GSDLAB TECHNICAL REPORT

Variability-Aware Performance Modeling: A Statistical Learning Approach

Jianmei Guo, Krzysztof Czarnecki, Sven Apel,
Norbert Siegmund, Andrzej Wąsowski

GSDLAB–TR 2012–08–18 August 2012

Generative Software
Development Lab

University of
Waterloo

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Variability-Aware Performance Modeling: A Statistical Learning Approach

Jianmei Guo*, Krzysztof Czarnecki*, Sven Apel†, Norbert Siegmund‡, and Andrzej Wąsowski§
*University of Waterloo, Canada
†University of Passau, Germany
‡University of Magdeburg, Germany
§IT University of Copenhagen, Denmark

***Abstract*—Customizable software systems allow users to derive configurations by selecting features. Building a performance model to understand the tradeoff between performance and feature selection is important to be able to derive a desired configuration. A challenge is to predict performance accurately when features interact. Another is that, in practice, we can often measure only few configurations as a sample for prediction, and we cannot select these configurations freely to cover certain feature interactions. We propose an incremental and variability-aware approach to performance modeling based on statistical learning. Our approach incorporates performance-relevant feature interactions and quantifies their influence implicitly during the process of performance modeling. It identifies the most relevant feature selections automatically for performance prediction. Empirical results on six real-world case studies show that our approach achieves an average of $94\,\%$ prediction accuracy measuring few randomly selected configurations.**

I. Introduction

Customizable software systems facilitate software configuration by *variability* management, i.e., by identifying and controlling commonalities and variations in a set of software artifacts such as requirements, architecture, components, and code [1]–[3]. Commonalities and variations are defined and managed by *variability models*, which are used in many real-world, customizable systems such as the Linux kernel and the embedded operating system eCos [4]. Today, the most widely used variability models are *feature models*, in which *features* are essentially configuration options relevant to users [5]–[7]. Every configuration derived from a customizable system is represented by a combination of features. A feature model defines all valid feature combinations (a.k.a., *feature selections*) of a customizable system. As shown in Figure 1, a feature model can be displayed as a tree-like structure that defines relationships among features. These relationships define the choices that users can make when configuring a system.

For a customizable software system, each derived configuration and its corresponding feature selection need to satisfy various user requirements [8]–[11]. One of the most important user requirements is *performance* (e.g., response time, throughput, and resource utilization), because it directly affects user perception and costs [12]. *Variability-aware performance modeling* identifies and incorporates the correlations between performance and feature selection and helps users make tradeoffs when configuring a system to satisfy a certain performance goal [13]. For example, the database illustrated in Figure 1 can be configured to achieve maximum performance when used on a server, but may also be configured for low energy consumption when deployed on a smartphone.

Users need performance models to understand the influence of a feature selection on performance and to answer questions like "What happens to performance if I use this configuration?" [13]. Users can also use performance models to anticipate and eliminate performance anomalies [14]. Moreover, software configuration in practice is often "reconfiguration" [15], [16], i.e., users start with a default configuration and then modify it; in this case, performance models help users choose a proper starting point close to the final performance goal.

It is non-trivial to build performance models for customizable software systems. First, even for a small feature model, feature combinatorics can produce an exponential number of configurations [10]. For example, the relatively simple feature model shown in Figure 1 gives rise to $2,560$ valid configurations. So, it is usually infeasible to carry out a comprehensive performance measurement for all valid configurations of a customizable system [9]. Hence, in practice, often only a limited set of configurations is measured, either by simulation in the lab or by monitoring in the field (i.e., in practical deployments) [13]. How to use *few* measured configurations to build performance models with reasonable prediction accuracy (e.g., above $90\,\%$) is still an open issue.

Second, *feature interactions* may cause unexpected behavior and unpredictable performance anomalies, which makes many straightforward assessment methods not applicable [17], [18]. For example, the performance influence of two features, both appearing in a configuration, may not be easily deducible from the performance influence of each feature in separation. It is still a major challenge to detect and incorporate feature interactions and their performance influence effectively [19].

Third, existing approaches detect feature interactions mainly by heuristics, based on different feature coverage criteria [8], [9], [20]. For example, the *pair-wise* heuristic assumes that features interact mainly in pairs; it selects and measures a specific set of configurations that cover all pair-wise feature interactions for performance prediction [8]. However, in practice, the configurations that we can measure or that we already have (e.g., by monitoring in the field) are often *random*;

and they may not meet any feature coverage criterion. Is it possible to use random configurations as a basis for reasonable performance modeling?

We aim at building variability-aware performance models to address the above challenges. We formalize the performance modeling problem and reduce it to a *nonlinear regression* problem, and we use a statistical learning technique to solve the problem and to model the correlations between performance and feature selection. Our approach incorporates feature interactions and quantifies their performance influence implicitly along with the learning process. It identifies the feature selections most relevant to performance automatically and uses them for performance prediction. Our approach works in an incremental way so that users can use it to produce prediction results starting with a small set of measured configurations, and extend it when further configurations are available. The main contributions are summarized as follows:

- We propose an incremental and variability-aware approach to performance modeling with reasonable prediction accuracy based on few random sample configurations.
- Our approach aims at automatically identifying the most relevant feature selections for performance prediction, which implicitly incorporate performance-relevant feature interactions; it also applies to higher-order feature interactions without additional effort.
- We demonstrate the practicality and generality of our approach by experiments on a public dataset with six customizable systems spanning different domains, implementation languages, and configuration mechanisms.
- Empirical results show that our approach produces an average of $94\,\%$ prediction accuracy on the evaluated dataset. Moreover, our approach shows a desirable and robust increasing trend of accuracy with the increase of sample configurations.

II. Problem Formalization and Challenges

Figure 1 shows the feature model of Berkeley DB (C version)[1] using the notation defined in [5], [21]. A feature model FM defines all N features $F = \{f_i\}$ with $1 \leq i \leq N$ and all valid configurations C of a customizable system. A configuration $c \in C$ is an assignment of selection or deselection to each feature in the feature model. A configuration is *valid* if the assignment is allowed by the feature model.

We define a Boolean-valued function over F denoting whether a feature is selected, i.e., $x : F \rightarrow \{Y, N\}$. That is, if $x(f) = Y$, then feature f is selected, otherwise it is deselected. We represent a configuration as a set of assignments to all features, i.e., $c = \{x(f_1), x(f_2), ..., x(f_N)\}$. For simplicity, $c = \{x_1, x_2, ..., x_N\}$, if we mark a position number on each feature in the feature model. For example, in Figure 1, a configuration with the selected features {DB, CRYPTO, PAGESIZE, PS1K, CACHESIZE, CS16M} can be encoded as $\{x_1 = Y, x_2 = Y, x_9 = Y, x_{10} = Y, x_{15} = Y, x_{17} = Y\}$.[2]

[1] http://www.oracle.com/us/products/database/berkeley-db/

[2] Here, we omitted the deselected features for brevity.

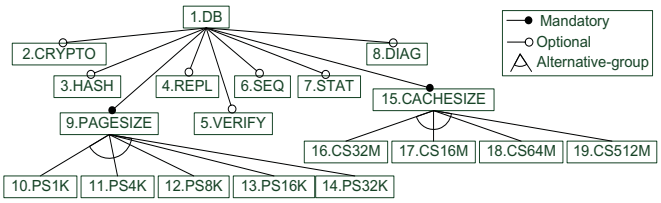


Figure 1. Feature model of Berkeley DB (C version)

For a certain performance metric P, we define a real-valued function on C to indicate the *actual* performance influence of each configuration, i.e., $p : C \rightarrow \mathbb{R}$. The problem is that we cannot measure $p(c)$ for every $c \in C$ due to the exponential number of configurations. In practice, we have only a limited set of already produced and measured configurations $\hat{C} = \{\hat{c}_1, \hat{c}_2, ..., \hat{c}_m\} \subseteq C$ (usually $|\hat{C}| \ll |C|$) and their corresponding performance measurements $P(\hat{C}) = \{p(\hat{c}_1), p(\hat{c}_2), ..., p(\hat{c}_m)\}$. Given $\hat{C}$ and $P(\hat{C})$ as input, the goal is to find a hypothesis function h to predict the performance influence of each configuration in C as accurately as possible:

$$h : C \rightarrow \mathbb{R} \; s.t. \; L(h(c), p(c)) \; is \; minimal \qquad (1)$$

where $L(h(c), p(c))$ is a *loss function* for penalizing errors in prediction. Assuming C as input, $p(c)$ as the target output, $\hat{C}$ and $P(\hat{C})$ as training examples, and $h(c)$ as the target hypothesis, the above problem can be formulated as a *learning* problem [22].

To solve such a learning problem, the challenges are manifold. First, given a certain performance metric (e.g., response time), each feature in the feature model has its own *relevance* to the metric. How to identify the most relevant features and filter out the irrelevant ones? Second, we cannot use simple linear models such as $h(c) = \sum_{i=0}^{n} \beta_i x_i$ as performance models, because the change of performance influence could be nonlinear due to feature interactions. How to build such a nonlinear model? Furthermore, the effects of many learning techniques depend heavily on the quantity and quality of input training examples [22]. In our case, we have only few random configurations ($|\hat{C}| \ll |C|$). How to achieve prediction with reasonable accuracy using limited training examples?

III. Overview of Approach

To address the challenges above, we propose an incremental and variability-aware approach to performance modeling based on statistical learning. Figure 2 illustrates the approach. From the user viewpoint, the approach answers the following question: What is the value of the performance metric P (e.g., response time or throughout) for the application A using configuration c? Our approach matches c with the most similar performance decision rule of A and returns a quantitative prediction of P (i.e., $h(c)$). The rules are obtained automatically by statistical learning from existing training examples. Further, users can verify the prediction by simulation in the lab or monitoring in the field. The verified results can be reused as training examples. Thus, our approach works in an incremental way and produces results based on existing data.

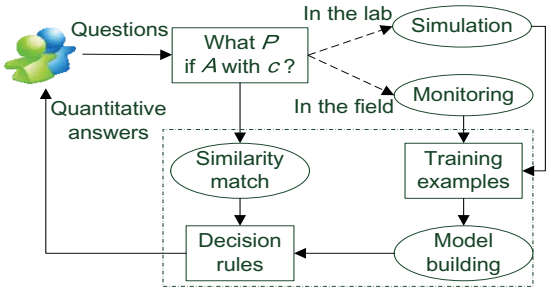


Figure 2. Overview of our approach

TABLE I
EXAMPLE DATASET OF BERKELEY DB

x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	x17	x18	x19	p(c)
Y	Y	N	N	Y	Y	Y	N	Y	N	N	Y	N	N	Y	N	N	Y	N	10.7352
Y	N	Y	N	Y	N	N	Y	Y	N	N	Y	N	N	Y	N	N	N	Y	0.7715
Y	N	N	Y	N	Y	Y	N	Y	N	N	Y	N	N	Y	N	N	Y	N	0.5008
Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	Y	Y	N	N	N	Y	2.1563
Y	N	N	N	Y	Y	N	Y	Y	N	N	N	N	Y	Y	N	Y	N	N	0.9484
Y	N	N	Y	N	N	Y	Y	Y	N	N	N	N	N	Y	N	Y	N	N	0.5802
Y	Y	Y	Y	Y	N	N	N	Y	N	N	N	N	Y	Y	N	N	Y	N	38.9490
...																			

IV. VARIABILITY-AWARE PERFORMANCE MODELING

This section presents our approach in detail. We describe each step of the process underlying our approach, providing the main intuitions behind the approach and the research question that need to be studied experimentally.

A. Data Interpretation and Preparation

According to the data nomenclature defined in [23], all training examples form a *dataset*. Each training example is an *observation*, namely a measured configuration $\hat{c} \in \hat{C} \subseteq C$ and its actual performance measurement $p(\hat{c})$. Each observation is recorded in terms of *variables*, which can be distinguished as *input variables* and *output variables*. Input variables are measured or preset data items; in our case, they are the variables to indicate the selection or deselection of features defined in each configuration, i.e., $\{x_1, x_2, ..., x_N\}$. For example, Table I shows an example dataset of Berkeley DB, in which input variables, x_1 through x_{19}, denote the features shown in Figure 1. Output variables are the variables influenced by the input variables. Here, they are the variables with respect to the performance metric P.

Generally, a feature in a feature model can be either *abstract* or *concrete* [24]. A feature is abstract if and only if it is not mapped to any implementation artifacts; a concrete feature is mapped to at least one implementation artifact. Thus, we have the following intuition:

Intuition 1: The actual performance influence of a configuration results from all of its concrete features.

Furthermore, some features will appear in all valid configurations, e.g., the features "DB", "PAGESIZE" and "CACHESIZE" shown in Figure 1. These features provide no information to help distinguish the performance influence of different configurations (a.k.a., *information gain* in information theory and statistics [25]). This observation yields another intuition:

Intuition 2: Features that appear in all valid configurations of a feature model provide no information for performance modeling.

According to the above two intuitions, we process the input dataset by removing the abstract features and by removing the features appearing in all valid configurations, which reduces the number of input variables.

B. A Nonlinear Regression Problem

Variables can store different types of data. In our case, input variables are *categoric* and output variables are *numeric*. A categoric variable is one that takes on a single value from a fixed set of possible values, while a numeric variable has values that are integers or real numbers [23]. In our case, every input variable x_i has two values "Y" and "N" to indicate if a feature is selected in a configuration or is not. The output variable for a configuration c has two kinds of numeric values: the actual performance measurement $p(c)$ and the predicted performance influence $h(c)$. The input dataset is the set of measured configurations $\hat{C}$ and the corresponding performance measurements $P(\hat{C})$. Due to the numeric output and possible feature interactions, the problem defined in equation (1) reduces to a *nonlinear regression* problem [26].

An effective approach to nonlinear regression is to subdivide, or *partition*, the dataset into smaller regions where feature interactions are more manageable, for example, higher-order feature interactions are decomposed into pair-wise feature interactions. We continue to partition the sub-divisions recursively, as in hierarchical clustering [26], until finally we get to chunks of the dataset that are so manageable that we can fit simple models to them.

C. Classification and Regression Trees (CART)

Classification and regression trees (CART) is a simple and efficient technique to implement the above recursive partitioning process [27]. CART uses a tree to represent the recursive partition. As shown in Figure 3, CART starts with the input dataset ($\hat{C}$ and $P(\hat{C})$) and then partitions it into two mutually exclusive parts by a *split node*. Here, the first split node is the input variable x_2, which indicates whether the feature "CRYPTO" is selected or not. Observations with $x_2 =$ "N" go to the left, and observations with $x_2 =$ "Y" go to the right. The observations in each part are partitioned again by further split nodes (e.g., the variable x_{14} of feature "PS32K"). The partition process continues recursively until each terminal node or *leaf* of the tree represents a *cell* of the partition and has attached to it a simple local model, which applies to that cell only. To determine which cell a configuration is in, we start at the first split node and then match the value of each split with the configuration until a leaf. Suppose that we use $|l|$ to indicate the number of all samples contained in any cell l, we define the *local model* in the cell using a simple *piecewise-constant* model [28], namely the sample mean of the output variable in that cell:

$$h_l = \frac{1}{|l|} \sum_{\hat{c}_i \in l} p(\hat{c}_i). \quad (2)$$

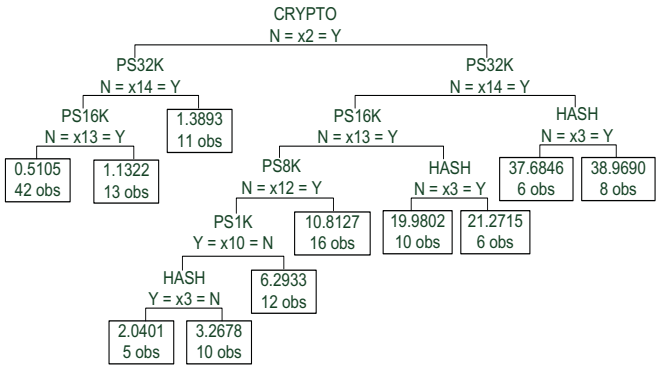


Figure 3. Example CART for 139 training examples of Berkeley DB

For example, in Figure 3, there are 42 observations in the leftmost cell and the sample mean is $h_l = \frac{1}{42}\sum_{i=1}^{42} p(\hat{c}_i) = 0.5105$. If a new configuration c satisfies $x_2 = N \wedge x_{14} = N \wedge x_{13} = N$, then it also falls into the leftmost cell and its predicted performance influence is $h(c) = h_l = 0.5105$.

Generally, all split nodes beyond the first split (i.e., root) imply *interaction effects* [28], i.e., the interacting influence of all input variables appearing in split nodes. For example, the influence of the variable x_{14} (feature "PS32K") depends on the value of variable x_2 (feature "CRYPTO"). Suppose that there are q leaves in a CART, we can formulate the *global model* of CART within the function framework of an *additive model* [26] as follows:

$$h(c) = \sum_{i=1}^{q} \beta_i \delta_i(x_1, x_2, ..., x_N) \tag{3}$$

where δ_i is a function to represent the combination of input variables in the branch from root to the i^{th} leaf, and β_i is the corresponding weight to indicate the interacting influence of all variables in the branch. Defining an indicator function $I : \{\delta_i\} \rightarrow \{0, 1\}$ to denote if formula δ_i is satisfied, we can represent all the branches of CART with indicator variables. For example, a part of the global model for CART shown in Figure 3 is as follows:

$h(c) = 0.5105 * I(x_2 = N \wedge x_{14} = N \wedge x_{13} = N) + 1.1322 * I(x_2 = N \wedge x_{14} = N \wedge x_{13} = Y) + \cdots$.

This equation can be easily translated into a set of *decision rules*, as shown in Figure 2. For example, the above equation can be paraphrased using if-then rules as follows:
If $x_2 = N \wedge x_{14} = N \wedge x_{13} = N$, then $h(c) = 0.5105$;
If $x_2 = N \wedge x_{14} = N \wedge x_{13} = Y$, then $h(c) = 1.1322$; $\cdots$.

From Figure 3 and the above equations and decision rules, we can easily see the effects of whenever two or more input variables interact. These combinations of interacting variables and their identified effects represent the feature selections most relevant to performance and their influence. Essentially, they also incorporate performance-relevant feature interactions and their influence implicitly. For example, in Figure 3, the branch from root to the rightmost leaf identifies a performance-relevant feature combination $(x_2 = Y \wedge x_{14} = Y \wedge x_3 = Y)$ and its influence 38.9690, which may incorporate the influence of pair-wise feature interactions such as $f_2\#f_{14}$ and $f_2\#f_3$ or higher-order feature interactions such as $f_2\#f_{14}\#f_3$.[3]

D. CART Design for Variability-Aware Performance Modeling

1) Choosing Split Nodes: The first problem that CART needs to solve is determining how to split the input dataset using information contained in the dataset. Recall the loss function used in equation (1). The most common loss function for a regression problem is *squared error loss* [26]: $L(h(c), p(c)) = (h(c) - p(c))^2$. The within-node sum of squared errors for a given CART T is:

$$S = \sum_{l \in leaves(T)} \sum_{\hat{c}_i \in l} (h_l - p(\hat{c}_i))^2 \tag{4}$$

where h_l is the local model defined in equation (2) for leaf l. Thus, we rely on the following intuition to build a CART:

Intuition 3: To achieve minimal prediction error, we choose the split node that *minimizes* S as the best one for each recursive partition of CART.

2) Scale of Training Set: Many learning techniques depend heavily on the quantity and quality of training examples [22]. Regardless of impact of noises, more training examples imply higher accuracy. However, in practice, it is usually difficult to obtain performance measurements for many configurations. For example, SQLite[4] has $3,932,160$ configurations (see Table II); measuring even just $1\,\%$ of them would be extremely time consuming. Since the number of all features in a feature model (i.e., N) is often far less than the number of all configurations (e.g., SQLite has only 39 features), a practical consideration is to use the linear number of features as the size of the input training set, e.g., N, $2 * N$, or $3 * N$. This leads to the following research question:

RQ 1: Can we achieve a reasonable prediction accuracy based on a small training set whose size is proportional to the number of features?

3) Pruning: An important problem for building a good CART is how to prune the tree to avoid *overfitting* training examples. A hypothesis function h is said to overfit the training examples if there exists another hypothesis h', such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the whole population of all possible inputs [22]. Overfitting is a significant practical difficulty for many learning techniques [22]. For CART, many pruning techniques have been proposed to avoid overfitting [28] and existing tools also provide parameters to control the pruning process [23]. However, systematical tuning of these parameters often leads to a manual, iterative process [13].

To implement a simple automated process of building CART, we use the size of the input training set (i.e., $m = |\hat{C}|$) to automatically set the values of two parameters $minsplit$ and $minbucket$. The $minsplit$ parameter specifies the minimum number of observations that must exist at a node in the tree before it is considered for splitting. The $minbucket$

[3]Here, $\#$ is the notation of feature interactions defined in [18].
[4]http://www.sqlite.org/

parameter is the minimum number of observations in any leaf node. Considering the scales of systems we studied, the rules of automated parameter setting conform to the following intuition:[5]

Intuition 4: If $0.1 * m < 10$, then $minbucket = \llcorner 0.1 * m + 0.5 \lrcorner \wedge minsplit = 2 * minbucket$; otherwise $minsplit = \llcorner 0.1 * m + 0.5 \lrcorner \wedge minbucket = \llcorner minsplit/2 \lrcorner$. Finally, $minsplit \geq 4 \wedge minbucket \geq 2$.

For example, as shown in Figure 3, we build a CART with $minsplit = 14$ and $minbucket = 7$ for Berkeley DB's 139 training examples. Furthermore, we use only the two parameters and fix others to control the pruning process and to generate CART automatically, which leads to another research question:

RQ 2: Can we use only the two parameters $minsplit$ and $minbucket$ set automatically by Intuition 4 to generate CART automatically and avoid overfitting?

4) Feature Coverage: Two common feature coverage criteria are *feature-wise*, which covers configurations with and without every individual feature, and *pair-wise*, which covers configurations with and without every pair-wise feature interaction [8], [9], [41]. The feature-wise heuristic determines a feature's performance influence by calculating the performance delta of two configurations with and without the feature. The pair-wise heuristic selects and measures a specific set of configurations that cover all pair-wise feature interactions relevant to performance. Some heuristics are also designed to cover various higher-order feature interactions [8], but they are based on feature-wise and pair-wise heuristics and need more measurements. The feature-wise and pair-wise heuristics depend on a set of specifically selected configurations, whereas our approach uses random configurations that do not need to conform to any feature coverage criterion. Thus, we have the following research question:

RQ 3: What is the difference of effects between our approach using random configurations and the feature-wise or pair-wise heuristics for comparable sizes of training sets?

5) Missing Data: Missing data are a problem for all statistical analyses [28]. In our case, the training examples are randomly selected and the size of training set is supposed to be comparatively small, so the training set is *likely* to have missing values for some features. Although CART is claimed to be robust to missing data [28], we are still interested in the following research question:

RQ 4: Is our approach robust when using few random training examples?

A special situation is that the training set has *completely* missing values for some features, i.e., *missing features* [28]. For example, users could prefer configurations with certain features; thus, the produced and measured configurations do not cover all features defined in the feature model and miss some features completely [20]. In our case, for all configurations in the training set, if an input variable $x_i \equiv$ "N", then the training set misses feature f_i; if $x_i \equiv$ "Y", feature f_i is excluded during performance modeling according to Intuition 2, which is essentially also a situation of missing features. The feature-wise or pair-wise heuristics cannot be used in this case. This leads to an interesting research question:

RQ 5: Does our approach produce reasonable predictions using a training set with some missing features?

E. Similarity Match

As shown in Figure 2, predicting performance for a new configuration c, our approach will match c with the most similar decision rule of A and returns the quantitative prediction of P (i.e., $h(c)$). According to the CART shown in Figure 3 or its derived decision rules, the similarity match is straightforward, i.e., matching the configuration with a complete branch from root to a leaf in the CART or the complete premise of a decision rule. For example, consider a new configuration with selected features {DB, CRYPTO, REPL, SEQ, PAGESIZE, PS32K, CACHESIZE, CS64M}, it matches the branch $(x_2 = Y, x_{14} = Y, x_3 = N)$ in the CART shown in Figure 3, and its performance influence predicted by the branch is 37.6846. Here, our approach finds out only the feature selections that are most relevant to the performance metric and produce minimal performance variance in the input dataset, which leads to the following research problem:

RQ 6: Can the performance influence of a configuration be predicted only by the features selections identified by CART?

V. Evaluation

To answer the above research questions, we implemented our approach using R 2.15.1 and Java (Eclipse 4.2 with JVM 1.7). R is a language and environment for statistical computing and graphics.[6] We used *Rattle* and *rpart* packages in R to generate CART [23]. We also developed a rule generator to automatically parse the generated CART and extract decision rules. Furthermore, we performed experiments to evaluate our approach. Our empirical results are presented in this section.[7]

A. Experimental Setup

We use a publicly-available dataset [8] for two reasons. First, the dataset covers a reasonable spectrum of practical application scenarios. As shown in Table II, there are six existing real-world customizable systems with different characteristics: different sizes (45 thousand to 300 thousand lines of code, 192 to millions of configurations), different implementation languages (C, C++, and Java), and different configuration mechanisms (such as conditional compilation, configuration files, and command-line options). Second, the dataset includes the performance measurements of all configurations of all systems (The exception is SQLite where the original authors measured 4,553 configurations as the sample and additional 100 random configurations for accuracy evaluation [8]). For each system, the performance has been measured using its

[5]Here, $\llcorner \lrcorner$ indicates rounding down, i.e., $\llcorner x \lrcorner = max\{n \in \mathbb{Z} | n \leq x\}$.

[6]http://www.r-project.org/

[7]An implementation of our approach and all experimental results including intermediate data are available at http://code.google.com/p/cpm/.

TABLE II
OVERVIEW OF THE ANALYZED SYSTEMS

No.	System	Domain	Lang.	LOC	|C|	N	PW
1	Apache	Web Server	C	230,277	192	9	29
2	LLVM	Compiler	C++	47,549	1,024	11	62
3	x264	Video Enc.	C	45,743	1,152	16	81
4	Berkeley DB	Database	C	219,811	2,560	18	139
5	Berkeley DB	Database	Java	42,596	400	26	48
6	SQLite	Database	C	312,625	3,932,160	39	566

Lang. - Language; **LOC** - Lines of Code; **|C|** - number of all configurations; **N** - number of all features; **PW**- number of measurements by pair-wise heuristic.

standard benchmark either delivered by its vendor (e.g., Oracle's standard benchmark for Berkeley DB) or often used in its application community (e.g., autobench and httperf for Apache Web Server).

In our experiments, *independent variables* are the kind of the analyzed system and the size of the training set. The prediction fault rate ($\frac{|actual-predicted|}{actual}$) and the model building time for each configuration of a system are measured as *dependent variables*. To reduce the fluctuations in dependent variables caused by random generation, we performed five repetitions for each combination of independent variables. All measurements were performed on the same Windows 7 machine with Intel Core i5 CPU 2.5 GHz and 8 GB RAM.

1) Generation of Training and Test Sets: We randomly selected a certain number of configurations from the dataset as the training set and all of the rest as the test set. Note that for all but one system the dataset contains the entire population of configurations. For each system, we generated four sizes of training sets: N, $2*N$, $3*N$, and PW. N is the number of features of each system, and PW is the number of measurements needed by pair-wise heuristic mentioned in Section IV-D4. The two numbers for each system are listed in Table II. For each size of the training set, the generation process is repeated five times.

2) Simulation of Missing Features: We simulated the situation of missing features, as discussed in Section IV-D5. For each system, we first randomly selected a certain number of features and fixed their indicator variables to a certain value ("Y" or "N"). Here, we set two percentages $20\,\%$ and $40\,\%$ of all relevant features (excluding those identified by Intuitions 1 and 2). Then, we randomly selected a set of configurations satisfying the above fixing conditions from the dataset as the training set and all of the rest as the test set. For example, in Figure 1, if the indicator variable of feature "CRYPTO" is fixed to "N", i.e., $x_2 \equiv$ "N", then all configurations in the training set miss feature "CRYPTO"; if $x_2 \equiv$ "Y", then feature "CRYPTO" appears in all configurations in the training set but is excluded during performance modeling (see Intuition 2), which is also a situation of missing features.

Here, the size of training sets is the number of measurements needed by the pair-wise heuristic, i.e., PW in Table II. However, sometimes, we may not be able to collect enough configurations satisfying the fixing conditions. For example, all configurations satisfying a fixing condition are less than PW. If the collected configurations are less than $\frac{1}{2}*PW$, we discard them and produce no results. The above generation process is repeated five times. That is, for each percentage ($20\,\%$ or $40\,\%$) and each value ("Y" or "N"), we randomly selected five different combinations of features and fixed their values to generate training sets and test sets.

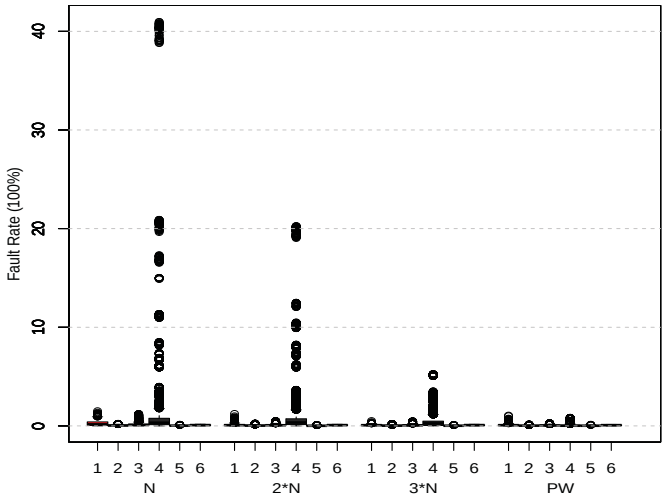
B. Results

We first measure the fault rate and the model building time of our approach for RQ 1, 2, 4 and 6. Then, we compare our approach with feature-wise and pair-wise heuristics for RQ 3. Finally, we measure the fault rate using training sets with some missing features for RQ 5. Each experiment includes hypothesis and evidence.

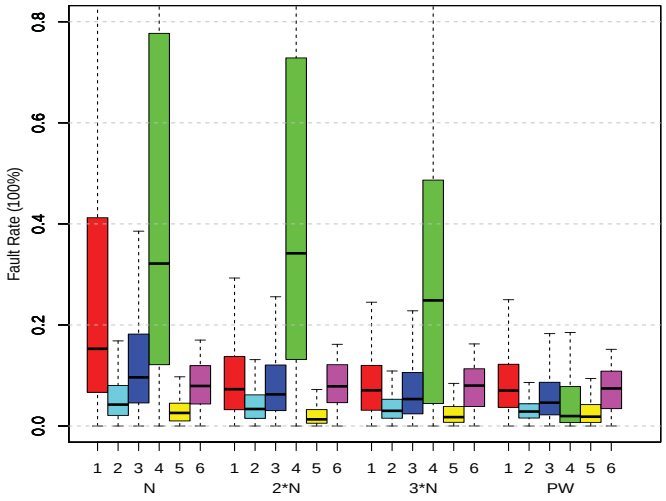
1) Fault Rate: The hypothesis of this experiment is that our approach can produce meaningful prediction trends and results when working with different kinds of systems and small training sets. Figure 4 shows the statistics of fault rates when applying our approach to the six systems listed in Table II and four sizes of input training sets (i.e., N, $2*N$, $3*N$, and PW). Figure 4(a) shows the boxplot of all statistics including outliers. We can see that the outliers decrease significantly as the size of the training set increases ($N \sim PW$). Figure 4(b) excludes the outliers so that the box, the median, and the whisker can be shown clearly. Table III lists the medians and means of fault rates. From Figure 4(b) and Table III, we can see an obvious decreasing trend of the fault rate with an increasing training set. When the size of the training set reaches PW, the median of fault rates reduces to less than $8\,\%$ and the mean less than $10\,\%$ for all systems. For half of the analyzed systems, our approach produces the median less than $8\,\%$ and the mean less than $9\,\%$ using only the training set with size N.

2) Model Building Time: A further hypothesis is that our approach can build performance models fast when working with different kinds of systems and small training sets. Figure 5 shows the measured time consumption of model building when applying our approach to the six systems and four sizes of training sets. For most of the systems (five out of the six), the time consumption is less than 0.04s. This is because these systems have less than 30 features in the feature model and less than 150 configurations as the training set, as listed in Table II. Only for SQLite with 38 features, the time consumption shows an increasing trend as the size of the training set scales from 39 to 566; however, it is still less than 0.09s. Since performing more measurements becomes infeasible in a reasonable amount of time, we conclude that the model building time is always practical.

3) Comparison on small training sets: We compared our approach with the feature-wise and pair-wise heuristics to observe their effects on small training sets. The hypothesis here is that our approach provides a more flexible and robust prediction. Figure 6 shows the comparison results when applying these approaches to the six systems and the training sets scaling from N to PW. We took the means of fault rates on the sizes of FW and PW, published in [8], and showed



(a) Including outliers



(b) Excluding outliers

Figure 4. Fault rate statistics for the six systems listed in Table II and four sizes of training sets ($N \sim PW$)

TABLE III
STATISTICS OF MEDIANS AND MEANS OF FAULT RATES (%)

No.	N		2*N		3*N		PW	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean
1	15.30	26.90	7.30	11.60	7.10	8.40	7.00	9.70
2	4.20	5.70	3.40	4.50	3.00	4.00	2.90	3.30
3	9.60	15.10	6.30	8.50	5.30	7.20	4.60	6.40
4	32.16	112.38	34.18	98.31	24.86	46.78	1.97	7.80
5	2.60	3.20	1.30	2.20	1.80	2.60	1.80	2.70
6	7.90	8.00	7.90	8.10	8.00	7.60	7.40	7.20
Average	11.96	28.55	10.06	22.20	8.34	12.76	4.28	6.18

them as the blue small circles in Figure 6. Here, FW is the number of measurements needed by the feature-wise heuristic; in our case, FW is equal to N for the six systems. Since our approach works in an incremental way, we took the means of fault rates on each size, listed in Table III, as sample points, and fitted them to a curve using a quadratic polynomial, i.e., the black asterisks and black solid lines shown in Figure 6. As listed in Table II, in most cases, PW is larger than $3 * N$.

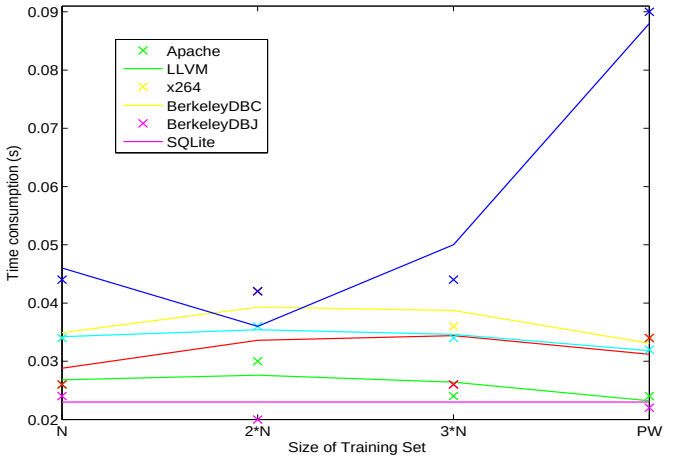


Figure 5. Time consumption of model building for the six systems and four sizes of training sets

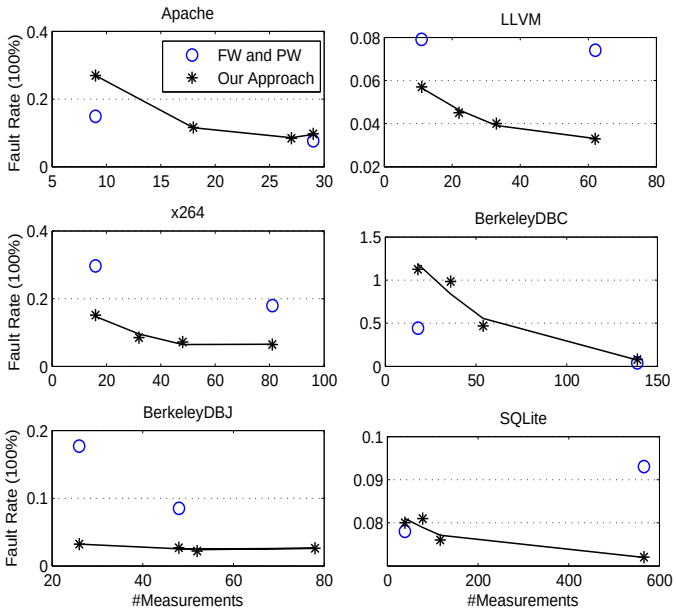


Figure 6. Comparison on small training sets

The exception is Berkeley DB Java where $PW = 48$ and $3 * N = 3 * 26 = 78$.

From Figure 6, we can see that our approach shows a robust decreasing trend of the fault rate on small training sets. For half of the analyzed systems (LLVM, x264, and Berkeley DB Java), the black solid line is located below the two blue circles; that is, our approach suffers smaller fault rates than the feature-wise and pair-wise heuristics. In two systems (Apache and Berkeley DB C), the fault rate of our approach is higher; however, it decreases dramatically as the size of training set scales from N to PW, and finally reaches a value very close to the fault rate of the pair-wise heuristic.

It is worth mentioning that x264 contains many higher-order feature interactions, so the feature-wise and pair-wise heuristics produce higher fault rates and additional special heuristics have to be considered to reduce the fault rate [8].

TABLE IV
FAULT RATES (%) FOR MISSING FEATURES

No.	20% Missing Features				40% Missing Features			
	x = N		x = Y		x = N		x = Y	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean
1	7.27	8.68	14.14	35.29	—	—	—	—
2	2.96	3.37	3.10	3.93	4.12	4.66	4.57	5.62
3	6.92	15.06	11.11	21.69	17.01	54.37	—	—
4	8.59	38.51	26.79	38.13	43.56	65.24	—	—
5	3.45	23.80	—	—	2.15	18.20	—	—
6	6.45	6.56	—	—	8.05	8.06	—	—
Average	5.94	16.00	13.79	24.76	14.98	30.11	4.57	5.62

Our approach works well in this case without special treatment of higher-order feature interactions.

4) Missing Features: Our hypothesis is that our approach does not work well when some important features (e.g., the features identified by CART) are missing completely. Table IV lists the medians and means of fault rates when applying our approach to the six systems and the training sets with some missing features generated by randomly fixing $20\,\%$ or $40\,\%$ of all features to "Y" or "N". We observe an increasing trend of the fault rate as the percentage of missing features increases ($20\,\% \sim 40\,\%$). In addition, we use the symbol "$-$" to indicate situations, in which we cannot obtain enough ($\frac{1}{2} * PW \sim PW$) configurations satisfying the fixing conditions. From Table IV, one can see that we often cannot obtain enough configurations when fixing features to "Y". It is because a feature model has many alternative groups (see Figure 1), in which only one feature can be selected.

C. Threats to Validity

To avoid misleading effects of specific-selected training sets and test sets, we generated them automatically by randomly selecting four sizes of configurations respectively from all valid configurations as the training set and all of the rest as the test set, and repeated each random selection five times with freshly generated training sets and test sets with the same size. The exception is the test set of SQLite, in which the original authors could not measure all possible configurations in reasonable time and thus only sampled 100 random configurations for prediction evaluation [8].

Existing CART implementation tools (e.g., Rattle and rpart in R) provide parameters to improve accuracy and reduce modeling time [23], [28], but systematical tuning of all parameters leads to a manual, iterative process [13]. To achieve a simple automated process of CART building, we used only two relevant parameters and fixed others to control the building process. We cannot guarantee that the accuracy and model building time depend on certain shapes of CART. However, to avoid influence of specific-shaped CART, we automatically set the two parameters' values according to the size of each input training set and then generated CART automatically.

To increase external validity, we used a public dataset with six systems spanning different domains with different sizes, different configuration mechanisms, and different implementation languages. All the systems are deployed and used in real-world scenarios. However, we are aware of that the results of our evaluations are not automatically transferable to all other customizable systems [8]. Moreover, the performance is measured by standard benchmarks in respective application domain. The generalities of measurement results depend on the generalities of the workloads.

D. Discussion

By experiments on six real-world customizable systems, we demonstrated the feasibility of our approach. With an average accuracy of $94\,\%$ (by statistics of means of fault rates in Table III), our approach produces reasonable predictions based only on a limited set of random configurations and their corresponding performance measurements.

Our approach has the following desirable properties. First, it achieves reasonable predictions with random configurations (RQ 4). Second, it can flexibly process any quantity of configurations, e.g., linear number of features (RQ 1). Third, it shows a robust decreasing trend of the fault rate with the increase of sample configurations; in most cases, it obtains similar accuracies to or even better than the feature-wise and pair-wise heuristics (RQ 3). The above three points support our approach as an incremental approach: it produces results based on existing data and improves accuracy continuously as the dataset increases. Moreover, our approach incorporates feature interactions and quantifies their performance influence implicitly during the process of model building. Our approach automatically identifies only the most relevant features selections for performance prediction (RQ 6), and performs well in the presence of higher-order feature interactions (e.g., for x264), without the need of additional measurements. Sometimes our approach works well using only N sample configurations (e.g., half of analyzed systems in our experiments), which makes it promising for systems with a large number of configurations.

With respect to RQ 5, if the data are really missing completely at random, the only loss is statistical power [28]. Our empirical results also verify this viewpoint. However, our experiment using the training sets with some missing features simulates a special case. In practice, users may prefer configurations with certain features and produce a set of configurations that are skewed to some features as the training examples, but the configurations they want to predict are likely skewed to the same features. If users have to predict new configurations with some features completely missing in existing training examples, we suggest measuring at least two (due to $minbucket \geq 2$) configurations covering these missing features before using our approach.

Our approach implements a simple automated process of building CART by controlling only two parameters and avoids overfitting to produce reasonable prediction accuracy (RQ 2). But we also noticed that the fault rate of our approach will decrease slowly when it has reached a "relatively" lower value. Take Apache, for example, as shown in Table III, the mean of fault rates reduces from $26.9\,\%$ to $11.6\,\%$, as the size of training set increases from N to $2 * N$, but it still maintains at $9.7\,\%$ when the size reaches PW. The problem is that the fault

rate of $9.7\,\%$ is higher than $3.9\,\%$ obtained by the pair-wise heuristic [8]. We believe that the problem can be alleviated by systematical parameter tuning for building a better CART, which will explore in future work.

VI. Related work

A. Performance Modeling

There are many model-based approaches to performance prediction [29], [30]. For example, linear and multiple regression model the correlations between input parameters and output performance metrics. Bayesian (or belief) networks are used to learn causal relationships between parameters. Machine learning approaches, such as principal component analysis, are used to learn the correlations between configurations and a performance metric and find dimensions of maximal variance in a dataset. However, the feasibility of these approaches depends heavily on the application scenario and program to be analyzed [8], because they usually require a large training set. Hence, it is not clear for an application scenario which method to choose. In contrast, we support any kind of application scenario independently of the program to be analyzed. We also empirically show that our approach can obtain good prediction effects using only a small training set.

Happe et al. [31] proposed a compositional reasoning approach based on the component specifications with resource demands and predicted execution time. Their approach is applicable only to component-based systems, whereas our approach is applicable to all customizable systems, once a variability model is built for them. Westermann et al. [12] presented an empirical study on automated inference of performance prediction models using statistical inference techniques, such as Kriging and MARS. They focused on building models with less measurement points and lower average fault rates, but they did not consider how to identify the configuration parameters relevant to performance.

Tawhid and Petriu [32] presented a model-driven approach to deriving a performance model from an extended feature model with performance analysis information. The approach requires up-front and detailed knowledge of domain-specific performance analysis, which makes tuning prediction for accuracy difficult. Our approach avoids the problems by directly working with performance measurements. Ramirez and Cheng [33] presented an approach that leverages goal-based models to facilitate the automatic derivation of utility functions at the requirements level, whereas our approach works at the configurations level. Thereska et al. [13] proposed a practical performance model for popular interactive client applications. They mainly consider peripheral configuration parameters such as CPU speed and memory size, while our approach focuses on the features of customizable systems.

B. Measurement-Based Prediction

Siegmund et al. [8] first proposed a general measurement-based approach that treats customizable systems as a black box and detects performance-relevant feature interactions. Their approach tries to find a sweet-spot between prediction accuracy and measurement effort based on heuristics. The approach needs a set of specifically selected configurations, whereas our approach works with few random configurations. Moreover, we do not have to detect all feature interaction explicitly, because our approach quantifies their influence implicitly in performance models. Nevertheless, we expect that a combination of both approaches is beneficial to further reducing measurements and increasing prediction accuracy.

Sincero et al. [34] used existing configurations and measurements to predict a configuration's non-functional properties. They designed the Feedback approach to find the correlation between feature selection and measurement and to provide a qualitative information about how a feature influences a non-functional property during configurations. In contrast to our approach, their approach do not actually predict a value quantitatively. Chen et al. [35] combined benchmarking and profiling to build a model to predict the performance of component-based applications. In contrast, our approach correlates performance measurements with configurations and can work with configurations collected by simulation or monitoring.

C. Root Cause Detection

Feature interactions are the main cause of unpredictable performance anomalies [17]. Many researchers have applied and extended various techniques to automatically detect feature interactions at the specification and semantic levels. For example, at the specification level, they used pair-wise measurement based on linear temporal logic [36] and state charts [37] to model and detect feature interactions. At the semantic level, namely feature interactions that change the functional behavior of a program, they used model checking [38], [39] and verification techniques [19], [40]. Some approaches aim at investigating the code base to detect structural feature interactions [18], [41]. Siegmund et al. [8] focused on all performance-relevant feature interactions in a black-box fashion. In contrast, our approach incorporates performance-relevant feature interactions and quantifies their influence implicitly during performance modeling.

Some researchers found that different execution paths [14] or different request flows [42] resulted from configuration changes are the root-causes of performance changes or anomalies. However, in most practice cases, we at most know the feature model of a customizable system and a limited set of existing configurations. It is difficult even for developers to know how the configurations will be executed. So we still consider customizable systems as a black box and model the correlations between performance and feature selection.

VII. Conclusion

We proposed an incremental and variability-aware approach to performance modeling of customizable software systems. We used a statistical learning technique named CART to correlate performance with feature selection. Our approach incorporates performance-relevant feature interactions along with the process of model building and quantifies their performance influence implicitly in performance models. Our

approach identifies only the most relevant feature selections automatically and uses them for performance prediction. We demonstrated the feasibility of our approach by experiments on six real-world systems spanning different domains, different implementation languages, and different configuration mechanisms. Our empirical results show that our approach produces an average of $94\,\%$ prediction accuracy using a limited set of random sample configurations. Our approach also shows a robust increasing trend of prediction accuracy as the number of sample configurations increases.

Our approach has the potential of wide application to help users understand tradeoffs between performance and feature selection when configuring a customizable software system. Next, we will consider systematical parameter tuning for CART to further improve prediction accuracy and modeling efficiency.

REFERENCES

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

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

[4] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *VaMoS*. ACM, 2012, pp. 173-182.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. "Feature-oriented domain analysis (FODA) feasibility study," Technical Report CMU/SEI-90-TR-021, SEI, CMU, 1990.

[6] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated analysis of feature models: challenges ahead," *Communications of the ACM*, vol. 49, no. 12, pp. 45-47, 2006.

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

[8] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *ICSE*. IEEE, 2012, pp. 167-177.

[9] 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.

[10] J. Guo, J. White, G. Wang, J. Li, and Y. Wang, "A genetic algorithm for optimized feature selection with resource constraints in software product lines," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208-2221, 2011.

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

[12] D. Westermann, R. Krebs, and J. Happe, "Efficient experiment selection in automated software performance evaluations," in *EPEW*. Springer, 2011, pp. 325-339.

[13] E. Thereska, B. Doebel, A. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," in *SIGMETRICS*. ACM, 2010, pp. 1-12.

[14] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *OSDI*. USENIX Association, 2010, pp. 237-250.

[15] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *ICSE*. IEEE, 2012, pp. 58-68.

[16] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: A perspective from the operating systems domain," in *ASE*. ACM, 2010, pp. 73-82.

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

[18] D. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in *GPCE*. ACM, 2011, pp. 13-22.

[19] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *ASE*. IEEE, 2011, pp. 372-375.

[20] B. Lamancha and M. Usaola, "Testing product generation in software product lines using pairwise for features coverage," in *ICTSS*. Springer, 2010, pp. 111-125.

[21] J. Guo, Y. Wang, P. Trinidad, and D. Benavides, "Consistency maintenance for evolving feature models," *Expert Systems with Applications*, vol. 39, no. 5, pp. 4987-4998, 2012.

[22] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[23] G. Williams, *Data Mining With Rattle and R: The Art of Excavating Data for Knowledge Discovery*. Springer, 2011.

[24] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund, "Abstract Features in Feature Modeling," in *SPLC*. IEEE, 2011, pp. 191-200.

[25] S. Kullback, *Information Theory and Statistics*. Courier Dover, 1997.

[26] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[27] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classication and Regression Trees*. Wadsworth, 1984.

[28] R.A. Berk, *Statistical Learning from a Regression Perspective*. Springer, 2008.

[29] A. Abdelaziz, W. Kadir, and A. Osman, "Comparative analysis of software performance prediction approaches in context of component-based system," *International Journal of Computer Applications*, vol. 23, no. 3, pp. 15-22, 2011.

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

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

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

[33] A. Ramirez and B. Cheng, "Automatic derivation of utility functions for monitoring software requirements," in *MoDELS*. Springer, 2011, pp. 501-516.

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

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

[36] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties: A case study," *Formal Methods in System Design*, vol. 28, no. 3, pp. 213-261, 2006.

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

[38] 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.

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

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

[41] 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.

[42] R. Sambasivan, A. Zheng, M. Rosa, and E. Krevat,"Diagnosing performance changes by comparing request flows," in *NSDI*. USENIX, 2011, pp. 1-14.