lose the information under which condition a feature's method is executed (i.e., which features must be present to execute a certain method). The outcome would be a performance model that expresses the *maximal* execution time per feature, but not the actual execution time depending on a given feature selection. To solve this problem, we use the identified feature interactions encoded with the prefixes in order to create the correct performance terms for the performance model, as shown in Algorithm 3.

Algorithm 3: Build performance model

```
Data: CallGraph callGraph, PerformanceModel model
Result: PerformanceModel model
foreach Method m in callGraph do
    Term t = m.prefix + m.featureName ; // from Alg 2
    t.time = m.clearedTime ;          // from Equation 2
    model.addOrUpdate(t) ;            // add term to model
end
return model;
```

3.2.4 Putting the Pieces Together

So far, we have described the four steps of family-based performance measurement: (1) the algorithm to build the call graph, (2) the algorithm to identify feature interactions, (3) the simplification rules, and (4) the algorithm to build the performance model. Next, for a better overview, we explain how these pieces interplay using on the example of Figure 7.

We enter the program from method `main`, so all preceding methods in the call graph depend on the presence of *Base*. *Base#* is passed as the prefix when considering method `print`. Next, we reach method `print_role_AutoScan` and subtract from the measured 12s all times coming from the corresponding child nodes (9.5s and 1.5s).[7] As a result, we store the term $\langle Base\#AutoScan,\ 1\text{s}\rangle$ in our performance model ($12\text{s} - 9.5\text{s} - 1.5\text{s} = 1\text{s}$). Then, we continue with the left child, which is method `print_role_ColorScan` of feature *ColorScan*. Since this call is a method refinement, we do not extend the prefix. We measure 9.5s for this method, whereas 3s are consumed by the first child and 1.5s are consumed by the second child. Hence, we include the term $\langle Base\#ColorScan,\ 5\text{s}\rangle$ in our performance model ($9.5\text{s} - 3\text{s} - 1.5\text{s} = 5\text{s}$). When entering the second child of method `print_role_ColorScan` (which is method `scan`), we have to extend the current prefix by *ColorScan*. That is, we store for method `scan_role_ColorScan` the term $\langle Base\#ColorScan\#ColorScan,\ 0\text{s}\rangle$, and so on. The term amounts to 0s, because the child nodes consume the whole execution time (the method simply delegates the call).

[7] In practice, we first compute the performance terms of all child nodes, because the method is recursive. We explain the algorithm in the order of program execution to ease understanding.

By applying the simplification rules and by removing terms with a zero performance value, we obtain the following performance model:

$$\begin{aligned} & \langle Base,\ 2\text{s}\rangle + \langle Logging,\ 1\text{s}\rangle + \langle ColorScan,\ 6\text{s}\rangle \\ + \ & \langle AutoScan,\ 2\text{s}\rangle + \langle ColorScan\#Logging,\ 0.5\text{s}\rangle \\ + \ & \langle AutoScan\#Logging,\ 0.5\text{s}\rangle + \langle ColorScan\#Base,\ 1\text{s}\rangle \\ + \ & \langle AutoScan\#Base,\ 1\text{s}\rangle \end{aligned}$$

Finally, the performance model is then used as follows: Based on a valid feature selection, the model computes the performance value by removing all terms that do not contain any feature in the selection; then, the sum of the times of the remaining terms represents the performance value of the corresponding program variant.

3.3 Mutually Exclusive Features

So far, we did not consider the case in which features are mutually exclusive, say two alternative implementations of logging. While mutually exclusive features can be integrated into a single variant simulator, they can never be active in the same execution (incorporating the variability model, we consider only valid execution paths). A simple solution to this problem is to execute the variant simulator once per mutually exclusive alternative, and to compute the times consumed by the corresponding features. While this increases the number of measurements, this measurement applies only to sets of mutually exclusive features, not to all kinds of feature combinations, as in brute-force or sampling approaches [17]. Furthermore, since we assume that the method execution time is constant, we do not have to measure all combinations of alternative features. We discuss this issue further in Section 5.

4. Evaluation

The aim of our evaluation is to explore whether family-based performance measurement is feasible. To this end, we evaluate the accuracy of the predictions of our approach, compared to the actual performance that is measured. Furthermore, we compare the measurement effort of family-based measurement with state-of-the-art sample-based prediction approaches [34]. Finally, we discuss to what extent our approach can identify feature interactions.

Based on these goals, we formulate three research questions:

1. How accurate are the predictions of our approach?
2. What is the measurement effort of our approach compared to brute-force and state-of-the-art sampling approaches (i.e., feature-wise and pair-wise measurement)?
3. What kinds of feature interactions is our approach able to detect?

The complete tool chain (i.e., the extension of FeatureHouse and the measurement infrastructure) as well as all experimental data are publicly available: `http://fosd.de/Family`.

4.1 Experimental Setup

Next, we explain the customizable programs we used for our evaluation and the measurement procedure.

Subject Programs. As subject programs, we selected five feature-oriented programs from a public repository.[8] The selection criteria were that these programs can be processed by our tool chain (i.e., FEATUREHOUSE) and that they can run a benchmark automatically without user intervention, to obtain reproducible results. For each subject program, we executed either a standard benchmark (deployed with the program) or a typical workload (e.g., we analyzed a substantial code base with AJSTATS). We give an overview of subject programs in Table 1.

In the following, we describe each subject program briefly, including the changes we made to run a benchmark:

- AJSTATS is a customizable code-analysis tool for AspectJ programs. Depending on the configuration, it collects different statis-

[8] `http://fosd.de/fuji/`

Program	Domain	# Features	# Variants	LOC
AJStats	Code Analyzer	20	131 072	14 782
Elevator	Simulator	6	10	2 488
Email	E-mail Client	9	40	1 455
ZipMe	Compression Lib	14	10	5 355
MBench	Micro Benchmark	11	1 014	120

Table 1. Overview of the subject programs

tic, including the number of aspects, pointcuts, etc. As a workload, we analyzed Orbacus, a customizable CORBA implementation.
- Elevator models an elevator with varying optional conditions, such as weight limitations and priority service. It has been used before as a benchmark for (functional) feature-interaction detection [4, 29]. We extended the model by realistic timing information (e.g., how long different functions of the elevator need). As a workload, we use a scenario, applied by other researchers for verification.
- Much like Elevator, Email has been used before to verify the behavior of differently customized e-mail clients [4, 18]. Again, we included timing information into the model. We used a typical e-mail scenario as workload.
- ZipMe is a compression library that allows users to customize it by selecting different compression algorithms. As a benchmark, we use a file of 6 MB size generated by UIQ, a standard benchmark generator for compression algorithms.[9]
- We wrote a micro benchmark, called MBench, to test corner cases of family-based measurement. For example, it (a) calls methods from within a refined method, (b) it calls `original` within a loop, to simulate complex call graphs, (c) it defines methods in a feature that are called only by other features, etc.

We performed all measurements for a single customizable program exclusively either on a Intel Core2 Quad CPU, Win7 64Bit professional system with 8 GB RAM or on an Intel Core i7 2GHZ, Win7 64Bit professional with 4 GB RAM.

Experimental Procedure. We extended FeatureHouse such that it generates a proper variant simulator for a given feature-oriented program. It generates (i) feature-name annotations for each method and (ii) methods to switch between alternative features. We implemented the measurement infrastructure and construction of the call graph using AspectJ, by weaving advice around each annotated method.

The measurement process is simple: We run the variant simulator to log the execution times and to compute the performance model. Then, we create and measure all variants of the respective program and compute the error rate of our prediction as the relative difference between predicted and actually measured performance: $\frac{|actual-predicted|}{actual}$. For AJStats, we could not measure all variants in a reasonable time. Instead, we measured 30 256 randomly selected configurations, requiring two weeks of measurement.

4.2 Results

Prediction Accuracy and Measurement Effort. We present the measurement results in Table 2, including the error rate for each program, the distribution of the error rate using box plots, as well as mean and standard deviation. On average, the error rate is 1.7 %, which is well within the general measurement error of 2 %.[10] In short, to answer research question 1, we achieve very accurate predictions.

To learn about the feasibility of our approach, we quantify the measurement effort. Actually, we had to run each variant simulator only once, because no alternative features were present (research question 2). This is far less than in the brute-force approach and in all sample-based approaches (i.e., linear number of measurements for feature-wise sampling and quadratic for pair-wise sampling with respect to the number of features). Also note that the times presented in the table do not include repeating a measurement several times, which is, however, required to reduce measurement bias. Increasing the robustness and reliability of measurements would mean to multiply the times by a factor of 10 or higher. Even feature-wise measurement (as explained next) requires more measurements for our subject programs, but with a higher error rate, as it does not consider feature interactions at all [34].

[9] `http://mattmahoney.net/dc/uiq/`

[10] In our experiments, measuring the same program multiple times results in variations of up to 2 %. That is, an prediction error rate of below 2 % is as accurate as actually measuring the program variants.

Program	Time (BF)	Error Rate (in %) Distribution	$\mu\pm\sigma$
AJStats	15s (53d)		3.2± 2.5
Elevator	46s (220s)		0± 0
Email	60s (882s)		0.4± 0.5
ZipMe	40s (405s)		3.1± 3.0
MBench	50s (4h)		2.0± 1.3

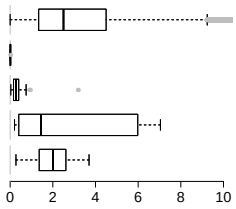

Table 2. Accuracy and measurement effort of the subject programs; BF refers to time needed for the brute-force approach. μ: arithmetic mean; σ: standard deviation

For illustration, we depict in Table 2 the times needed for family-based performance measurement in relation to the times needed to measure all variants (in brackets). The benefit of family-based measurement increases with the number of optional features (but decreases with the number of mutually exclusive features, as we discuss in Section 5).

Comparison against Sampling. Does family-based performance measurement outperform state-of-the-art sampling approaches: (a) feature-wise and (b) pair-wise measurement [34]? *Feature-wise measurement* samples the customizable program to quantify the influence of each feature on performance. To this end, we measure two variants that differ only in a single feature. By computing the delta of both measurements, we quantify the impact of this feature on performance. *Pair-wise measurement* aims at improving prediction accuracy of feature-wise measurement by keeping track of all pair-wise (i.e., first-order) feature interactions. To this end, we measure for each pair of features an additional variant and compare predicted against measured performance [32]. In Table 3, we compare family-based performance measurement with feature-wise and pair-wise measurement regarding prediction error rate and measurement effort. In all cases, family-based measurement outperforms or is, at least, equally accurate as feature-wise and pair-wise measurement. Considering the substantial reduction of measurement effort, we conclude that family-based performance measurement is a promising alternative to existing sampling approaches.

Feature Interactions. To answer research question 3, we analyzed the interaction terms of the generated performance models. In Figure 8, we show the distribution of interaction terms depending on their order. Terms with an order of zero represent the influence of a single feature on performance. With an increasing order, the terms are more difficult to detect.

In Elevator, we could not identify any feature interaction. Considering the perfect prediction accuracy, we assume that there are, in fact, no interactions present in this program. For all other programs, we identified a considerable number of feature interactions. More than 80 % of all terms represent feature interactions. For ZipMe, we found interactions up to an order of four and, for AJStats, up to an order of five. These results demonstrate that, in principle, there is no limit to detect interactions of arbitrary orders, which is not the case

Program	Appr.	Effort #M	Effort Time	Error Rate (in %) Distribution	Error Rate (in %) $\mu \pm \sigma$
AJSTATS	Family	1	15s		3.2 ± 2.5
	FW	18	486s		2.4 ± 2.1
	PW	115	2 425s		8.9 ± 6.9
ELEVATOR	Family	1	46s		0 ± 0
	FW	5	147s		0 ± 0
	PW	9	178s		0 ± 0
EMAIL	Family	1	60s		0.4 ± 0.5
	FW	7	149s		12.4 ± 13.3
	PW	27	586s		0 ± 0
ZIPME	Family	1	40s		3.1 ± 3.0
	FW	5	192s		4.3 ± 3.7
	PW	8	246s		1.8 ± 2.5
MBENCH	Family	1	50s		2.0 ± 1.3
	FW	11	205s		24.2 ± 18.7
	PW	67	1 885s		10.5 ± 11.9

Table 3. Comparison of family-based measurement with sampling approaches. #M: number of measurements; FW: feature-wise; PW: pair-wise; μ: arithmetic mean; σ: standard deviation

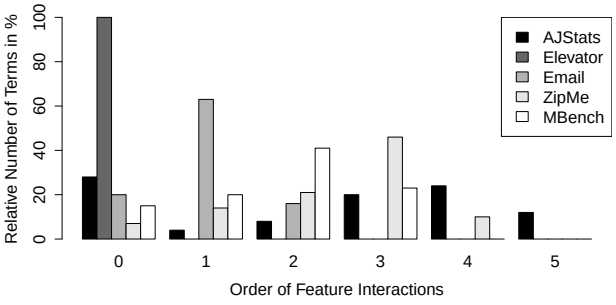

Figure 8. Number of identified performance terms for each customizable program. An order higher than zero indicates a performance term representing a feature interaction

for sample-based approaches (e.g., pair-wise is able to detect only first-order interactions).

In Figure 9, we show how much time we spent in determining interactions terms of different orders during the execution of the variant simulator. This illustrates the influence of feature interactions on the overall execution time. We see that the higher the interactions the less influence they have on the execution times. But, still, higher-order interactions exist! This finding confirms heuristics used in previous work [33].

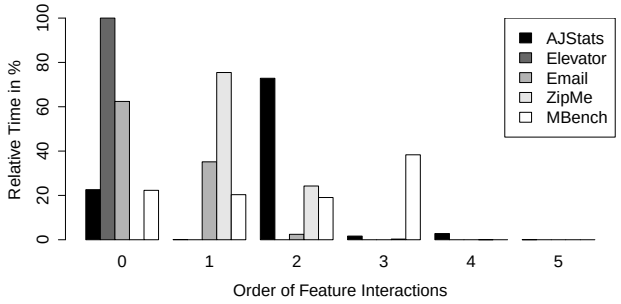

Figure 9. Relative time contributed by interaction terms of different orders

5. Discussion

5.1 Threats to Validity

A threat to external validity is certainly the limited selection of subject systems. This is due to the novelty of our approach and the fact that the tool chain still imposes certain technical requirements on the subject systems. Nevertheless, we argue that our experiments demonstrate, at least, the potential of our approach, which, of course, shall be evaluated with more rigor in the future.

Our approach detects feature interactions based on the control flow only, not considering the data flow. The presence of additional data-flow interactions may change the results, but not the big picture, because of the already quite accurate results. In further work, we plan to combine family-based performance measurement with family-based data-flow analysis [27].

Furthermore, the aspect that performs the measurement and that creates the call graph may induce an overhead in execution time. To reduce this effect, we (a) measured the overhead of the aspect when visiting a method and (b) how often we visit a method per feature. Based on this information, we subtract the overhead for each feature. Note that the overhead varies depending on the subject program.

Our conclusions can be transferred to other customization and implementation techniques only with care. Our approach traces the execution of a method to the feature it belongs to. For feature-oriented programs, this information is available at compile time, and can be transferred to run time. For related compile-time customization techniques, this information can be obtained in a similar way, for example, for aspect-oriented programming and component systems. Results on analyzing C programs with preprocessor directives demonstrates that even fine-grained, undisciplined annotation-based customization techniques can be used for tracing [22, 27]. Furthermore, there is ongoing work on translating preprocessor directives to conditional statements [31].

Finally, performance is important, but not the only non-functional property of interest. Instead of measuring the execution time, we can measure memory usage, energy consumption, throughput. We expect our approach to be applicable to all other non-functional properties that are measurable like performance.

5.2 Applicability and Limitations

As a novel approach, we face some limitations and focus on certain application scenarios. Next, we discuss each limitation and highlight how to overcome it in further research.

Mutually Exclusive Features. An issue that we largely set aside is how to handle mutually exclusive features. In our experiments, we needed only a single run per subject program, as they do not contain any mutually exclusive features. As explained in Section 3.3, with an increasing number of mutually exclusive features, the effort for family-based measurement (and also other sampling-based

approaches) increases, because alternative execution paths must be visited to cover all possible program paths. However, previous studies suggest that only a small fraction of features in real-world customizable programs are mutually exclusive [26]. Furthermore, in the presence of mutually exclusive features, our approach needs substantially fewer runs than state-of-the-art sampling-based approaches, as for them the number of variants increases not only with the number of mutually exclusive features, but also with the number of optional features. In fact, even in the worst case that all features are mutually exclusive, we require only a linear number of measurements (given our assumption that a method's execution time is constant), which is the lower bound of existing sampling approaches.

Program-Flow Analysis and Context Sensitivity. Our approach assumes that a method has a constant execution time no matter what optional feature is called before. The time can change only if the data used by the method depends on previously executed functions. Identifying such a change would require a program-flow analysis, which is a direction of future research. However, despite this limitation, we observed a high prediction accuracy. We assume that, in most cases, a changing workload has more significant influence on method execution times than feature selection. Furthermore, features in our evaluation usually implement a modular piece of functionality that does not depend on data manipulated by other features. For instance, applying an additional CRC check, signing an e-mail message, or additionally counting static members in an aspect in AJSTATS introduces a constant execution time for a static workload. Clearly, there may be cases in which feature execution times depend on other features' functionality and program-flow analysis is important, but our experiments suggest that focusing on control-flow can be often sufficient.

Determining the execution time of a single multi-threaded program is already challenging, but multi threading in the context of customizable programs imposes even more challenges, because we may encounter temporal dependencies among features. For instance, a feature may be executed in parallel to another feature. This means that depending on the feature selection, we may have an overlapping execution time. We invite the community to jointly tackle this problem.

Granularity and Implementation Techniques. Currently, we support only FeatureHouse-style programs, because we rely on the transformation of compile-time variability to run-time variability. An interesting question is whether we can apply these transformations also for other implementation techniques, supporting a finer granularity (e.g., #ifdefs in C). In a parallel line of research, we currently develop means to transform also #ifdef-based customizable programs to variant simulators, including type changes of variables and so forth. Furthermore, Kästner and others have shown that an automated transformation of an #ifdef-based program to a corresponding set of feature modules is possible [21]. An alternative way which makes our approach independent of the variant simulator is to make current profilers feature-aware. That is, if we can trace which statements belong to which feature at runtime, we can certainly create our call graph and subsequently our performance model.

To sum up, we expect that our assumptions (e.g., constant method execution time, no program-flow analysis) are usually met by feature-oriented programs, because features often implement coarse-grained and cohesive pieces of functionality. This picture may change when more fine-grained configuration options are considered as features, for instance, a feature that enables 64 bit support or doubles the precision of certain computations.

6. Related Work

The term 'family-based' stems from a recent classification of analysis techniques for product lines [36]. Currently, family-based analysis is performed mainly in the context type checking [20, 23], static analysis [8, 27], model checking [4, 13], and deductive verification [37]. We use a family-based approach for performance prediction, which imposes unique challenges, in particular, tracing and aggregating execution times, identifying and incorporating feature interactions, as well as making performance models themselves variable.

Performance Prediction. Chen and others [11] use a combined benchmarking and profiling approach to predict the performance of component-based applications. Based on a JAVA profiling tool, a performance model is constructed for application-server components. In contrast, we correlate the measurements to the feature selection and have to perform the measurement only a single time.

Guo and others predict performance of software product lines using classification and regression trees [17]. They measure multiple configurations and classify the performance results by means of selected and deselected features. When predicting performance of a configuration, they use the most similar feature selection for which they have already measured the corresponding configuration.

In our previous work, we proposed an approach to quantify the influence of each feature on performance [34]. To this end, we measure two variants that differ only in a single feature and interpret the delta of the measurements as the performance impact of the differing feature. We refined this approach to detect feature interactions by using several heuristics [33]. While improving prediction accuracy, this also increased the number of measurements.

Family-based measurement differs form all these approaches in that it requires a principally lower number of measurements, while it achieves slightly improved prediction accuracy in our experiments. However, family-based measurement is not a black-box approach. We require the source code of the program.

Feature Interactions. Feature-interaction detection has been addressed in a substantial body of previous research (see Calder et al. [10] for a comprehensive overview). There are measurement-based approaches, such as by Calder and Miller, who use pair-wise measurement based on linear temporal logic to detect feature interactions [9]. Another approach to identify feature interactions is iTree [35]. It aims at reducing the complexity of combinatorial testing of customizable programs by identifying sets of features that are most likely to interact, especially, for higher-order interactions. Other techniques can be classified as model-based detection, for example, reachability graphs [30] and model checking [2, 13]. In contrast to previous work, we concentrate on performance feature interactions and analyze the control flow of a variant simulator to identify interactions. The relation to other kinds of feature interactions shall be explored in further work.

Performance Profiling Performance profiling has a long tradition and has similar equal goals. *Calling context trees* are related to our annotated call graphs [1]. They allow us to handle different execution times of the same method depending on the calling context. We believe this technique can be combined with our approach. There are a number of approaches that use profiling data to create a performance model of a program [25]. For instance, Jovic and others analyze samplings of call stacks of deployed versions of a program to find performance bugs [19]. Grechanik and others propose to learn rules for the generation of workloads that reveal program paths with a degraded performance [16]. However, these approaches tackle workload variability rather than program variability.

7. Conclusion

Most of today's software systems are customizable in terms optional and alternative features. The selection of features can affect the performance of a program substantially, and often users need to customize a program to maximize performance or to satisfy certain performance requirements. Measuring the performance of all program variants is usually infeasible due to the combinatorial explosion of possible feature combinations. Instead of measuring all variants, we predict their performance based on the feature selection and with as few measurements as possible.

We proposed family-based performance measurement—an approach that uses a variant simulator to measure performance of each feature with only few runs. The variant simulator encodes compile-time variability at run time, such that it subsumes the behavior of all program variants. When executing the simulator, we log the execution time of each method, the features to which the methods belong, and the features from which the method calls came from. Based on this information, we analyze the call graph to determine interactions among features and to aggregate execution times for each method to produce a performance model. Then, we use this model to predict the performance of a certain feature selection.

We evaluated our approach by means of five customizable programs implemented with feature-oriented programming. The results show that our predictions reach an accuracy of 98 %, on average, while requiring only a single measurement per program. On a final note, this work is not intended to be complete. Instead, we want to open a door to further work on the analysis of feature interactions and the prediction and optimization of non-function properties.

Acknowledgments

The work of Siegmund is supported by the German ministry of education and science (BMBF), number 01IM10002B. The work of Apel and von Rhein is supported by the German Research Foundation (AP 206/2, AP 206/4, AP 206/5, and AP 206/7).

References

[1] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. PLDI*, pages 85–96. ACM, 1997.

[2] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. ASE*, pages 372–375. IEEE, 2011.

[3] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. ICSE*, pages 482–491. IEEE, 2013.

[5] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[7] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *Proc. GPCE*, pages 13–22. ACM, 2011.

[8] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.

[9] M. Calder and A. Miller. Feature interaction detection by pairwise analysis of LTL properties: A case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[10] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks and ISDN Systems*, 41:115–141, 2003.

[11] S. Chen, Y. Liu, I. Gorton, and A. Liu. Performance prediction of component-based applications. *Journal of Systems and Software*, 74 (1):35–43, 2005.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. ICSE*, pages 335–344. ACM, 2010.

[14] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology*, 21(1):1–27, 2011.

[15] C. Ghezzi and A. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology*, 55(3):508–524, 2013.

[16] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proc. ICSE*, pages 156–166. IEEE, 2012.

[17] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. ASE*. IEEE, 2013. to appear.

[18] R. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[19] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proc. OOPSLA*, pages 155–170. ACM, 2011.

[20] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proc. ASE*, pages 258–267. IEEE, 2008.

[21] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proc. GPCE*, pages 157–166, 2009.

[22] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. OOPSLA*, pages 805–824. ACM, 2011.

[23] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

[24] S. Kolesnikov, S. Apel, N. Siegmund, S. Sobernig, C. Kästner, and S. Senkaya. Predicting quality attributes of software product lines using software and network measures and sampling. In *Proc. VaMoS*, pages 25–29. ACM, 2013.

[25] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Automatic generation of efficient performance predictors for smartphone applications. In *Proc. USENIX*, pages 297–308. Usenix Association, 2013.

[26] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. ICSE*, pages 105–114. ACM, 2010.

[27] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. ESEC/FSE*. ACM, 2013.

[28] J. Liu, D. Batory, and C. Lengauer. Feature-oriented refactoring of legacy applications. In *Proc. ICSE*, pages 112–121. ACM, 2006.

[29] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[30] K. Pomakis and J. Atlee. Reachability analysis of feature interactions: A progress report. In *Proc. ISSTA*, pages 216–223. ACM, 1996.

[31] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. ASE*, pages 347–350. IEEE, 2008.

[32] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Proc. SPLC*, pages 160–169. IEEE, 2011.

[33] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

[34] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[35] C. Song, A. Porter, and J. Foster. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *Proc. ICSE*, pages 903–913. IEEE, 2012.

[36] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schäfer, and G. Saake. Analysis strategies for software product lines. Technical report, University of Magdeburg, Nb.: FIN-04-2012, 2012.

[37] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proc. GPCE*, pages 11–20. ACM, 2012.

[38] I. H. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques*. Elsevier, Morgan Kaufman, 2. edition, 2005.