draft -- to appear in ICSE, June 2012

Predicting Performance via Automated Feature-Interaction Detection

N. Siegmund*, S. S. Kolesnikov†, C. Kästner‡, S. Apel†, D. Batory§, M. Rosenmüller*, and G. Saake*
* *University of Magdeburg, Germany*
† *University of Passau, Germany*
‡ *Phillips University Marburg, Germany*
§ *University of Texas at Austin, USA*

***Abstract*—Customizable programs and program families provide user-selectable features to tailor a program to an application scenario. Knowing in advance which feature selection yields the best performance is difficult because a direct measurement of all possible feature combinations is infeasible. Our work aims at predicting program performance based on selected features. The challenge is predicting performance accurately when features interact. An interaction occurs when a feature combination has an unexpected influence on performance. We present a method that automatically detects performance feature interactions to improve prediction accuracy. To this end, we propose three heuristics to reduce the number of measurements required to detect interactions. Our evaluation consists of six real-world case studies from varying domains (e.g. databases, compression libraries, and web server) using different configuration techniques (e.g., configuration files and preprocessor flags). Results show, on average, a prediction accuracy of 95%.**

I. Introduction

There are many ways to customize a program. Commonly, a program uses command-line parameters, configuration files, etc. [1]. Another way is to derive tailor-made programs at compile-time using *software product line (SPL)* technology. In SPL engineering, stakeholders derive tailored programs by means of a program generator to satisfy their requirements. The generation process is based on *features* where a feature is a stakeholder-visible requirement on a program [2]. By mapping features to implementation units, a generator produces a program of an SPL based on a user's feature selection. In this paper, we use product-line terminology and call any customization option that stakeholders can select at compile-time or load-time a *feature* of a program.

Stakeholders can also customize programs to satisfy non-functional requirements. For example, a *database management system (DBMS)* is usually customized to achieve maximum performance when it is used on a server, but may be customized for low energy consumption when it is deployed on a battery-supplied system (e.g. on a smartphone or sensor node). Besides the target platform, other factors influence non-functional properties of a program. DBMS performance depends on the workload, cache size, page size, disk speed, reliability and security features, and so forth. Requirements can be customized by selecting a specific set of features. However, finding *the best* configuration efficiently is a hard task. There can be hundreds of features resulting in billions of configurations. Having just 33 optional and independent features yields a configuration for each human on the planet, and 320 optional features yields more configurations than there are estimated atoms in the universe. To find the configuration with the best performance for a specific workload requires an intelligent search; brute-force is infeasible.

We aim at *predicting* a configuration's non-functional properties for a specific workload based on selected features [3][4]. That is, we aggregate the influence of each selected feature on a non-functional property to compute the properties of a specific configuration. In this paper, we concentrate on performance predictions only. Unfortunately, the accuracy of performance predictions may be low, because many factors influence performance. Usually, a property, such as performance, is program-wide: it emerges from the presence and interplay of multiple features. For example, DBMS performance depends on whether a search index or encryption is used and how both features operate together. If we knew how the combined presence of two features influences performance, we could predict a configuration's performance more accurately. Two features interact if their simultaneous presence in a configuration leads to an unexpected behavior, whereas their individual presences do not [5][6]. We call feature interactions that affect performance *performance feature interactions (PFIs)*.

Today, developers can detect feature interactions only by analyzing the program (e.g. source code or control flow) or the specification of features [7]. These and similar approaches require substantial domain knowledge, exhaustive analysis capacities, or availability of source code to achieve the task. Furthermore, each implementation technique (e.g., configuration options, `#ifdef` statements, generators, components, and aspects) requires a specialized solution limited to only a subset of SPLs and customizable programs. To the best of our knowledge, there is no generally applicable approach that treats a customizable program as a black box and detects PFIsautomatically.

We improve the accuracy of predictions by means of (i) detecting which features interact and (ii) measuring to what extent. We aim to find the sweet spot between prediction accuracy, generality in terms of a black box approach, and measurement effort. The distinguishing property of our work is that we neither require domain knowledge, source-code nor complex program analysis methods and are not restricted to special implementation techniques, programming languages, or domains. Overall, we make the following contributions:

- An approach to efficient (in terms of measurement complexity), automated detection and quantification of PFIsto enable an accurate prediction of a configuration's performance.
- An improved tool, called *SPL Conqueror* [8], to perform measurements, detection of feature interactions and predictions in an automated manner.
- We use six customizable programs and SPLs from different domains, programming languages, and implementations to demonstrate the practicality and generality of our approach.
- 95% prediction accuracy when feature interactions are included, which is an 15% improvement over an approach that takes no interactions into account.

II. A Model of Feature Interactions

A recent model of feature interactions presents a fresh way to explain feature-based code synthesis [9]. Although code synthesis is *not* our interest, it is relevant to our model of feature-based performance. Both rely on the same ideas.

A program is built by composing features sequentially. If program p consists of features a, b, and c, we write:

$$p \quad = \quad a \cdot b \cdot c \tag{1}$$

where $\cdot$ denotes sequential composition.

A well-known fact about features is that they interact: features that work (or perform) one way in isolation may work (or perform) differently when other features are present. A classic example is call-forwarding and call-waiting. Call-forwarding (cf) allows a customer to specify another phone number to which calls are forwarded when a customer's line is busy. Call-waiting (cw) allows a call to be suspended when a second call is answered. If only one of cw or cf is present, the action to take when a call arrives when a line is busy is unambiguous. But when cw and cf are both present, a telephone system must make a decision whether to follow the action of cw, or the action of cf, or do something else. This decision is always present, it is specified separately in $cw\#cf$, called an *interaction*. If a telephone system is to have both call-waiting and call-forwarding, it must include cw, cf, and $cw\#cf$. More generally, if a program p contains features a and b, it must also include interaction $a\#b$.[1]

Basic mathematics encodes these ideas. When an architect wants features a and b, (s)he asks for their product ($\times$):

$$a \times b \quad = \quad (a\#b) \cdot a \cdot b \tag{2}$$

where ($\#$) is the interaction composition operation. That is, an architect not only wants features a and b, (s)he also wants the interaction ($a\#b$) that makes a and b work correctly together.

With this in mind, we revise the expression that synthesizes program p: now we take the product of features a, b, and c to produce a dot-composition of terms, one term for every feature or combination of features (interactions):[2]

$$\begin{aligned} p \quad &= \quad a \ \times \ b \ \times \ c \\ &= \quad a\#b\#c \ \cdot a\#c \cdot b\#c \cdot a\#b \cdot a \cdot b \cdot c \end{aligned} \tag{3}$$

To relate the above to performance synthesis, we state that performance of a feature composition $\Pi(a \cdot b)$ be the sum of their individual performances:[3]

$$\Pi(a \cdot b) = \Pi(a) + \Pi(b) \tag{4}$$

From equations (2)–(4), we produce an estimate of p's performance as follows:

$$\begin{aligned} \Pi(p) \quad &= \quad \Pi(a \times b \times c) \\ &= \quad \Pi(a \cdot b \cdot c \cdot a\#b \cdot a\#c \cdot b\#c \cdot a\#b\#c) \quad // \ (3) \\ &= \quad \Pi(a) + \Pi(b) + \Pi(c) + \Pi(a\#b\#c) \\ &\quad + \ \Pi(a\#b + \Pi(a\#c) + \Pi(b\#c) \quad // \ (4) \end{aligned}$$

To improve prediction accuracy, we need to determine the influence of an interaction on performance. We use a basic result that follows from (2) and (4). If we can measure a performance value for $\Pi(a)$ and $\Pi(b)$, we certainly can measure the value of $\Pi(a \times b)$. We therefore know the value of $\Pi(a\#b)$:

$$\Pi(a\#b) \quad = \quad \Pi(a \times b) - \Pi(a) - \Pi(b) \tag{5}$$

Here is the challenge: a product of n features yields $O(2^n)$ terms. We do not want to compute a value for each term, as this is infeasible. Further, Equation (5) assumes that we can measure performance influence of each feature in isolation, and be able to do this for all features. This is not always possible. We avoid both problems by composing multiple terms that cannot be separately measured into a single term, called *delta*. Given a base configuration C, we compute the

[1] Performance interactions rise at runtime, with consequences ranging from changing performance values, to unintended behavior such as crashes. $a\#b$ does not necessarily manifest itself as a code fragment, but we can still observe and measure its behavior.

[2] Commutativity and other axioms of sequential, interaction, and product composition are spelled out in [9]; their details beyond what is presented here are non-essential to this paper.

[3] As a limitation of this approach, we require additivity of a feature's performance.

impact of a feature A on C's performance as the delta for feature A as:[4]

$$\Delta A_C \quad = \quad \Pi(A \times C) - \Pi(C) \tag{6}$$

From equations (6) and (3), we derive an equivalent definition of ΔA_C:

$$\begin{aligned} \Delta A_C &= \Pi(A \times C) - \Pi(C) \quad // \ (6) \\ &= \Pi(A\#C) + \Pi(A) + \Pi(C) - \Pi(C) \quad // \ (3) \\ &= \Pi(A\#C) + \Pi(A) \end{aligned} \tag{7}$$

If C is an empty set of features, then $\Delta A_C = \Pi(A)$. Otherwise, term $\Pi(A\#C)$ expands to all interaction terms with feature A and any combination of features in C. If C is a product of i features, ΔA_C is a sum of $O(2^i)$ terms.

As we explain in subsequent sections, knowing ΔA_C for some C is often sufficient to predict the performance of programs that include A; we do not need to assign values to each of ΔA_C's terms. Herein lies the key to the efficiency and practicality of our approach.

III. Predicting Performance

We predict performance (and other non-functional properties) by measuring the influence of all features, its delta, in isolation and aggregating the deltas for all relevant features.[5] The main idea is that with few measurements (linear in the number of features), we can predict all configurations (exponential in the number of features). Although the approach is simple, we will see it yields surprisingly good results.

The general concept of quantifying the influence of each feature on performance is as follows: For each feature A, we find a *minimal* base configuration $min(A)$, such that $min(A)$ does not contain A and both $min(A)$ and $A \times min(A)$ are valid configurations. Subsequently, we determine each feature's delta as:

$$\Delta A_{min} \quad = \quad \Pi(A \times min(A)) \ - \ \Pi(min(A))$$

Let us illustrate the approach with an example. Consider the feature model in Figure 1 whose SPL has five features. The minimal configuration for each feature is:[6]

Feature A	$min(A)$
B	$\{\}$
I	B
T	B
E	B
D	$B \times E$

We need only five measurements to determine the influence of each feature (all values in our example measured in transactions per second):

$$\begin{aligned} \Delta B_{min} &= \Pi(B) - 0 &&= \ 100 \\ \Delta I_{min} &= \Pi(B \times I) - \Pi(B) &&= \ 15 \\ \Delta T_{min} &= \Pi(B \times T) - \Pi(B) &&= \ -10 \\ \Delta E_{min} &= \Pi(B \times E) - \Pi(B) &&= \ -20 \\ \Delta D_{min} &= \Pi(B \times E \times D) - \Pi(B \times E) &&= \ -10 \end{aligned}$$

To predict the performance of a configuration, we simply add the deltas of all relevant features. For example, for configuration $B \times T \times E$, we predict $\Delta B_\emptyset + \Delta T_B + \Delta E_B = 100 - 10 - 20 = 70$.

Unfortunately, this prediction scheme is inaccurate. As mentioned earlier, when measuring feature deltas, we might obtain very different results when using different base configurations. Consider Figure 1b, which computes the delta for feature T based on a different base configuration. Our first value, computed above, was $\Delta T_{min} = -10$, whereas our new measurement leads to $\Delta T'_{B\times I} = -5$. Consequently, predictions for the same configuration $B \times T \times E$ will differ when using ΔT_{min} or $\Delta T'_{B\times I}$. The difference is due to PFIs. Detecting and quantifying the influence of interactions allows us to overcome the differences among different deltas leading to consistent predictions. The question is: Which features interact that cause this discrepancy?

If we know that two features interact, we can improve our prediction by measuring the delta for the interaction. We measure a configuration C with both features A and B and compare the result to the deltas of both features:[7]

$$\begin{aligned} \Delta A\#B_C &= \Pi(A \times B \times C) - \Pi(A\#C) - \Pi(A) \\ &\quad - \Pi(B\#C) - \Pi(B) - \Pi(C) \\ &= \Pi(A \times B \times C) - \Delta A_C - \Delta B_C - \Pi(C) \\ &= \Pi(A\#B) + \Pi(A\#B\#C) \end{aligned} \tag{8}$$

Similar to the delta of a feature, the delta of interaction $A\#B$ includes not only a single term (i.e., the interaction of A with B), but also all (interaction) terms of A and B with features in C.

In Figure 1c, we illustrate such measurement for the interaction $I\#T$. Knowing the interaction's delta improves our predictions; in our example, it patches the value of ΔT_{min}. If more than two features interact (a.k.a., higher-order interactions [10]), we proceed in a similar way. The

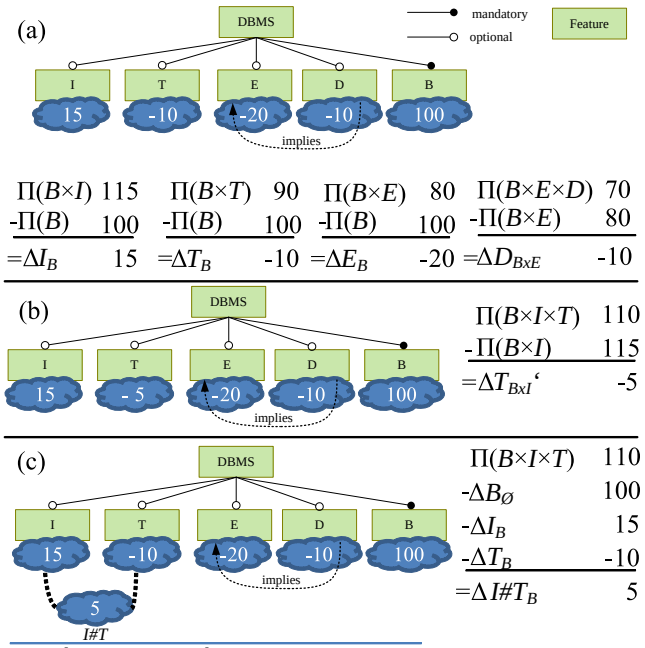
[4] A configuration is a set of features that yields to a valid program.

[5] In practice, features may not always be independent so that we cannot measure arbitrary configurations. We explored calculating deltas in the presence of complex domain dependencies previously [3]. It is out of scope of this paper.

[6] A feature model, a standard notation in SPL engineering [2], describes features and their relationships. Features are decomposed into a hierarchical structure and can be marked as mandatory, optional, or mutually exclusive. So to select a child feature, the parent feature must be selected. A configuration is valid if its feature selection fulfills all constraints of the feature model.

With constraints from feature models, in principle, there can be multiple minimal configurations (for example in the presence of mutually exclusive features). In this case we use any minimal configuration. Furthermore, we admit the empty or null program as minimal configuration when determining the performance of a root feature.

[7] Recap, if C is empty, then $\Delta A_\emptyset = \Pi(A)$, $\Delta B_\emptyset = \Pi(B)$, and $\Delta A\#B_\emptyset = \Pi(A\#B)$.



Legend: Features: *I* = Index, *T* = Transactions, *E* = Encryption, *D* = Decryption, *B* = Core Functionality (Base);
A#B Interaction between features *A* and *B*, selected when *A* and *B* are selected.
All measured values have the same unit of measurement: thousand transactions per second.

Figure 1. Measuring Deltas for Features and Interactions

challenge is how to find interactions that actually contribute to performance out of an exponential number of potential interactions.

IV. Automated Detection of Feature Interactions

Our goal is to identify feature interactions automatically from a small number of measurements. Our approach consists of two steps: (1) identifying features that participate in an interaction (called *interacting features*) and (2) finding combinations of features that actually cause a PFI. We use the setting from Figure 1 as our running example.

A. Detecting Interacting Features

First, we identify features that interact at all. The rationale is that the search space can be significantly reduced when we do so. Suppose an SPL has 16 features, but only 4 interact: We would have to look only at $2^4 = 16$ instead of $2^{16} = 65536$ configurations to detect interactions.

In the presence of interacting features, the delta for a feature A differs depending on which base configurations it was measured with. We say A is *not an interacting feature* if ΔA_C is the same for all possible base configurations C' (within some measurement accuracy). Conversely, if ΔA_C changes with different configurations of C, we know that A is interacting. We can express this as:

$$A \text{ is interacting} \Leftrightarrow \exists\, C \neq C' \text{ such that } \Delta A_C \neq \Delta A'_C$$

To avoid measuring ΔA_C for all configurations of C, we use a heuristic. We determine the deltas of A that are most likely to differ, because it is affected by the largest number of PFIs: We compare ΔA_{min}, the delta for the minimal configuration, with ΔA_{max}, a delta for a configuration with most features selected. Hence, let $max(A)$ and $A \times max(A)$ be two valid configurations, such that $max(A)$ does not contain A and is a maximal set of features, here called a *maximal configuration*. ΔA_{max} is their performance difference:

$$\Delta A_{max} \quad = \quad \Pi(A \times max(A)) \; - \; \Pi(max(A))$$

The rationale for $max(A)$ is that it maximizes the number of features that could interact with A. Consequently, if ΔA_{min} and ΔA_{max} are the same, then A does not interact with the features that are present in $max(A)$ but not in $min(A)$. Otherwise, A interacts with those features (we do not know yet with which features and by how much). Thus, with at most four measurements per feature (two for ΔA_{min} using $\Pi(A \times Min)$ and $\Pi(Min)$, and two for ΔA_{max} using $\Pi(A \times Max)$ and $\Pi(Max)$), we discover interacting features.[8]

In our running example, we determine the following maximal configurations and assume the following corresponding measurements:

Feature A	$max(A)$	$\Pi(max(A))$
B	$\emptyset$	0
I	$B \times T \times E \times D$	60
T	$B \times I \times E \times D$	85
E	$B \times I \times T$	110
D	$B \times I \times T \times E$	90

Note, $max(E)$ does not include D as D requires E for a valid configuration (Figure 1). With the additional measurements and $\Pi(B \times I \times T \times E \times D) = 80$, we compute the additional deltas as follows:

$$\begin{aligned}
\Delta I_{max} &= \Pi(I \times max(I)) - \Pi(max(I)) = 20 \\
\Delta T_{max} &= \Pi(T \times max(T)) - \Pi(max(T)) = -5 \\
\Delta E_{max} &= \Pi(E \times max(E)) - \Pi(max(E)) = -20 \\
\Delta D_{max} &= \Pi(D \times max(D)) - \Pi(max(D)) = -10
\end{aligned}$$

With these values, based on five measurements, we conclude that features I and T are interacting:

$$\begin{aligned}
\Delta I_{min} \neq \Delta I_{max} \quad & since \quad 15 \neq 20 \\
\Delta T_{min} \neq \Delta T_{max} \quad & since - 10 \neq -5 \\
\Delta E_{min} = \Delta E_{max} \quad & since - 20 = -20 \\
\Delta D_{min} = \Delta D_{max} \quad & since - 10 = -10
\end{aligned}$$

Note that if we find more than two interacting features, we have no information which features combination causes an interaction. This the goal of the next step.

[8]Of course, there is an obvious situation that we can not detect: When two interactions cancel each other (e.g., one has influence $+4$ and another one -4), we will not detect them. We have no evidence that this situation is common, but we are aware of its existence.

B. Identifying Feature Combinations Causing Interactions

After detecting all interacting features, we have to find the specific, valid combinations that actually have an influence on performance. Suppose we know that features A, B, and C are interacting features. We have to identify which of the following interactions have an influence on performance: $A\#B$, $A\#C$, $B\#C$, or $A\#B\#C$. Again, we do not want to measure all combinations (which is exponential in the number of interacting features).

We use three heuristics. Each makes an assumption under which it can detect additional interactions (thus improving performance prediction) with a few additional measurements. Our heuristics are based on the experience we gained during the manual analysis of feature interactions for the prediction of a program's binary footprint [3]. Other heuristics are based on assumptions we make due to analyses of source-code feature interactions and on related work (see Section VI). We explore in our evaluation whether our heuristics actually reduce measurement effort and improve accuracy of our predictions.

Auxiliary – Implication Graph: In all three heuristics, we need to reason about chains in an implication graph. An implication graph is a graph in which nodes represent features and directed edges denote implications between features. Using implications, we conclude that ΔA_{min} always includes the influence of all interactions with features implied by A (i.e., all features in A's implication chain). For example, if feature A always requires the presence of feature B, then we are implicitly quantified the influence of interaction $A\#B$ when computing ΔA_{min}. This mechanism reduces computation effort in all heuristics for hierarchically-deep feature models and for feature models with many constraints.

Heuristic 1 – Pair-Wise Interactions (PW): We assume that pair-wise (or first-order) interactions are the most common form of PFIs.

We justify this assumption as follows: Related research often uses a similar pair-wise approach: The software-test community often uses pair-wise testing to verify the correctness of programs [11][12]. Pair-wise testing was also applied successfully to test feature interactions in the communication domain [13] and to find bugs in SPL configurations [14]. Furthermore, analysis of variability in 40 SPLs showed that structural interactions are mostly between two features [15]; although structural interactions do not necessarily cause PFIs, we assume that this distribution also holds for performance, because the additional code may have some affect on performance.

Within the set of interacting features, we use this heuristic to locate pair-wise interactions first (as they are the most common). We only test for higher-order interactions with the remaining heuristics.

Heuristic 2 – Composition of Higher-Order Interactions (HO): We assume that higher-order PFIs(i.e., interactions among more than two features) can be predicted by analyzing already detected pair-wise interactions.

The rationale is if three features interact pair-wise in any combination, they may also participate in a 3-way interaction. That is, if we know $A\#B$ and $B\#C$ are non-zero interactions, then and only then will we check whether $A\#B\#C$ has an influence on performance. For example, if both $A\#B$ and $B\#C$ allocate 1 GB RAM, then it is likely that there is an interaction $A\#B\#C$ that results in a lower performance (because 2 GB RAM was allocated). We experienced this phenomenon in previous work on measuring and predicting footprint [3] when we manually identified footprint interactions. A different footprint may also indicate a possible impact on performance, because either functionality is added (for an increased footprint size) or is removed (decreased footprint). This added or removed functionality can cause notably performances changes.

Heuristic 3 – Hot-Spot Features (HS): Finally, we assume the existence of *hot-spot* features. We experienced that there are usually a few features that interact with many features and there are many features that interact only with few features. High coupling between features or many dependencies can impact the performance of the whole system, because both features strongly interact with each other at the implementation level.

These observations are analogous to observed coupling in feature-oriented and object-oriented software [16][17]. For example, Apel and Beyer showed that there are significant differences between individual features of an SPL and across individual SPLs regarding cohesion indicators [16], which in turn may indicate significant differences between feature interactions. These studies suggest that the links l (or dependencies) of modules and features follow *power-law distributions*, which says that the the probability $P(l)$ of linking to a given node is proportional to the number of existing links that the node has and the scaling exponent γ [18]: $P(l) \sim l^{-y}$. That is, if a node n has already many dependencies to other nodes than it is likely that a new dependency will incorporate n.

In previous work, we found that footprint feature interactions do not normally distribute over all features in an SPL [3]. Instead, there are hot-spot features that interact with (nearly) all other features (power-law distr.). For example, the Linux kernel feature $CC\_Optimize\_For\_Size$ interacts with all other features. That is, it changes the size of all other Linux modules. We anticipate the same distribution for PFIs.

Using this heuristic, we perform additional measurements to locate interactions of hot-spot features with other interacting features. Specifically, we attempt to locate third-order interactions for hot-spot features, because they seem to represent a performance-critical functionality in a program. We identify hot-spot features by analyzing the interactions we have found so far using our previous heuristics.

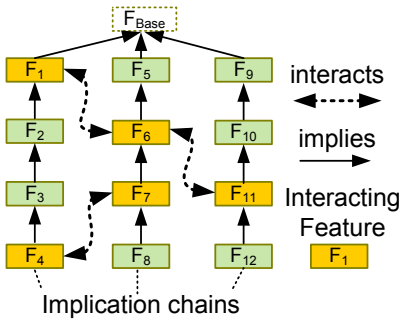


Figure 2. Implication chains with identified interacting features.

C. Realization

So far, we described a general approach to (1) detect interacting features and (2) to find feature combinations that cause interactions. Next, we detail how we implemented these techniques and heuristics in our tool.

As underlying data structure, we use an implication graph, as described earlier. We can easily generate this graph from a feature model using a SAT solver [19].

To locate pair-wise interactions (PW heuristic), we only consider pair-wise interactions between interacting features of different chains. We do not have to determine interactions of features belonging to the same implication chain, because the interaction is already included in ΔA_{min}. Furthermore, the order of the measurement is crucial. Our algorithm starts from the top of one implication chain and determines the influence of interactions with features of another implication chain also starting from the top. Afterwards, we continue with the next chain. For example, in Figure 2, the order we use to detect pair-wise interactions is $F_1\#F_6$, $F_1\#F_7$, $F_4\#F_6$, $F_4\#F_7$, $F_6\#F_{11}$, $F_7\#F_{11}$.

To identify whether two features A and B interact, we compare the measured performance $\Pi(A \times B)$ with the performance *prediction* of the same configuration that includes all known PFIsup to this time. If the result of $\Delta A\#B_C = \Pi(A \times B \times C) - \Delta A_C - \Delta B_C$ exceeds a threshold (e.g., we use the standard deviation of measurement bias as a threshold), we record $\Delta A\#B_C$.

Next, we search for higher-order interactions among features that interact in a pair-wise fashion (HO heuristic). Again, we perform additional measurements and compare them to the predicted results. For example, if we noticed that F_1 interacts with F_7 and F_7 interacts with F_{14}, we would examine whether interaction $F_1\#F_7\#F_{14}$ has an influence on performance.

Finally, search for further higher-order interactions involves hot-spot features (HS heuristic). Therefore, we count the number of interactions per feature identified so far. Next, we compute the average number of interactions per feature. We classify all features that interact more than the average as hot-spot features. With hot-spot features, we search (with the usual mechanism: additional measurements, comparing deltas) for interactions involving (1) a hot-spot feature, (2) a feature that already interacts with this hot-spot feature, and (3) an interacting feature that does not interact pair-wise with the hot-spot feature. We automated the entire process.

Project	Domain	Lang.	LOC	Features	Configs
Berkeley DB	Database	C	219,811	18	2560
Berkeley DB	Database	Java	42,596	32	400
Apache	Web Server	C	230,277	9	192
SQLite	Database	C	312,625	39	3,932,160
LLVM	Compiler	C++	47,549	11	1024
x264	Video Enc.	C	45,743	16	1152

Table I
OVERVIEW OF SAMPLE PROGRAMS USED IN OUR EVALUATION.

V. EVALUATION

Our approach to performance prediction is simple. But it is the simplicity that makes it practical. We demonstrate this with six real-world case studies.

The goal of our evaluation is to judge prediction accuracy and the utility of our heuristics. That is, we analyze how we detect PFIswith additional measurements and how detected PFIsimprove prediction accuracy. To that end, we compare predictions with actual performance measurements.

A. Evaluation Setup

We selected six existing programs (i.e., three customizable programs and three SPLs) with different characteristics to cover a broad spectrum of scenarios (see Table I) They are of different sizes (45 thousand to 300 thousand lines of code, 192 to millions of configurations), implemented in different languages (C, C++, and Java), and configured with varying mechanisms (such as conditional compilation, configuration files, and command-line options). Further, we selected industrial programs rather than self-developed programs.

The programs we selected have usually under 3 000 configurations. The reason is that we can actually measure *all* configurations of these programs in a reasonable time. Hence, even though it required quite some time, we could actually perform the brute-force approach and determine accuracy of our prediction over *all* configurations.

1) Setup: We measure *all* configurations of all programs. The exception is SQLite in which we measure only the configurations needed to detect PFIsand additionally 100 random configurations to evaluate the accuracy of predictions. We identified features in each case study and created a feature model describing their dependencies. We selected features that are likely to affect performance. For example, we did not select the feature Diagnostic in Berkeley DB C, because its functionality is never used in a benchmark. All feature models and measurement results are available online at: `http://fosd.de/SPLConqueror`

We automated the process of generating programs according to specific configurations and running the benchmark. Since Berkeley DB C and Java and SQLite use compile-time configuration, we compiled a new program for each

configuration that includes only the relevant features. For Apache, LLVM, and x264, we mapped the configuration to command-line parameters. We used five standard desktop computers for the measurements.[9]

We repeated each measurement between 5 to 20 times depending on the measurement bias. Since it is known that measurement bias can cause false interpretations and are difficult to control [20], especially for performance [21], we repeated measurements for a single configuration so that it reaches a confidence interval of 0.9. Usually the measurement bias varied for all case studies between 2 % and 10 %. We used these values to specify the threshold for detecting PFIs. We use the average of all measurements of a single configuration C as $\Pi(C)$.

2) Benchmarks: We use standard benchmarks either delivered by the vendor or often used in the community of the respective application. We did not develop our own benchmark to reduce measurement bias and uncommon performance behavior caused by flaws in benchmark designs.

Since performance predictions are especially important in the database domain, we list three DBMS SPLs: Berkeley DB's Java and C version (which differ significantly in their implementation and provided functionality) and SQLite. For each SPL, we use the benchmark delivered by the vendor. For example, we use Oracle's standard benchmark to measure the performance of Berkeley DB. The workload produce by the benchmarks is a sequences of operations typical of the given databases.

Furthermore, we selected the Apache Web server to measure its performance in different configurations. We used the tools *autobench* and *httperf* to produce the following workload: For each server configuration, we send 810 requests per second to a static HTML page (2 KB) provided by the server. After 60 seconds, we increase the request rate by 30 until 2700 requests per seconds are reached. After this process, we analyzed at which request rate the Web server could no longer respond or it produced connection errors.

LLVM is a modular compiler infrastructure. For our benchmarks, we use the *opt-tool* that provides different compile-time optimizations. We measure the time LLVM needs to compile its standard test suite in several configurations (i.e. with different optimizations like inline functions and combine redundant instructions enabled.) In this case, the workload is the program code from the LLVM test suite that has to be compiled with the enabled optimizations.

x264 is a command line tool to encode video streams into H.264 and MPEG-4 AVC format. The tool provides several options, such as parallel encoding on multiple processors. We measured the time needed to encode a video trailer (735 MB). The trailer is used by different video-encoding projects to put the comparable workload on the encoder.

[9]Intel Core 2 Quad CPU 2.66 GHZ, 4GB RAM, Vista 64Bit; AMD Athlon64 2.2GHz, 2GB RAM, Debian GNU/Linux 7; AMD Athlon64 Dual Core @2.0GHz, 2GB RAM, Debian GNU/Linux 7; Intel Core2 Quad @2.4GHz, 8GB RAM, Debian GNU/Linux 7. Each project was benchmarked on only one of these systems.

B. Results

We compute a fault rate of our prediction as the relative difference between predicted and actual property: $\frac{|actual-predicted|}{actual}$.

1) Accuracy and Effort: In Table II, we show the results of our six case studies: For each approach we depict the required number of measurements, the time needed for these measurements, and the number of identified interactions. Further, we show the distribution of the fault rate of our predictions with box plots. Finally, we show the mean fault rate of all predictions for a specific approach including the standard deviation of the prediction error. Note that with adding a new heuristic, we keep the previous heuristic working, because they are successively applied to a program.

With the feature-wise (FW) approach, we achieve already good predictions for programs in which interactions have not a substantial influence on performance. For example, our predictions have an average error rate of less than 8 % for all LLVM configurations. However, we usually have a high fault rate (e.g. over 44% for BerkeleyDB C version) when no interactions are considerd.

Using the pair-wise heuristic (PW) usually improves predictions significantly, because the majority of interactions are pair-wise. The benefit of implication chains compared to the common pair-wise measurement is that it reduces the number of measurements. For example, we require 81 measurements to detect first-order interactions for x264 (see Table II) which is 82 less than 163 that would be needed to measure all pairs of features.

With the higher-order (HO) heuristic, we achieve an average accuracy of 93.7% for all case studies. Interestingly, for LLVM, we could not find a feature combination that satisfies our preconditions. It is important to note that this heuristic usually doubles the number of measurements. For Apache the fault rate increases, because measurement bias over the determined threshold lead to a false detection of PFIs.

Finally, the hot-spot heuristic (HS) uses all three heuristics and thereby achieves the highest accuracy, which is 95.4% on average. Considering that the measurement bias for a single measurement of the case studies Apache, LLVM, and x264 is 5%, for SQLite it is 1%, and for Berkeley C and Java version it is 2% our predictions are as accurate as the bias of a single measurement.

2) Influence of Heuristics: Since all our heuristics are consecutively applied, we can visualize the trade-off between additional measurements and error rate of predictions in Figure 3. Dashed lines represent the average error rate of our predictions and straight lines depict the percentage of measurements compared to the maximum number of measurements. As expected, with an increasing number of

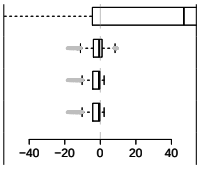
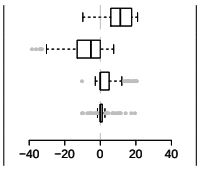
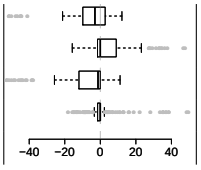
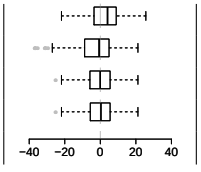
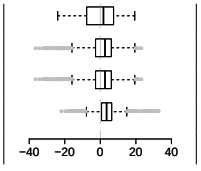
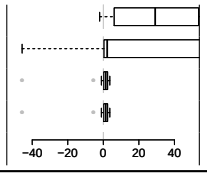
Project	Ap.	Effort: Measurements	Effort: Time (in h)	Interactions	Fault Rate (in %): Distribution	Fault Rate (in %): Mean±StdDev
Berkeley C	FW	15 (0.6 %)	2.5	0		44.1±42.3
	PW	139 (5.4 %)	23	14		3.9±5.3
	HO	160 (6.3 %)	26.6	22		2.8±3.7
	HS	164 (6.4 %)	27.3	22		2.8±3.7
	BF	2560 (100 %)	426	-		0±0
Berkeley JE	FW	10 (2.5 %)	8.4	0		17.7±19.6
	PW	48 (12 %)	40.2	24		8.5±9.6
	HO	16 (4 %)	97.1	51		3.8±5.7
	HS	162 (40.5 %)	137	69		1.7±3.5
	BF	400 (100 %)	335	-		0±0
Apache	FW	9 (4.7 %)	10	0		14.9±24.8
	PW	29 (15.1 %)	32.1	18		7.7±11.2
	HO	80 (41.7 %)	88.8	44		11.6±22.7
	HS	143 (74.5 %)	158.7	73		5.3±10.8
	BF	192 (100 %)	213	-		0±0
SQLite	FW	26 (0 %)	2.1	0		7.8±9.2
	PW	566 (0 %)	47	2		9.3±12.5
	HO	566 (0 %)	47	3		7.1±9.1
	HS	569 (0 %)	47.4	3		7±9
	BF	3,932,160 (100 %)	327,680			0±0
LLVM	FW	11 (1.1 %)	2.1	0		7.8±9
	PW	62 (6.1 %)	12.2	27		7.4±10.2
	HO	62 (6.1 %)	12.2	27		7.4±10.2
	HS	88 (8.6 %)	17.3	38		5.7±7
	BF	1024 (100 %)	202	-		0±0
x264	FW	12 (1 %)	2.3	0		29.6±22
	PW	81 (7 %)	15.7	13		17.9±27.2
	HO	89 (7.7 %)	17.2	17		5.1±15.1
	HS	89 (7.7 %)	17.2	17		5.1±15.1
	BF	1152 (100 %)	224	-		0±0

Table II
EVALUATION RESULTS FOR SIX CASE STUDIES. APPROACHES (AP.) ARE ENCODE AS: FEATURE-WISE (FW), PAIR-WISE (PW), HIGHER-ORDER (HO), HOT-SPOT (HS), BRUTE FORCE (BF). MEAN: MEAN FAULT RATE OF PREDICTIONS, STDDEV: STANDARD DEVIATION OF PREDICTIONS.

measurements the fault rate decreases. The results show that the relative number of measurements strongly differ to achieve the same accuracy for different programs. Further note that we require approximately 0.1% of measurements which demonstrates the scalability of our approach.

Implication Chains: Implication chains are beneficial for SPLs, because the average depth and number of constraints in an SPL's feature model are quite high [22]. On the other hand, no benefit is gained for programs customized via program parameters, because for these studies, the parameters do not depend on each other. Nevertheless, a recent study has shown the existence of such dependencies [23]. The reason is that the feature models of these programs are flat (i.e., with an average depth of one). Hence, the benefit of using implication chains strongly depends their length that can be created from a feature model.

Pair-Wise vs. Higher-Order Interactions: Look at the Apache case study (which is similar to others): A higher-order PFIusually improves predictions. We detected 18 first-order interactions and 55 higher-order interactions. Using the PW heuristc, we have some features that interact with many other features (e.g., KeepAlive) and other features that interact only with one or two features (e.g., ExtendedStatus). Although we do not use the hot-spot heuristic at this stage, we observe that some features are more likely to interact. Furthermore, we found two features (Base and InMemory) that do not interact. When applying the HO heuristic, all interacting features interact with each other. This indicates that it is important to consider higher-order PFIs.

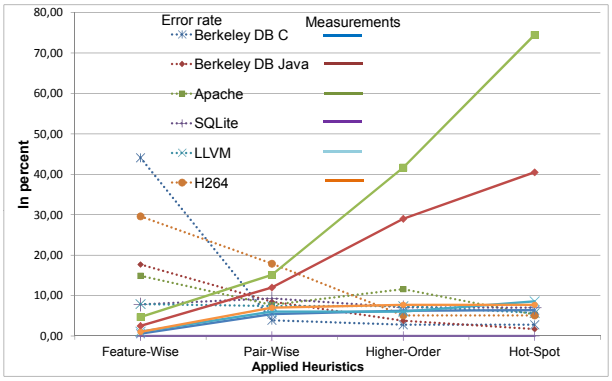


Figure 3. Comparing percentage of measurements (straight lines) with average error rates of predictions (dashed lines) for each heuristic.

C. Threats to Validity

Internal Validity: Regarding SQLite, we cannot measure all possible configurations in reasonable time. Hence, we only sampled 100 configurations to compare prediction and actual performance values. We are aware that this evaluation leaves room for outliers.

We are aware that measurement bias often appear and that they can cause false interpretations [20]. Since we aim to predict performance for a special workload, we do not have to vary benchmarks. Further, we repeated measurements to yield reach a confidence interval of 0.9.

External Validity: We aimed at increasing external validity by choosing programs from different domains with different configuration mechanisms and implemented with different programming languages. Furthermore, we used programs that are deployed and used in the real-world. Nevertheless, we are aware of that the results of our evaluations are not automatically transferable to all other configurable programs. In addition to our sample program selection, the strong and exhaustive evaluation (over 60 days of measurement with 5 computers) indicate that our heuristics hold for many practical application scenarios.

D. Discussion

Although we use a simplistic performance model, we demonstrated that the approach is feasible. With an average accuracy of 95% we achieve predictions that even stay in the range of the observed measurement bias for the case studies. It is important to note that we experienced large differences in accuracy when we changed the threshold at which a PFIis detected. Having a too small threshold causes many false detections of interactions. The fault rate increases, because we sum the influence of measurement bias instead of the influence of interactions.

We observed that we need a relatively large number of measurements when many alternative features exist compared to independent features. The reason is that having many alternative features limit the number of valid configurations substantially. For example, we can only generate 400 configurations in Berkeley DB Java, though there are 32 features present. This number is below a quadratic complexity. Hence, already the detection of interacting features require a relatively large number of measurements. However, having programs with a small number of valid configurations make a brute-force approach feasible, which is not our intended application scenario.

Furthermore, we do not consider performance behavior of a program independently of the workload. We can make accurate statements for any configuration given a *specific* workload. That is, we address end-users that have a certain application scenario in mind but do not know which configuration performs best. Measurements can be performed on a live system in a real environment, which produces more suitable performance predictions than standard benchmark results in a synthetic environment. For a new customer or a new workload, we have to repeat the measurements. We believe that many interactions still exist, though the values of the interactions will change. This however means that we may save additional measurements for new customers, since we already know which features interact.

VI. Related Work

A. Performance Prediction

There are several approaches that aim at predicting performance of a customizable program or an SPL. Abdelaziz et al. provide an overview of component-based prediction approaches [24]. Typically, the approaches belong to one of three categories: model-based, measurement-based (as we use in this paper), and mixed.

Model-based: Model-based predictions are common [25][26]. For example, linear and multiple regression model the relationship between input parameters (features) and a measurement output. Based on such a regression model, different estimation methods (e.g., ordinary least squares) can be used to predict the performance for specific input parameters. Bayesian (or belief) networks are used to model dependencies between variables in the network [27]. They are often used to learn causal relationships and hence may be applicable to detect PFIs. Furthermore, machine-learning approaches can be used to find the correlation between a configuration and a measurement, e.g., *canonical correlation analysis* [28], which uses dataset pairs to identify those linear combinations of variables with the best correlation. *Principal component analysis* [29] finds dimensions of maximal variance in a dataset that can also be used to detect PFIs. Ganapathi et al. provides an analysis for different machine-learning approaches in the context of performance prediction of database queries [30].

The feasibility of these approaches heavily depends on the application scenario and program to be analyzed. Our work differs in that it offers a general way to produce accurate

predictions independent of the application scenario and uses heuristics to significantly reduce measurement effort.

Krogmann et al. [31] combine monitoring data, genetic programming, and reverse engineering to reduce the number of measurements in order to create a platform-independent behavioral model of components. For a platform-specific prediction, they use bytecode-benchmark results of concrete systems to parameterize the behavior model. We predict the performance independently of the used programming language and availability of bytecode.

Happe et al. present a compositional reasoning approach, based on the *Palladio component model* [32]. The idea is that each component specifies its resource demands and predicted execution time in a global repository. The approach is applicable only to component-based programs, whereas we use a single approach for all customizable programs.

Also in this vein, MDE-based work uses feature models to customize or synthesize performance models (e.g. [33]). Feature models may themselves include performance models. This line of research requires up-front and detailed knowledge of domain-specific performance modeling, where tuning predictions for accuracy can be difficult. Our approach avoids these problems by directly measuring performance.

Measurement-based: Sincero et al. [4] predict a configurations's non-functional properties with a knowledge base consisting of measurements of already produced and measured configurations. They aim at finding a correlation between feature selection and measurement. This way, they can provide qualitative information about how a feature influences a non-functional property during configuration. In contrast to our approach, they do not actually predict a value quantitatively. Furthermore, they do not provide means to detect PFIs.

Chen et al. [34] use a combined benchmarking and profiling approach to predict the performance of component-based applications. Based on a benchmark and a Java profiling tool, a performance prediction model is constructed for application server components. In contrast, we correlate the measurements to the configuration, and measure only those configurations from which we expect to detect PFIs.

Abdelaziz et al. argue that most measurement approaches lack generality [24], as they are applicable only to specific application scenarios or infrastructures [34][35]. Our work can be used for a broad range of applications of different domains, implementation techniques, etc.

B. Feature-Interaction Detection

There is a large body of research on automated detection of feature interactions (e.g. see Nhlabatsi et al. [6] and Calder et al. [36] for surveys). Many approaches aim at detecting feature interactions at the specification level. For example, Calder and Miller use a pair-wise measurement approach based on linear temporal logic to detect feature interactions [7]. They specify the behavior of an SPL in Promela (a modeling language). Using a model checker, they generate for each pair-wise combination a model checking run to verify whether the defined properties are still valid. Other approaches use state charts to model and detect feature interactions [37]. For example, in [38] feature specifications are translated to a reachability graph. The authors use state transitions to detect whether a certain state is not exclusively reachable in isolation (i.e. a feature interaction occurs).

There are approaches that provide means to detect semantic feature interactions, i.e., feature interactions that change the functional behavior of a program. Some use model checking techniques to find semantic feature interactions [39][40]. Apel's work uses verification techniques to verify whether semantic constraints still hold in a particular feature combination [41][42]. Other approaches aim at investigating the code base to detect structural feature interactions. For example, Liu et al. [9][43] propose to model feature interactions explicitly using algebraic theory. Liebig et al. [15] analyzed the variability 40 SPLs including the nesting of feature code (i.e. the interaction of features at the source-code level). In contrast to these approaches, we focus on PFIsin a black-box fashion.

In [3][8], we proposed a way to predict a configuration's footprint based on measurements and a manual specification of footprint feature interactions (e.g., manually adding interactions found with available domain knowledge or using source code analyses). Further, we provided a way to measure all pair-wise feature interactions. In this paper, we (1) do not rely on domain knowledge, (2) reducing measurement effort for pair-wise measurement, (3) use performance instead of footprint, (4) incorporate higher-order feature interactions, and (5) evaluate our approach with more industrial product lines.

VII. Conclusion

We presented a method that allows stakeholders to accurately predict the performance of customized and generated programs. It detects interactions among configuration options or features and evaluates their influence on performance. We detect feature interactions in a step-wise manner. First, we find features that interact. Second, we detect the combinations of these features that cause a measurable interaction and quantify their impact on the performance of a configuration. The common weak spot of such an approach is the exhaustive number of measurements required to detect interactions. We solved this problem by means of three heuristics that reduce the number of measurements without sacrificing precision in predictions.

Our evaluations demonstrate that an accuracy of 95 % is possible, on average, when using our heuristics. We demonstrated generality by using applications of varying domains, implemented with different programming languages and techniques, and configured via configuration files or compilation options.

REFERENCES

[1] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *ICSE*. ACM, 2011, pp. 131–140.

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

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

[4] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk, "Approaching non-functional properties of software product lines: Learning from products," in *APSEC*. IEEE, 2010, pp. 147–155.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," *Comput. Netw.*, vol. 41, no. 1, pp. 115–141, 2003.

[6] A. Nhlabatsi, R. Laney, and B. Nuseibeh, "Feature interaction: The security threat from within software systems," *Progress in Informatics*, no. 5, pp. 75–89, 2008.

[7] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties: A case study," *Form. Methods Syst. Des.*, vol. 28, no. 3, pp. 213–261, 2006.

[8] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. to appear, 2011.

[9] D. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in *GPCE*. ACM, 2011, to appear.

[10] C. H. P. Kim, C. Kästner, and D. Batory, "On the modularity of feature interactions," in *GPCE*. ACM, 2008, pp. 23–34.

[11] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[12] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE TSE*, vol. 28, no. 1, pp. 109 –111, 2002.

[13] A. W. Williams, "Determination of test configurations for pair-wise interaction coverage," in *TestComm*. Kluwer, B.V., 2000, pp. 59–74.

[14] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *SPLC*, 2010, pp. 196–210.

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

[16] S. Apel and D. Beyer, "Feature cohesion in software product lines: An exploratory study," in *ICSE*. ACM, 2011, pp. 421–430.

[17] C. Taube-Schock, R. J. Walker, and I. H. Witten, "Can we avoid high coupling?" in *ECOOP*. Springer, 2011, pp. 204–228.

[18] A. L. Barabasi, R. Albert, and H. Jeong, "Mean field theory for scale-free random networks," *Physica A Statistical Mechanics and its Applications*, vol. 272, pp. 173–187, Oct. 1999.

[19] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*. ACM, 2011, pp. 461–470.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*. ACM, 2009, pp. 265–276.

[21] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *OOPSLA*. ACM, 2007, pp. 57–76.

[22] A. Metzger and P. Heymans, "Comparing feature diagram examples found in the research literature," Technical report TR-2007-01, Univ. of Duisburg-Essen, Tech. Rep., 2007.

[23] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *ICSE*. ACM, 2010, pp. 445–454.

[24] A. A. Abdelaziz, W. M. W. Kadir, and A. Osman, "Comparative analysis of software performance prediction approaches in context of component-based system," *IJCA*, vol. 23, no. 3, pp. 15–22, 2011.

[25] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE TSE*, vol. 30, no. 5, pp. 295–310, 2004.

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

[27] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. Springer, 2007.

[28] K. V. Mardia, J. T. Kent, and J. M. Bibby, *Multivariate Analysis (Probability and Mathematical Statistics)*, 1st ed. Academic Press, 1980.

[29] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, vol. 24, no. 6, pp. 417–441, 1933.

[30] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE*. IEEE, 2009, pp. 592–603.

[31] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE TSE*, vol. 36, no. 6, pp. 865–877, 2010.

[32] J. Happe, H. Koziolek, and R. Reussner, "Facilitating performance predictions using software components," *IEEE Software*, vol. 28, no. 3, pp. 27 –33, 2011.

[33] R. Tawhid and D. C. Petriu, "Automatic derivation of a product performance model from a software product line model," in *SPLC*, 2011, pp. 80–89.

[34] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *Journal of Syst. Softw.*, vol. 74, no. 1, pp. 35–43, 2005.

[35] S. Yacoub, "Performance analysis of component-based applications," in *SPLC*. Springer, 2002, vol. 2379, pp. 1–5.

[36] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," *Comp. Netw. and ISDN Systems*, vol. 41, pp. 115–141, 2003.

[37] C. Prehofer, "Plug-and-play composition of features and feature interactions with statechart diagrams," *Software and Systems Modeling*, vol. 3, no. 3, pp. 221–234, 2004.

[38] K. P. Pomakis and J. M. Atlee, "Reachability analysis of feature interactions: A progress report," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 216–223, 1996.

[39] 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 *ICSE*. ACM, 2010, pp. 335–344.

[40] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE*. IEEE, 2009, pp. 269–280.

[41] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting dependences and interactions in feature-oriented design," in *ISSRE*. IEEE, 2010, pp. 161–170.

[42] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *ASE*. IEEE, 2011, to appear.

[43] J. Liu, D. Batory, and S. Nedunuri, "Modeling interactions in feature-oriented designs," in *ICFI*. IOS Press, 2005, pp. 178–197.