Bachelor's Thesis

FEATURE PERFORMANCE ANALYSIS: DIFFERENCES BETWEEN BLACK-BOX AND WHITE-BOX MODELS IN CONFIGURABLE SYSTEMS

MANUEL MESSERIG

March 26, 2023

Advisor:

Florian Sattler	Chair of Software Engineering
Christian Kaltenecker	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Jan Reineke	Real-Time and Embedded Systems Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________
(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Nearly all modern software systems are configurable. A significant reason developers introduce configurability is end-user flexibility with configurable options; the user can tailor the system to their needs, turning functionality on and off.

However, nowadays, configurable systems offer various configurable options, resulting in many unique configurations. All of these different configurations result in other behaviors of the system that we want to analyze; stakeholders are interested in which degree these behaviors influence various system metrics. One way to obtain these metrics is by examining the system's running time under each configuration.

In this thesis, we introduce two different analyses to identify the influence of each feature on a configurable system, a white-box analysis and a black-box analysis. We use the data generated by those approaches to build performance-influence models and use them to analyze and compare both methods.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

Modern software systems are designed to be configurable; we offer users flexibility by providing them to turn functionality on and off. We also expect a configurable software system to satisfy the demand of multiple users by offering a single software system that contains multiple features. [1].

An example of such a configurable software system would be the Linux kernel, whose code base itself contains over 6'000'000 lines of code containing more than 10'000 features [8]. All these optional features allow the user to create an operating system that meets his needs.

Now all these features affect the system in different ways. To keep track of all these features and their interactions, we use a feature model, which is essentially a tree, that depicts how features interact with another. Inside the feature model, there can be different kinds of constraints to visualize the relationships between features in a configurable system [1].

Now that we have an overview of the system we are interested in what capacity each feature or feature interaction influences the system, To identify these influences we present two different analyses: a white-box analysis and a black-box analysis. However, before this, we analyze the system's configuration space and select the crucial features of the system. Then, we create configurations from the selected features that contain the interactions we want to observe.

In the black-box analysis, we run the system with each configuration as input, and during execution, we can measure various metrics, such as memory and energy consumption. Whereas we will focus on measuring the execution time. We will use the measurement we collect to learn each feature's influence on the system using multiple linear regression.

In our white-box analysis, we have more information because we have access to the system's source code. We use an analysis that helps us determine which parts of the code are influenced by which feature. Then during execution, we can measure the time spent inside these features.

Now both our analyses generate different types of data that are different. Therefore, we need to transform this data into a model that we can then use to evaluate both analyses by comparing these models; for this, we use performance-influcence model. These models represent our configurable system as a polynomial, where each term represents either a feature or an interaction of features [11]. We build these models by using the data generated by the white-box analysis and black-box analysis.

To show the validity of our models, we establish a ground truth. To do this, we design a small configurable system to test both of our models. This system will contain several features, some of which interact with each other in different ways. Since we developed this system ourselves, we know how each feature should impact the runtime of our system, so we create a baseline performance-influence model to compare our models against.

1.1 GOALS OF THIS THESIS

In this thesis, we aim to determine if white-box and black-box analyses can identify the influence of the different features and feature interactions on a configurable system. Furthermore, we are interested in the differences of the performance-influcence models built with data generated by each analysis.

We are interested in whether both models can correctly identify the influence of each feature. If they reach the same conclusion, we can already see that it is feasible to use either of the two models to analyze a system, but from there, we still work out the advantages and trade-offs between the models so that the user can choose the one that meets his needs. Suppose they reach a different conclusion. In that case, we analyze the reason for the differences between them and examine whether one model performs particularly poorly in some instances and why this is so.

We collect all these interest points to form the following research questions:

RQ1 : How accurately does white-box and black-box models detect feature interactions?

RQ2 : Do performance models created by our white-box and black-box reach the same conclusion?

RQ3 : What are the reasons for similarities or differences between performance models?

2 BACKGROUND

2.1 CONFIGURABLE SYSTEMS

We call a system configurable if it offers options that allow developers and end-users to turn functionality on and off. This gives the system the benefit of satisfying the demand of multiple user groups by providing a single software system containing various features [16].

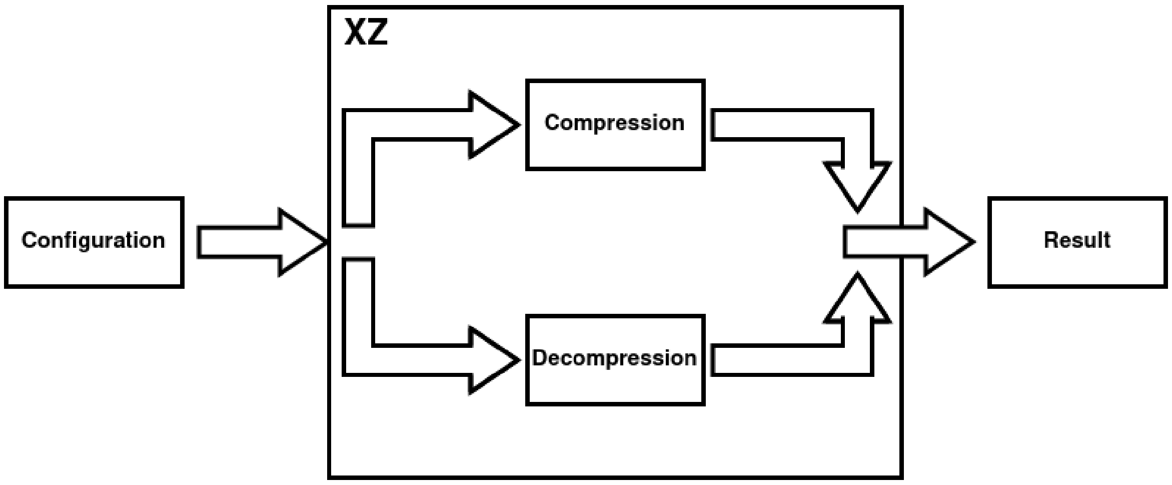

Figure 2.1: Simplified version of XZ.

As an example, let us inspect the compression tool XZ[1]. In Figure 2.1 depicts a simplified version of XZ, which contains two main functions, encryption, and decryption. It is up to the user to decide what he needs, but regardless of his choice, a single software system contains both functions.

2.2 FEATURES AND CONFIGURATIONS

During the years there have been many definitions to what a *feature* is, on one side, features are used as a means of communication between the different stakeholder of a system, where on the other hand, a feature is defined as an implementation-level concept. To unify both applications Apel et.al. introduced the following definition:

Definition 2.2.1. "A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option."[1]

Thus, a feature is both an abstract concept that refers to particular functionality of a system and the implementation of that functionality. In our example Figure 2.1 both, *Compression*

1 Visited at 21.03.2023 `https://tukaani.org/xz/`

and *Decompression*, are unique features, they refer to a piece of functionality of the system and the implementation.

We differentiate between *binary* features *numeric* features. A binary feature can be either selected or deselected; commonly, we denote a binary feature with 1 if we select it and 0 otherwise. We denote a numeric feature with a numerical value; these can have various meanings depending on the feature it implements. For Figure 2.1, *Decompression* could be a binary feature since we either have the option to decrypt a file or not; on the contrary, *Compression* could be implemented as a numeric feature, where the numeric value represents the quality of compression we want.

A configuration option is a predefined way for developers to change the functionality of the configurable system; these options allow us to select features we want to include or exclude. Thus, a configuration is a set of configuration options, whereas we call a configuration valid as long as it satisfies the system's constraints.

The configuration space refers to the set of all configurations. However, some of these configurations are invalid. In practice, we cannot execute systems with invalid configurations. Therefore, we shall implicitly exclude all invalid configurations whenever we refer to a configuration space.

As an example, XZ offers the user different features such as *Compression* and *Decompression*, whereas the configuration options *0-9* specify the degree of compression. An invalid configuration would be to select multiple configuration degrees of compression.

2.3 FUNCTIONAL AND NON-FUNCTIONAL PROPERTIES

When analyzing a configurable system, we distinguish between what a system can do and how the system archives that goal; the former refers to the functional properties, while the latter describes the non-functional properties of the system[12]. When we talk about functional properties, we mean everything a system can do, including which problem it solves and what features the system provides us, the user. For example, in Figure 2.1, we can see the features of XZ, *Compression*, and *Decompression*; they refer to the functionality XZ provides.

While functional properties refer to what a system can do, non-functional properties refer to how or in which circumstance that functionality is achieved[12]. To do so we can measure most of the nun-functional properties, such as performance, memory consumption, CPU usage and energy consumption. However, not all non-functional properties are quantifiable. Some relate to the quality of a system that we can not easily measure, such as how secure a system is or how high the code quality is in regard to code readability or documentation[12].

2.4 FEATURE MODEL

A configurable system often contains numerous features, all of which may interact with one each other or have different dependencies. However, as mentioned in Section 2.2 not all features are freely combinable, some of them can only be selected in the absence of others. The large the system the harder it gets to keep all these interactions in mind, therefore we use *Feature Models* to describe the relation between features and define which feature selections are valid [1].

Due to the fact that large configurable systems tends to have numerous feature we use feature diagrams which are a common visual representation of feature models.

2.4.1 *Feature Diagram*

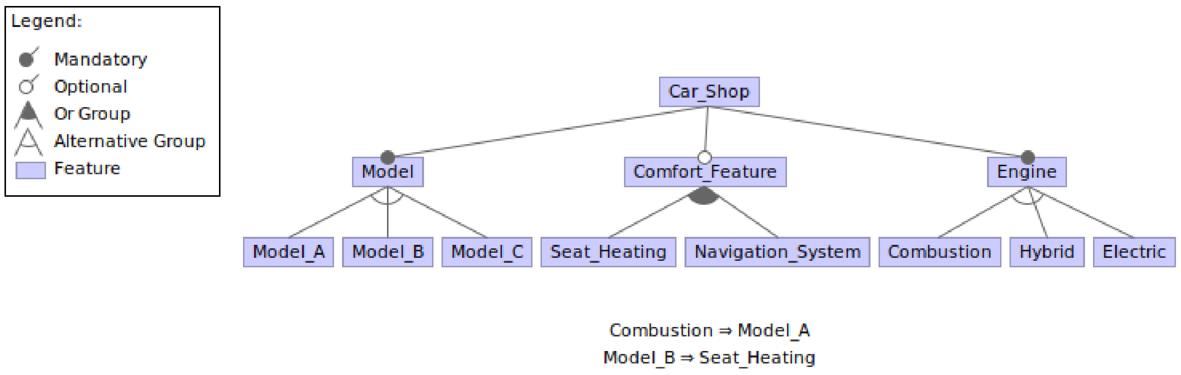

Figure 2.2: A feature diagram of a car dealership.

A feature diagram is conceptually built as a tree where each node of the three represents a feature, where an edge between a parent node and a child usually implies that the parent is a more general concept and the child a specialization. This leads to the fact that the further we descent in the hierarchy of the diagram, the more specialized the feature gets.

Figure 2.2 is an example for a feature diagram that depicts the structure of a car dealership. As we can see the root is the first node *car_shop* represents the general concept. Now if we look at the child nodes of the *Engine* feature, we can see concrete engine types, such as *Combustion*, *Hybrid*, and *Electric*.

We differentiate between two types of features, *abstract* and *concrete* features. Abstract features are used for structural purposes and do not correspond to implementation. In contrast, concrete features reflect variability inside the system. In Figure 2.2, the feature *Engine* would be considered an abstract feature since it does not implement anything, and its purpose is to group all types of engines the system contains. The specific engines, like *Combustion* would be concrete features since they implement a specific type of functionality.

Each feature contains a graphical notation indicating whether the feature is mandatory or optional. If the feature is mandatory, it is indicated with a black bubble; if it is optional, it is displayed with an empty bubble [2]. In Figure 2.2 we can see that *Engine* and *Model* are mandatory features, which make sense since both are necessary for any car, but *Comfort_Feature* an optional since they are not necessary for a car to function.

In addition to mandatory and optional features, there are alternative and choice groups. A parent can have one of these groups; we mark an alternative group with an empty half circle and an optional group with a filled circle. When we use a selection group, one feature needs to be selected, but others can also be selected; a choice group corresponds to the logical or operator. In an alternative group, we can select only one feature; the configuration is invalid if more than one feature is selected [2]. In Figure 2.2, we see that *Engine* has an alternative group, which makes sense since each car can only contain one *Engine*, whereas it makes sense that *Comfort_Feature* contains a choice group, you can have a navigation system and seat heating in a car without conflict.

A feature diagram may contain various constraints that need to be satisfied, defined as boolean algebra. In Figure 2.2, we see two constraints, $Combustion \implies Model\_A$ and $Model\_B \implies Seat\_Heating$. The reason for such constraints could be that *Model_B* is a luxury model that only gets shipped with seat heating.

Any feature model can also be translated into propositional formula. Therefore, a feature configuration is valid only iff it satisfies the propositional formula. We can therefore use a feature model to check whether a configuration is valid, which is particularly useful if we want to sample or enumerate all valid configurations [2].

2.5 PERFORMANCE-INFLUENCE MODELS

In the previous Section 2.4, we introduced a way to represent functional properties by using feature models. However, while doing so we ignored the non-functional properties which are as equally important inside a configurable system. Hence, to model the measurable non-functional properties and to which degree each feature influences the configurable system; we introduce performance-influence models.

To define performance-influcence model, we use the formal definitions from the paper "Performance-Influence Models for Highly Configurable Systems" by Sven Apel et al. [11].

A performance-influcence model is a polynomial consisting of several terms, each representing either a feature or an interaction between features, whereas the coefficient in each term represents the degree to which these features influence the system. The sum of all terms represents the time the performance-influcence model predicts given a configuration of features. Formally, let $\mathcal{O}$ be the set of all configuration options and $\mathcal{C}$ the set of all configurations, then let $c \in \mathcal{C}$ be a function $c : \mathcal{O} \implies \mathbb{R}$ that assigns either 0 or 1 to each binary option. If we select a feature o, then $c(o) = 1$ holds, otherwise $c(o) = 0$. In general, a performance-influence model is a function Π that maps configurations $\mathcal{C}$ to an estimated prediction, therefore $\Pi : \mathcal{C} \implies \mathbb{R}$.

We encode all our features as binary features and distinguish between single features o denoted as ϕ_o and feature interactions $i...j$ denoted as $\Phi_{i...j}$. Based on these definitions, we define a performance-influence model formally as:

$$\Pi = \beta_0 + \sum_{i \in \mathcal{O}} \phi_i(c(i)) + \sum_{i...j \in \mathcal{O}} \Phi_{i...j}(c(i)...c(j)) \tag{2.1}$$

While β_0 denotes the base performance, which refers to the time taken by the system regardless of configuration, $\sum_{i \in \mathcal{O}} \phi_i(c(i))$ is the sum of each feature and $\sum_{i...j \in \mathcal{O}} \Phi_{i...j}(c(i)...c(j))$ is the sum of each feature interaction.

Listing 2.1: Example code

```
1  void foo() {
2      bool A, B, C, D;
3      assign_feature(A, B, C, D); //Assigns true to each selected feature
4
5      fpcsc::sleep_for_secs(2); //Spending time in base feature
6      if(A)
7          fpcsc::sleep_for_secs(1);
8      if(B)
9          fpcsc::sleep_for_secs(2);
10     if(C)
11         fpcsc::sleep_for_secs(1);
12     if(D)
13         fpcsc::sleep_for_secs(2);
14     if(A && B)
15         fpcsc::sleep_for_secs(2);
16     if(C && D)
17         fpcsc::sleep_for_secs(0);
18 }
```

In Algorithm Listing 2.1 we see a simple code snippet with some features that affect the performance in different ways, in Line 3 four features A, B, C, and D, are defined; each can be either *true* or *false* depending on the configuration chosen. Line 7, 9, 11, 13 will only be executed if the corresponding features are active. If this is the case, the system sleeps for the specified time. In Line 14, we have a feature interaction where {A, B} must be active in order for Line 15 to be executed, we would attribute the time spent in Line 15 to the feature interaction {A, B} and not to either feature alone. The performance-influence model for our system would look as follows:

$$\Pi = 1 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(D) + 2 \cdot c(A) \cdot c(B) + 0 \cdot c(C) \cdot c(D)$$

For simplicity, let us assume that the execution of the code takes no time at all, and we spend no time in any feature except the time specified in the *slee_for_seconds* function. The constant 1 here refers to β_0, the time we spend in our base feature in Line 5. If we decide on the configuration {A, B, C, D} the equation would look like this:

$$\begin{aligned}
\Pi &= 1 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(D) + 2 \cdot c(A) \cdot c(B) + 0 \cdot c(C) \cdot c(D) \\
\Pi &= 1 + 1 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 + 2 \cdot 0 + 2 \cdot 1 \cdot 1 + 0 \cdot 1 \cdot 0 \\
\Pi &= 1 + 1 + 2 + 1 + 2 + 2 \\
\Pi &= 9
\end{aligned}$$

Thus, for the configuration containing {A, B, C} our performance-influence model would yield an expected time of 9 second.

2.6 BUILDING A PERFORMANCE-INFLUCENCE MODEL USING BLACK-BOX ANALYSIS

We have introduced performance-influcence model to represent the influence of each feature and feature interaction on a system. In this section, we expand on this topic and introduce the *black-box analysis* a method to collect data to build performance-influcence model.

A black-box of a configurable system is conceptually simple, we execute a given system with a configuration, and after finishing, we receive an output. However, the critical part is that we are unaware of how the black-box produces the output. Since we cannot see inside the system, we need an approach that does not require this. Therefore, in a black-box analysis, we solve this issue by observing the machine on which the system is executed. In Section 2.6.1 we will explain this process in detail.

Subsequently, in Section 2.6.2, we use our black-box analysis data with *multiple linear regression* to build a performance-influcence model.

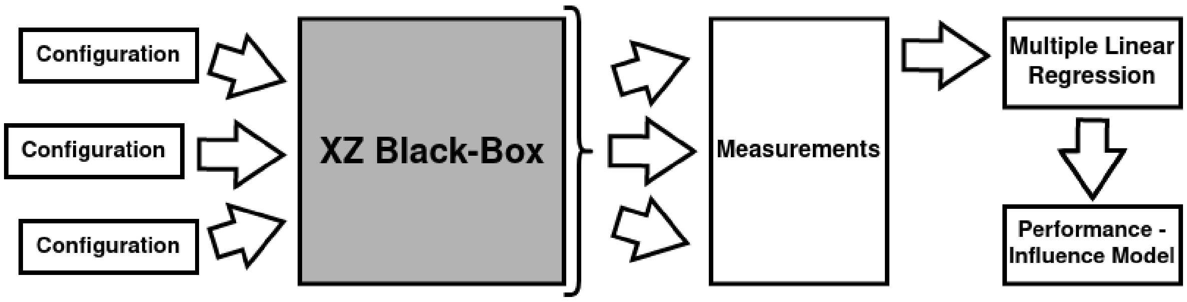

Figure 2.3: Process of using a black-box analysis to build a performance-influcence modelfor *XZ*.

In Figure 2.3, we use a black-box analysis for *XZ* to build a performance-influcence model. We start by focusing on finding the features that we are interested in. Then we build ourselves multiple configurations that hold different interactions between these features. Then, we run XZ as a black-box on each configuration; during the execution, we can measure different non-functional properties. In our case, we are interested in the performance of each feature. We repeat this process for each configuration during the black-box analysis and collect the measurements. Next, we use these measurements together with multiple linear regression to build a performance-influcence model.

2.6.1 *Black-Box Analysis*

Before we start analyzing the system, we first have to select the features we are interested in, since for most configurable systems it is not feasible to use the whole configuration space due to its size. This issue is called *combinatorial explosion* in Section 2.6.1.1 we explain how we deal with this problem.

After deciding which features are of interest to us, we can now turn to the question of how we analyze the system and collect the data we need to build a performance-influcence model

As shown in Figure 2.3, we cannot analyze how the system produces the output; therefore, we are limited to the non-functional properties we can observe from the outside. For this reason, we execute the system with each configuration and measure the property we are interested in, such as energy consumption, memory usage, and computational resources used.

2.6.1.1 *Combinatorial Explosion*

One of the larger problems we face when using black-box analysis is the issue of combinatorial explosion, which refers to the effect that when features increase linearly, the number of possible configuration increase exponentially [3].

Suppose we have a configurable system where each feature is a binary option. We also define that in this system, each feature is entirely independent of another (i.e., the system has no constraints, and selecting or deselecting one feature has no effect on other features). The number of unique configurations this system can produce is 2^n, where 2 refers to the type of feature options allowed, binary in our case, and n denotes the number of features.

Here is the problem, because all these different features can interact with each other in different ways, and for very small systems we can certainly brute-force our way by benchmarking all possible configuration, however this does not scale, and certainly not feasible for larger systems. For example, the Linux kernel contains 10'000 different features [8], for reference it is estimated that the universe contains about 10^{79} atoms, which is still less than the number of unique configurations a system with 263 features produces, it is already impossible to use brute-force to analyze such a system, let alone the Linux kernel.

Hence, we cannot fully explore the entire configuration space and must select a subset representing the system with high accuracy. For this purpose, we take advantage of the findings of Xu et al. [16], where they have shown that not all features are equally important and that up to 54,1% of features are rarely set by users. We use this information with our domain knowledge to extract the most important features we are interested in.

2.6.2 *Multiple Linear Regression*

After collecting all our measurements, we use them to build our performance-influcence model of the system. This section explains the reasoning behind using *multiple linear regression*. Afterward, in Section 2.6.2.1, we explain *ordinary least squares*, an estimator to calculate the coefficient of each term inside the performance-influcence model. While using *ordinary least squares*, we have to handle the problem of *multicollinear features*; we do this in Section 2.6.2.2. To reduce the degree of *multicollinear features* we introduce the *Variance Inflation Factor* in Section 2.6.2.3.

When building a performance-influcence modelfrom our black-box data we have the requirement that the model needs to be interpretable. While multiple methods to predict performance have been introduced, such as neural networks, they lack the interpretability of the model, which on contrary *multiple linear regression* provides.

To illustrate why interpretability is important lets inspect the following performance-influcence model:

$$\Pi_1 = -1000 + 1001 \cdot \mathit{Feature\_A} + 1002 \cdot \mathit{feature\_B} - 1000 \cdot \mathit{Feature\_A} \cdot \mathit{feature\_B}$$
$$\Pi_2 = 0 + 1 \cdot \mathit{Feature\_A} + 2 \cdot \mathit{feature\_B} + 0 \cdot \mathit{Feature\_A} \cdot \mathit{feature\_B}$$

Under the condition that either *Feature_A* or *Feature_B* needs to be selected, Π_1 and Π_2 predict for any configuration the same amount of time, however, if we take a close look at how Π_1 assigns the influence of each feature, we can see that this is not interpretable. Here Π_1 assigns to the *Base* feature an influence of -1000 thousand, which does not make sense since the time spent in the *Base* feature can not be negative. This also leads to Π_1 assigning unrealistic amounts of time to features or feature interactions.

We use the following formula for linear regression for matrices [5]:

$$\begin{aligned} Y &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... \beta_n x_n + \epsilon \qquad (2.2) \\ Y &= X\beta + \epsilon \end{aligned}$$

$$\begin{aligned} Y &= \textit{Dependent variable} \\ X &= \textit{Independent variable} \\ \beta &= \textit{Regression coefficient} \\ \epsilon &= \textit{Error} \end{aligned}$$

The model is used to estimate the relationships between the independent variables and the dependent variable. In our case, Y is a vector with n elements containing the output of our black-box model, i.e., the measurements for each configuration in our set of configurations $\mathcal{C}$.

Our independent variable X is an $n \times m$ matrix, where n is the number of configurations used, and m is the number of features and feature interactions across all configurations. To accommodate feature interactions in this linear model, we add a term for each interaction we want to include. For example, if we consider the interaction between features x_i and x_j we add the term $\beta_k x_i x_j$.

We are interested in the values of the coefficients β, since they quantify the influence of each feature or feature interaction on the whole system. In addition, β_0 denotes the intercept, representing the influence of the base code, meaning the part of the code executed regardless of the chosen configuration.

The value of each feature in the matrix is 1 if the feature is selected or 0 if it is not selected. If we have numerical features with l different options, we split these features into l binary features and encode them as an alternative group in our feature model.

All our measurements have a possible error represented by ϵ [6].

2.6.2.1 *Ordinary Least Squares*

Now that we have seen the general formula of multiple linear regression and know what the different components stand for, we still need to figure out how to calculate the regression coefficient β, the values that tell us the influence of each feature.

For this purpose, we use the ordinary least squares estimator, which is optimal for the class of linear unbiased estimators, but is unreliable when the independent variables X contains a high degree of multicollinearity, we explain this in detail in Section 2.6.2.2. The principal of ordinary least squares is to minimize the sum of the squared residuals, where the residual is the difference between the predicted value of the estimator and the actual value. [6]

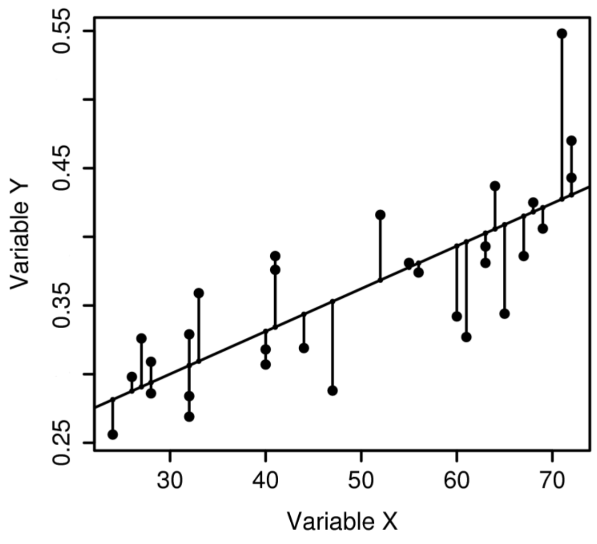

Figure 2.4: Ordinary Least Squares regression model residuals [2]

We see an illustration of an ordinary least squares estimator for a linear regression model in Figure 2.4, where we have only one variable X and the corresponding measurements Y, allowing us to compute the single regressor for this linear regression model.

To compute the regression coefficients using ordinary least squares, the following formula is used [6]:

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y \tag{2.3}$$

$$\hat{\beta} = \textit{Ordinary Least Squares estimator}$$
$$\top = \textit{Matrix Transposed}$$

Now $\hat{\beta}$ contains the regression coefficient we are interested in.

As an example, we can refer to our code from Listing 2.1 and sample some configurations to build a multiple linear regression model using ordinary least squares.

2 Visited at 06.03.2023, https://datajobs.com/data-science-repo/OLS-Regression-[GD-Hutcheson].pdf

Base	A	B	C	A ∧ B	A ∧ C	**Π(∗)**
1	0	0	0	0	0	**1**
1	1	0	0	0	0	**2**
1	0	1	0	0	0	**3**
1	0	0	1	0	0	**2**
1	1	1	0	1	0	**6**
1	1	0	1	0	1	**3**
1	0	1	1	0	0	**4**
1	1	1	1	1	1	**7**

Table 2.1: Configuration samples of Listing 2.1

Using the configuration samples from Table 2.2 we can now determine X and Y:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, Y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 6 \\ 3 \\ 4 \\ 7 \end{bmatrix}$$

Using the ordinary least squares Equation 2.6.2.1 we obtain the following results:

$$\hat{\beta} = 1 + \begin{bmatrix} 0,00 \\ 1,00 \\ 2,00 \\ 1,00 \\ 2,00 \\ 0,00 \end{bmatrix} \tag{2.4}$$

All values have been rounded to 2 decimal places. We can see that all values have been assigned correctly.

Now we can use the values to create the performance-influcence model:

$$\Pi = 1 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(A) \cdot B + 0 \cdot c(A) \cdot c(C) \tag{2.5}$$

2.6.2.2 *Multicollinear Features*

We already mentioned that ordinary least squares is optimal as long as our configurations do not contain multicollinearity. The reason for that is in presence of multicollinearity the variance of the estimator inflates, which in result hurts the interpretability of the model. We call features multicollinear when there exist a near linear dependency between these features, meaning we can nearly represent one feature as a combination of different features and the feature does only provide a small amount of new information to the system [6]. If a feature does not provide any new information to the system, then we speak of perfect multicollinearity.

As an example take a look at a performance-influcence model for some the monthly expenses of a student:

$$monthly\_expenses = food + take\_out$$

Now this shows multicollinearity between the features *food* and *take_out*, since *take_out* is already present in the cost of *food*, furthermore it is a case of perfect multicollinearity, since the feature does not provide any new information.

One way multicollinearity is introduced into a system is by using alternative groups, since the selection of a feature in the alternative group can be expressed by the combination of all other feature. [4]

Base	A	B	C	$\Pi(*)$
1	1	0	0	**5**
1	0	1	0	**10**
1	0	0	1	**15**

Table 2.2: Configuration example illustrating multicollinearity in an alternative group. Where $\Pi(*)$ is the sum of all selected features inside the row

Now consider the example of Table 2.2, where we see a configuration example that contains multicollinear features due to an alternative group. The example contains a mandatory *Base* feature and 3 features in an alternative *A*, *B*, and *C*. Now we can always model the present of a feature in an alternative group due to the absence of other feature, here for feature *C* to be selected, *B* and *A* needs to be deselected.

$$\Pi_0(c) = 0 + 5 \cdot c(A) + 10 \cdot c(B) + 20 \cdot c(C)$$
$$\Pi_1(c) = 5 + 10 \cdot c(A) + 5 \cdot c(B) + 20 \cdot c(C)$$
$$\Pi_2(c) = 8 + 20 \cdot c(A) + 10 \cdot c(B) + 7 \cdot c(C)$$

This leads to multiple performance-influcence model that are accurate with respect to the individual measurement, but make completely different statements when compared.

Another way multicollinearity is introduced into the system is to have features that are mandatory or connected by a condition. If we have features that are mandatory, we cannot distinguish these features with our black-box analysis because they are always selected together, and we cannot determine the extent to which each feature influences the system. [4]

In Table 2.3, we can see the predictions each of the 3 performance-influcence modelmake for the configurations $\{Base\}$. Now Π_0, Π_1, and Π_2, assign completely different values to

Π	Base	A	B	C	$\Pi(\{Base\})$
Π_0	0	5	10	20	**0**
Π_1	5	10	5	20	**5**
Π_2	8	20	10	7	**8**

Table 2.3: Performance predictions of Table 2.2

the *Base* feature, which makes it impossible for us, the user, to identify the correct value. The reason is that both *Base* and a feature of the alternative group are mandatory; therefore, we cannot measure one without the presence of the other. Hence, the values of *Base* or the alternative group feature can be set in any ratio as long as the sum of the two values equals the measured time.

When we decide on which features to use in our configuration set, we use our domain knowledge of the system to reduce the amount multicollinear features to a minimum.

2.6.2.3 *Variance Inflation Factor*

In reality, multicollinearity is often unavoidable in configurable systems, when we want to model the influence of a feature interaction between features A and B we introduce a term $A \cdot B$, this feature interaction is only selected when feature A and B are selected. Due to that reason we can not remove terms that introduce multicollinearity, however we can remove perfect multicollinear since they do not provide our system with new information.

To check for perfect multicollinearity we use the variance inflation factor (VIF), where a VIF factor of inf indicates that there is perfect multicollinearity between features of feature interactions [4].

We compute the VIF using the following equation:

$$VIF_j = \frac{1}{1 - R_j^2} \tag{2.6}$$

$$R_j^2 = 1 - \frac{\sum\limits_{\forall c \in \mathcal{T}} (c(o_j) - \bar{c}(o_j))^2}{\sum\limits_{\forall c \in \mathcal{T}} (c(o_j) - f_j(c \setminus o_j))^2} \tag{2.7}$$

Where $\mathcal{T}$ is the trainings set containing j features o_j. The VIF$_j$ can be calculated for each feature by using the coefficient of determination R^2. To do this, we need to calculate R_j^2 for each feature o_j, fitting a linear regression function f_j to predict whether o_j is selected in the configuration $c \setminus o_j$, using all other features as predictors and the overall mean $\bar{c}(o_j)$ [4].

2.7 WHITE-BOX MODEL

A black-box analysis is very useful for systems where we do not have access to the source code, but if we do, we take advantage of this additional information, by using a white-box analysis. While a black-box analysis measures the time we spend inside the system from start to finish, a white-box analysis archives a higher level of granularity by using different algorithms to determine how much time we spend in each feature.

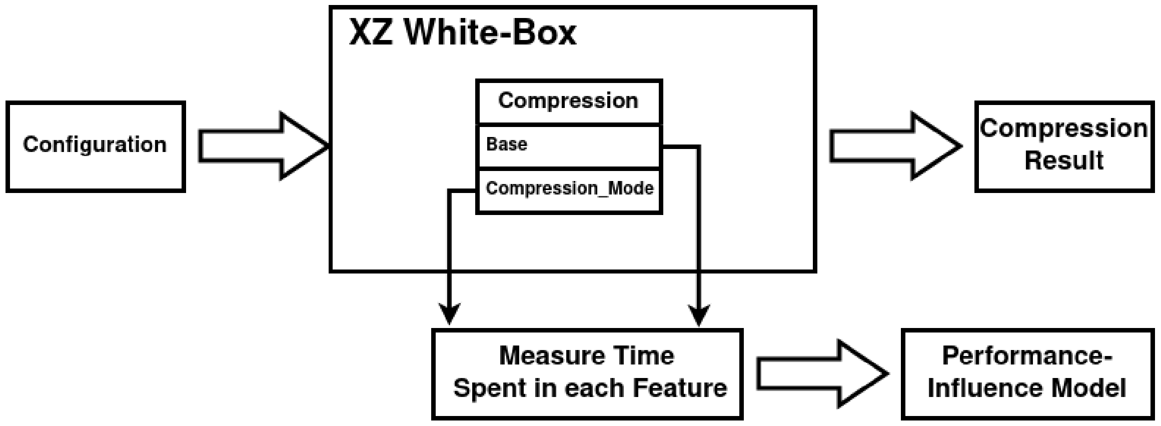

Figure 2.5: Process of using a black-box analysis to build a performance-influcence model for *XZ*.

We now use a white-box analysis, to analyze the example system from Figure 2.1. In Figure 2.5, we have access to the source code of *XZ*; our first step is to manually analyze the code and find the variables that implement configurability inside the system. In this example, we found the feature *Base* and *compression_mode*. During compression, we use our white-box analysis to measure the time spent by *XZ* in the different features. This example measures the time spent in *Base* and *Compression_Mode*. After the system finished the process, we collected all the measurements, which we then used to build a performance-influcence model for this configuration.

In Section 2.7.1, we explain the current state-of-the-art strategies used to analyze systems using white-box. Subsequently, in Section 2.7.2 we introduce VaRA, the analyzing tool we use, and its underlying principles. In Section 2.7.3, we explain how to build the performance-influcence model using the white-box data.

2.7.1 *Strategies*

When analyzing systems using a white-box approach, different strategies have been introduced. In this chapter, we explain three different strategies, ConfigCrusher and Comprex, both model configurability on a feature level, whereas Weber et al. introduce a strategy that models configurability on a method level.

Velez et al. introduced us to ConfigCrusher [13], a white-box analysis that uses static data-flow analysis to see how features influence variables and the control flow of the system. In addition, ConfigCrusher leverages three insights about configurable systems from previous works, namely irrelevance, orthogonality, and low interaction degree. They use irrelevance to identify features relevant to the system's data flow, reducing the number of configurations required to analyze the system. They use orthogonality to identify features that do not interact with each other and, therefore, can be measured together. Since only a few features interact, ConfigCrusher focuses on the configurations with interacting features to reduce the number of configurations to be analyzed. From these findings, two techniques are developed, namely compression and composition. They use compression to reduce the number of configurations required to analyze the system by simultaneously analyzing regions that are independent of each other so that they can use a single configuration to analyze different features. Whereas composition takes advantage of the fact that performance-influcence model can be built compositionally by building a performance-influcence model for each region separately and then assembling all local performance-influcence model into one model for the entire system. After using the data-flow analysis to generate a control flow graph and a statement influcence map, which maps statements to the configuration options that influence that statement. Afterward, they use both the control flow graph and statement influcence map to instrument the regions in the system that correspond to features and execute the instrumented system to track execution time of each feature. From these measurements, they build the performance-influcence model for the system.

Velez et al. introduced COMPREX [14], an approach that builds on CONFIGCRUSHER but uses an iterative dynamic taint analysis instead of static analysis to determine how and to what extent features affect the control flow of the given system. By doing so, they identify which code regions are influenced by which configurations and, during execution measure the time spent in these regions to then build the performance-influcence model.

Compared to CONFIGCRUSHER and COMPREX, Weber et al. [15] uses a profiling approach to generate performance-influence models that analyze configurability on a method level. To achieve this, they first used JPROFLIER, a coarse-grained profiler, to learn a performance-influence model for every method that has been learned successfully. To identify the hard-to-learn methods, they use filtering techniques and then KIEKER, a fine-grained profiler, to learn these methods. At the end, for each method, they obtain a performance-influcence model that shows how strong each feature influences the performance of that method.

2.7.2 *VaRA*

To analyze the system we are interested in, we use VARA, a framework for analyzing configurable software systems that is built on LLVM. In addition, we use the VARA TOOL SUITE[3], which provides us with a framework that supports us when analyzing configurable systems using VARA.

The purpose of VARA is to provide various analyses for systems where the user only needs to focus on the high-level conceptual information of the system, while VARA handles the low-level-details. Since VARA is built on top of LLVM, it is able to analyze systems written in languages that can be compiled by LLVM, such as C, C++ or Rust [9].

2.7.2.1 *Feature Region*

To analyze configurable systems, VARA identifies code regions associated with a feature or feature interaction; these regions are called *feature region*. The first step for VARA to be able to detect these regions is to find the feature variable that represents the features inside the code and mark them as feature variables. A feature region is, therefore, a part of the code that is executed depending on the value of the feature variable. Whenever we detect a feature region, we inject code into the system to measure the time spent in these regions.

3 Visited at 14.03.2022 `https://vara.readthedocs.io`

Listing 2.2: Feature region example

```
1 void encrypt() {
2     bool Encryption; //Feature Variable
3     assign_feature(Encryption); //Assigns true if Encryption is selected
4 
5     if(Encryption)
6         foo();
7     else
8         bar();
9 }
```

In Listing 2.2, we can see the structure of the *Encryption* feature region. The feature variable in Line 2 represents whether *Encryption* is selected or deselected; depending on *Encryption*, either the *then* or *else* branch is executed. Together, these two branches form an *Encryption* feature region.

VaRA uses different detection approaches to identify these feature regions, both of which use a *taint analysis*.

2.7.2.2 *Taint Analysis*

Before explaining VaRA feature region detection, we explain the concept behind a taint analysis since both approaches build upon this analysis.

The common usage of a taint analysis is in cybersecurity, where we trace the data flow of data that originates from an outsider or an untrusted source; this input is labeled a *tainted*. We then track how this tainted data is propagated through the system until it reaches a point where the data is again accessible to the outside. We call the access where the data is injected *sources*, and the points where the data is extracted *sinks* [10].

VaRA uses taint analysis, too; however, in contrast to the common usage of a taint analysis, we are not interested in finding the sinks where data is leaked to the outside. However, instead, we are interested whenever instructions access our feature. For the taint analysis, VaRA uses feature variables as sources and instruction as sinks. Whenever an instruction accesses a feature variable, this instruction is tainted by that feature [7].

VaRA uses the taint analysis in both approaches, the *if approach* and *dominator approach*, to detect feature regions.

IF APPROACH We call the first approach the *if approach*. Here, whenever a feature variable that was declared as a source is accessed in the condition of an if statement, then the *then* case and the *else* case are marked as a feature region [17].

In Listing 2.2 in Line 5 an if condition access the feature variable *Encryption*, here the if approach would mark Line 6 and Line 8 as a feature region of *Encryption*.

DOMINATOR APPROACH We call the second approach the *Dominator Approach*, in here, VaRA uses domination relationships to identify feature regions.

To do this, VaRA works with basic blocks of the control flow graph, whereas a basic block is an instruction sequence that contains an entry label, which is the entry point for this code, and a terminator at the end, which determine the control flow of the block. An example of a terminator is an if condition that uses a feature variable.

To discover these domination relationships, VaRA checks out which basic block dominates other basic blocks with dependent terminator instructions. A basic block BB_1 dominates a different basic block BB_2 when the terminator of BB_1 decides if BB_2 is executed. Now instructions in BB_2 would depend on the terminator instruction of BB_1. The feature for the feature region of BB_2 is the feature that corresponds to the feature variable used by the terminator of BB_2 [17].

2.7.2.3 *Locating feature variables*

VaRA is not able to automatically detect which variables represent features. Therefore, we provide VaRA with a feature model as a XML file containing every feature's location inside the code.

Listing 2.3: Feature model of Listing 2.2 in XML

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE vm SYSTEM "vm.dtd">
3  <vm name="SingleLocalSingle" root="root">
4      <binaryOptions>
5          <configurationOption>
6          <name>Encryption</name>
7          <parent></parent>
8          <optional>True</optional>
9          <locations>
10             <sourceRange category="necessary">
11                 <path>src/my_encryption.c</path>
12                 <start>
13                     <line>2</line>
14                     <column>10</column>
15                 </start>
16                 <end>
17                     <line>2</line>
18                     <column>19</column>
19                 </end>
20                 </sourceRange>
21         </locations>
22         </configurationOption>
23     </binaryOptions>
24     <numericOptions></numericOptions>
25     <booleanConstraints/>
26 </vm>
```

As an example Listing 2.3 encodes Listing 2.2 as a feature model. Inside the XML we differentiate between two kinds of feature, <*binaryOptions*> in Line 4 and <*numericOptions*> in Line 24, we declare each feature as a child of either those two tags. We encapsule every feature we want to track by using the <*configurationOption*> tag in Line 5, inside we can define the <*name*> of the feature, in Line 6 its parent, and if the feature is optional, but most importantly in Line 9 we define the <*location*> to specify where the feature variable is defined inside the code. The location tag contains, at least one <*sourceRange*> tag, in which we specify the feature variable that is associated with the feature, however a location can contain multiple source ranges, whereas the feature is implemented by multiple feature variables. Each <*sourceRange*> tag contains a <*path*> tag that specifies the location of the file containing the feature variable. After specifying the path, we need even further to specify the location of the feature variable inside the file. For this, we see in Line 12 the <*start*> tag and in Line 16 the <*end*> tag, inside both, we specify the <*line*> and <*column*> for where the feature variable starts end ends.

For Listing 2.2, we would specify that the feature variable *Encryption* is in Line 2 and begins in *column 10* and ends in *column 19*.

After identifying all the feature regions of each feature, we run *VaRA*, together with the feature model, to locate all feature regions [9].

2.7.2.4 *VaRA Tool Suite*

We use VaRA in combination with the VaRA Tool Suite, a framework written in python that assists us when analyzing configurable software systems using VaRA by specifying an *experiment* to analyze a particular *project*. The result our analysis produces is written into a *report*. We emphasize that both experiments and projects are independent, allowing us to use different experiments to analyze a project in different ways. To start the analysis, we use the command-line tool *vara-run*, which specifies which experiments we want to run for which project.

PROJECT A *project* inside the VaRA Tool Suite specifies the configurable software system we want to analyze. Here we define how we build the system or which version of the system we want to study.

In our example of Figure 2.5, the project would be XZ and would specify which version of XZ we want to analyze and how the binary is build.

EXPERIMENT Inside VaRA Tool Suite an *experiment* specifies how we want to analyze a software project. This allows us to make the analysis easy to reproduce and repeat.

In Figure 2.5 we want to analyze XZ, in the experiment we specify that we want to use VaRA and the different steps we want to execute, like which report we want to use.

REPORT VaRA Tool Suite offers us a variety of *reports* which are used to store different data, which we generate during the analysis.

For Figure 2.5 we use the trace event format report which we explain in Section 2.7.3

2.7.3 *Trace Event Format*

When we use VaRA the code of the configurable software system gets instrumented to measure the time spent inside each feature region. During the execution of the whole system, we enter various feature regions multiple times. We collect all these measurements in a trace event format (TEF) [4] report.

Every time we enter or leave a feature region, a trace event is triggered that contains the following information:

```
"name": "Base",
"cat": "Feature",
"ph": "B",
"ts": 0,
"pid": 726119,
"tid": 726119,
"args": {
    "ID": 0
}
```

Listing 2.4: Trace event

In Listing 2.4, we see how a trace event is structured. Each trace event contains a *name* that refers to the names of the features that affect this region, *ph* represents the event type, while *B* signals the beginning and *E* the end of an event. All events contain a timestamp *ts* that refers to the time in milliseconds when the event began or ended. *Args* contain an *ID* that refers to each event, so the event that initiates the beginning *E* of a region has the same *ID* as the event that signals when we leave the region with *E*.

When a trace event for a feature region begins before a trace event for a different feature ends, we say these features interact. As an example, if we have two features, *foo* and *bar*, where the trace event of *foo* starts at *0* and ends at *4* seconds and the trace event of *bar* starts at *1* and ends at *3* seconds. We spent *2* seconds in feature *foo*, from *0 to 1* and *3 to 4*, but since we did not leave the feature region *foo* before entering *bar* we have an interaction between these features. Therefore, we spent *2* seconds inside the feature interaction *(foo, bar)* instead of the feature *bar*.

Since we are only interested in which features interact, we ignore the order in which the interaction happens, which means the previous interaction of *(foo, bar)* is the same as *(bar, foo)*. This also makes sense in the context of performance-influcence model since we defined the interaction of features as a product, where the time spent in that feature interaction is added if all features of that interaction are selected, in this case, $2 \cdot c(foo) \cdot c(bar)$. Since the product is commutative, we also ignore the order in which the features interact inside the performance-influcence model.

When we have nested trace events of the same feature, we do not add a feature interaction between the same features, due to the reason that inside the performance-influcence model the feature interaction $2 \cdot c(foo) \cdot c(foo)$ is the same as $2 \cdot c(foo)$.

After the system finishes its execution, we have to transform the TEF report into a performance-influcence model by aggregating all events. To do so, we sum up all the time spent in each region and attribute this time to the feature that influenced that region.

We calculate the time spent on each feature as follows:

4 Visited at 15.03.2022
https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU

$$time(feature) = \sum_{event \in feature} \left(\sum_{(ts_E, ts_B) \in event} ts_E - ts_B \right) \tag{2.8}$$

$$feature\_coefficients = \sum_{feature \in TEFReport} time(feature) \tag{2.9}$$

In Equation 2.8, we calculate the time spent for the given feature. For this equation, we define that a *feature* is a list of *events*, whereas each event is a pair of trace events representing when the feature region is entered and when it is left. To calculate the time spent in the region, we compute $ts_E - ts_B$.

Equation 2.9 is a sequence of the total time spent in each feature or feature interaction we measured in the *TEF Report*. Now that we obtained all the coefficients that represent the influence of each feature and feature interaction, we use them to build the performance-influcence model.

3 GROUND TRUTH

In this chapter we introduce a ground truth to establish that both black-box analysis from Section 2.6 and white-box analysis from Section 2.7 conceptually work.

To do so we design multiple configurable systems that test both analyses in different scenrios, afterwards we evaluate them using research questions 1.1. Since we design these systems ourselves we have a baseline that we use to manually build a performance-influcence modelfor each system. All the different system will be designed with different focuses in mind such as, a system that includes multicollinearity to see in what extent they influence the analyses.

4 EXPERIMENTS

This chapter describes the main experiments that are conducted for this thesis.

5 EVALUATION

This chapter evaluates the thesis core claims.

5.1 RESULTS

In this section, present the results of your thesis.

5.2 DISCUSSION

In this section, discuss your results.

5.3 THREATS TO VALIDITY

In this section, discuss the threats to internal and external validity.

6 RELATED WORK

This chapter presents related work.

For example, **KG:SME06** investigated . . .

ABKS:BOOK2013 analyzed . . .

In earlier work [**KAK:GPCE09**, **ABKS:BOOK2013**], they have shown . . .

7 CONCLUDING REMARKS

7.1 CONCLUSION

7.2 FUTURE WORK

A APPENDIX

This is the Appendix. Add further sections for your appendices here.

BIBLIOGRAPHY

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013, pp. 26–36. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013, pp. 243–244. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[4] Johannes Dorn, Sven Apel, and Norbert Siegmund. “Mastering Uncertainty in Performance Estimations of Configurable Software Systems.” In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 684–696. DOI: 10.1145/3324884.3416620. URL: https://doi.org/10.1145/3324884.3416620.

[5] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. “Predicting Performance of Software Configurations: There is no Silver Bullet.” In: *CoRR* abs/1911.12643 (2019). arXiv: 1911.12643. URL: http://arxiv.org/abs/1911.12643.

[6] Jürgen Groß. *Linear Regression*. Springer Berlin Heidelberg, 2003. DOI: 10.1007/978-3-642-55864-1. URL: https://doi.org/10.1007/978-3-642-55864-1.

[7] Janik Keller. *"Feature Taint Analysis: How Precise can VaRA Track the Influence of Feature Variables in Real-World Programs"*. Germany, 2023.

[8] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. “Evolution of the Linux Kernel Variability Model.” In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 136–150. DOI: 10.1007/978-3-642-15579-6_10. URL: https://doi.org/10.1007/978-3-642-15579-6_10.

[9] Florian Sattler. ““A Variability-Aware Feature-Region Analyzer in LLVM.”” MA thesis. Germany: University of Passau, 2017.

[10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).” In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 317–331. DOI: 10.1109/SP.2010.26. URL: https://doi.org/10.1109/SP.2010.26.

[11] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-influence models for highly configurable systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 284–294. DOI: 10.1145/2786805.2786845. URL: https://doi.org/10.1145/2786805.2786845.

[12] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. "Approaching Non-functional Properties of Software Product Lines: Learning from Products." In: *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. Ed. by Jun Han and Tran Dan Thu. IEEE Computer Society, 2010, pp. 147–155. DOI: 10.1109/APSEC.2010.26. URL: https://doi.org/10.1109/APSEC.2010.26.

[13] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: towards white-box performance analysis for configurable systems." In: *Autom. Softw. Eng.* 27.3 (2020), pp. 265–300. DOI: 10.1007/s10515-020-00273-8. URL: https://doi.org/10.1007/s10515-020-00273-8.

[14] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems." In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1072–1084. DOI: 10.1109/ICSE43902.2021.00100. URL: https://doi.org/10.1109/ICSE43902.2021.00100.

[15] Max Weber, Sven Apel, and Norbert Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach (Replication Package)." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 232–233. DOI: 10.1109/ICSE-Companion52605.2021.00107.

[16] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 307–319. DOI: 10.1145/2786805.2786852. URL: https://doi.org/10.1145/2786805.2786852.

[17] Tom Zahlbach. *"Finding Feature-Dependent Code: A Study on Different Feature-Region Detection Approaches"*. Germany, 2023.