Bachelor's Thesis

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

MANUEL MESSERIG

March 9, 2023

Advisor:
Florian Sattler Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Second Examiner Affiliation 2

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 major reason for that is that we as the developer want to give the user the flexibility of configuring the system to their needs, by offering them to turn functionality on and off.

However, nowadays configurable systems offer various features which result in a large amount of unique configurations. All of these different configurations result in different behaviors of the system, one of the metrics shareholder are interested is the running times of system under each configuration and in what capacity each feature influences the system. To analyze the influence of these features we will introduce two different approaches, a white-box and black-box approach. We will use the data generated by those approaches to build performance-influence models and use them to analyze and compare both approaches.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

IDE Integrated Development Environment

1 INTRODUCTION

Modern software systems are designed to be configurable, we want to provide flexibility to the user by offering 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 software system would be the Linux kernel, whose code base itself contains over 6,000,000 lines of code containing more than 10,000 optional features [9]. All these optional features allow you to create an operating system that meets your needs. To effectively analyze such systems, we present two different analyses - a white-box and a black box-analysis.

This results in many features that affect the system in different ways, to keep track of all these features and their interactions, we will use a feature model, which is essentially a tree that can contain different kinds of constraints to visualize the relationships between features in a configurable system [8].

In the black-box analysis, we have only one system that receives an input, in our case a selection of several features. The system is then be executed with our configuration. During execution, we can measure various metrics, such as memory and energy consumption. Whereas we will focus on the measured execution time.

In our white-box model, we have more information because we know the inner workings of the system itself, i.e., we know which code contributes to which feature and therefore measure the time we spend in each feature by summing up the time spent in the code when its executed.

Our two models generate different types of data that we still need to compare and evaluate, for which we use performance- influence models. These models represent our configurable system as a polynomial, where each term represents either a feature or an interaction of features [10]. We build these models by using the data generated by the white-box 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.

After confirming the validity of our models, we will apply both models to real world systems, such as the compression tool XZ. We will apply both models on the same system and data then repeat the experiment 30 times for each configuration to reduce external factors such as measurement noise.

1.1 GOALS OF THIS THESIS

In this thesis the main focus is about comparing performance-influence models between white-box and black-box models. Before we can even compare these models, we need to check whether they are able to detect interactions between features, and if so, how accurate they are. If they can identify the interactions between the features, we can start to compare them.

First and foremost, we are interested in whether both models come to the same conclusions, after all, they have both analyzed the same system. 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. If they do not reach the same conclusion, we analyze the reason for the differences between them and examine whether one model performs particularly poorly in certain cases and why this is so. We answer 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 : Can we identify the reasons for similarities or differences between performance models?

2 BACKGROUND

2.1 CONFIGURABLE SYSTEMS

Nearly, all modern software systems are configurable, there are multiple reasons for that. We want to give the user flexibility, by offering them to turn functionality on and off. In addition to that a configurable software system satisfies the demand of multiple user by offering a single software system that contains multiple features. [14]

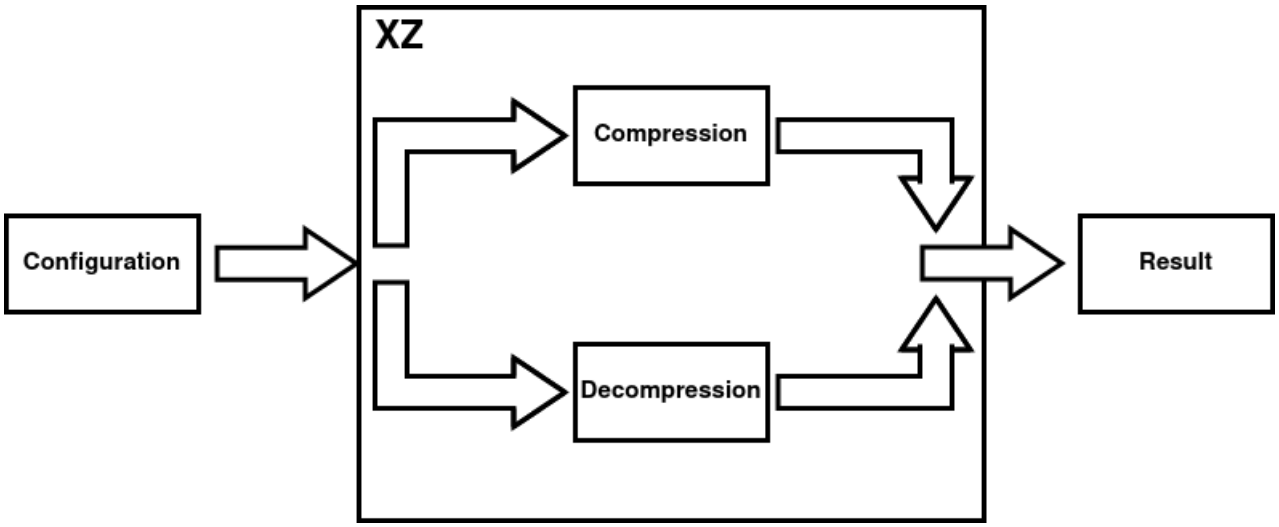

Figure 2.1: Scaled down version of xz

As an example let us inspect the compression tool xz [1]. In Figure 2.1 we can see a scaled down version of xz, which contains two main functions (In reality many more), encryption and decryption, it is up to the user to decide what he needs, but regardless his choice both functions are contained in a single software.

2.2 FEATURES AND CONFIGURATIONS

In the previous section we already said that we want to be able to turn functionality on and off, to achieve this we use features.

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 incorporate both concepts Apel et.al. [1, p. 18] introduced us to the following definition:

1 https://linux.die.net/man/1/xz

"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"

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 2.1 both, encryption and decryption, are unique features, they refer to a piece of functionality of the system and the implementation. A configuration is a set of features, where the features selected in the configuration decide which functionality of the system is turned on or off. The set of all valid configurations of a system is called the configuration space, it depicts the whole functionality of the system. In 2.1 the configuration we select, decides if we want to have encryption or decryption enabled.

2.3 FEATURE MODEL

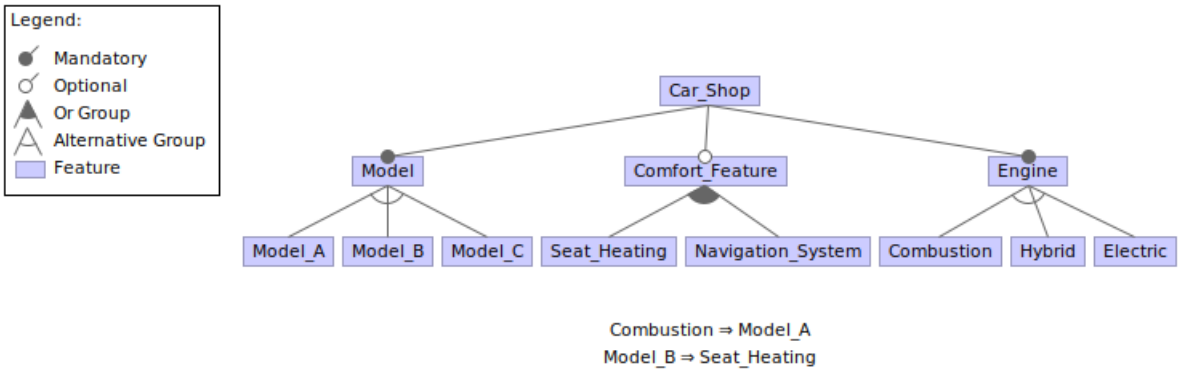

Figure 2.2: A feature model of a car dealership

A configurable system often contains numerous features, all of which may interact with one each other or have different dependencies. To keep track of the system, we introduce feature model [2]. An example of a feature model for a car dealership can be found in 2.2

A feature model is essentially a tree that models a configurable system, where each node in that tree represent a specific feature, while the root represent the system itself. When a feature is selected, it implies that also the parent of that features must also be selected as well, the further down the tree we go, the more specific features become. In 2.2 we see Engine is a feature whose children a concrete types of engines, in particular Combustion, Hybrid and Electric.

Each feature can contain a graphical notation indicating whether the feature is mandatory or optional, if the feature is mandatory it is indicated with a black bubble, and if it is optional it is displayed with an empty bubble, in 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 a optional since they are not necessary for a car to function.

In addition to mandatory and optional features, there are also alternative and choice groups. A parent can have one of these groups, an alternative group is marked with an empty half circle and an optional group with a filled circle. When a selection group is used, one feature needs to be selected, but others can be selected as well, a choice group

corresponds to the logical or operator. In an alternative group, only one feature can be selected, if more than one feature is selected the configuration is invalid. In 2.2 we see that Engine has a alternative group, which makes sense since each car can only contain one Engine, where it makes sense that Comfort_Feature contains a choice group, you can have a navigation system and seat heating in a car without conflict.

In addition, a feature model may contain various constraints that need to be satisfied, these constraints are defined as boolean algebra. In 2.2 we see that there are two constraints, $Combustion \implies Model\_A$ and $Model\_B \implies Seat\_Heating$, the reasons for such constraints could be that Model_B is a luxury model where it only gets shipped with seat heating.

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

2.4 PERFORMANCE-INFLUENCE MODELS

Both of white-box and black-box approach produce data that differ from each other, so we cannot compare them directly, for this reason, we introduce performance-influence models. We generated such a model for each dataset generate by both white-box and black-box, then we are able to compare and analyze both model. We use the formal definitions from the paper "Performance-Influence Models for Highly Configurable Systems" by Sven Apel et al. [10].

A feature performance model is a polynomial that consists of several terms, each term representing either a feature or an interaction of features and the influence they have onto the system itself. The sum of all terms represent the time the performance model predicts given a configuration of features. All our features are encoded as binary features, meaning they can either be turned on or off. Formally, let $\mathcal{O}$ be the set of all configuration options and $\mathcal{C}$ the set of all configuration, 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, and if we deselect a feature $c(o) = 0$. In general, a performance-influence model is a function Π that maps configurations $\mathcal{C}$ to an estimated time prediction, therefore $\Pi : \mathcal{C} \implies \mathbb{R}$.

We distinguish between single features o denoted as ϕ_o and feature interactions $i...j$ denoted as $\Phi_{i...j}$. With these definitions, we can now define how a performance-influence model formally looks like:

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

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 individual feature and $\sum_{i...j \in \mathcal{O}} \Phi_{i...j}(c(i)...c(j))$ is the sum of each feature interaction.

In Algorithm 1 we see a simple code snippet with some features that affect the runtime in different ways, in line 2 four features A, B, C and D are defined, each of them can be either True or False depending on the configuration chosen. The code in line 5,7,9,11 will is

Algorithm 1 Feature Interaction

```
1:  procedure FEATURE INTERACTION
2:      A,B,C,D ← True or False
3:      sleep_for_seconds(1) #Lets the system sleep for 1 second
4:      if A then
5:          sleep_for_seconds(1)
6:      if B then
7:          sleep_for_seconds(2)
8:      if C then
9:          sleep_for_seconds(1)
10:     if D then
11:         sleep_for_seconds(2)
12:     if A and B then
13:         sleep_for_seconds(2)
14:     if C and D then
15:         sleep_for_seconds(0)
```

executed only if the corresponding features are active. If this is the case, the system sleeps for the specified time. In lines 12 and 14, we have two feature interactions where A and B must be active, so in order for line 13 to executed, we would attribute the time spend in line 13 to the feature interaction {A, B} and not to either feature alone. The performance-influence model for our system would look like this:

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

For simplicity, let us assume that the execution of the code takes no time at all and that no additional time is spent on any feature except the specified in the *textitslee_for_seconds* function. The constant 1 here refers to β_0, which is the time we spend in our base feature in line 2. If we decide to turn on features a, b and c 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.

We build a performance-influence model from the date collected by both approaches, the challenge being to produce accurate data to produce an accurate model. Since we apply both approaches to the same data, we expect them to produce a comparable performance-influence model. We explain how we collect and work with the data in Chapter 2.5 and 2.6

2.5 BLACK-BOX ANALYSIS

When developing a software system, it often happens that we need different frameworks or other systems to develop ours. While it would be possible to develop these systems ourselves, in most cases this is inefficient and slows down the release of our system. To speed up this process, we can use other systems that are already available to us. However, the question is whether these systems meet our requirements that are not related functionality, such as performance.

For this purpose, we can perform a black-box analysis of the system. A black-box analysis is conceptually simple, given a configurable system, we select a set of configurations that contains the feature we are interested in. We run the system with different configuration from our set and during execution we measure the non-functional properties we can observe. Next, we use these measurements to build a performance-influcence model, that becomes more accurate as the number of measured configurations measured increases. In the end, we have a performance-influcence model that represents our system, created with a black-boy analysis.

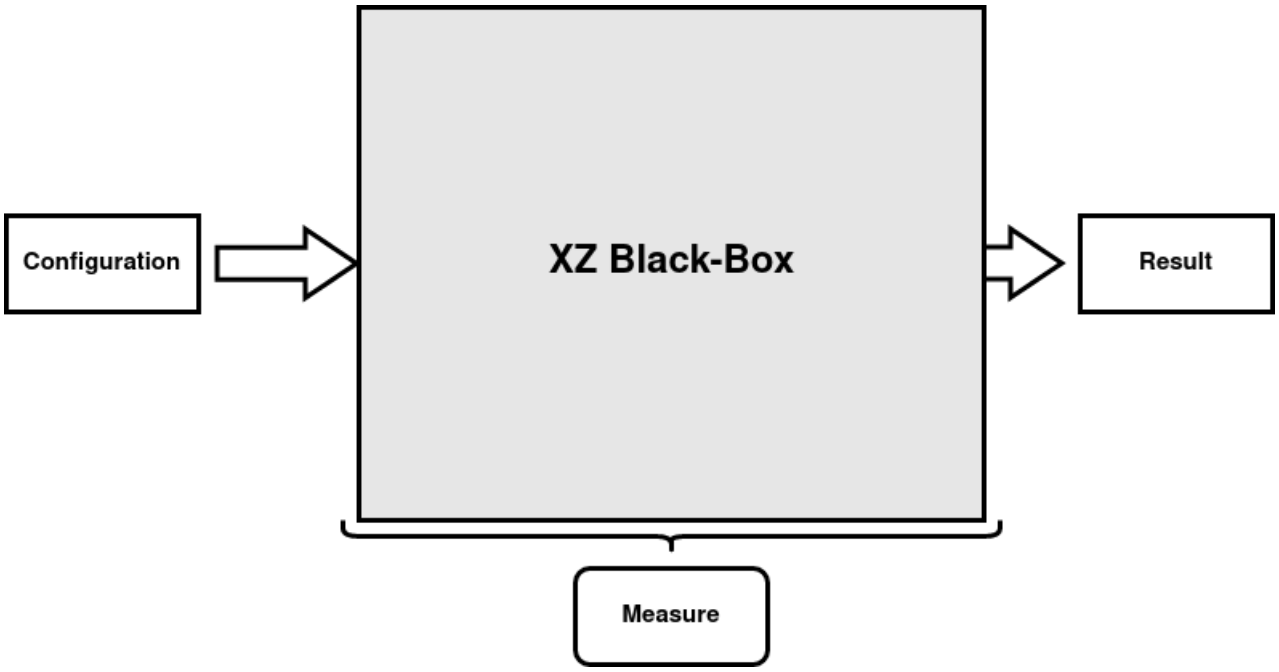

Figure 2.3: A Black-box Version of XZ

As an example for a black-box analysis we modify XZ from Figure 2.1. To perform a black-box analysis, we use XZ with the different configuration that contain the features to be measured. Now, for each configuration, we observe the non-functional properties we are interested in, such as time spent during the execution of the system. During this time, we are unaware on how XZ produces these results, since we can solely observe the effects of XZ on the machine on which it is running. After repeating this process several times, we use all the collected measurements to build a performance-influcence model representing XZ.

2.5.1 *Disadvantages of black-box model*

Although a black-box analysis is inherently simple, it faces two major challenges, combinatorial explosion and multicollinear features, both of which must be handled carefully to build an accurate performance-influence model.

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, and hence system variants, increase exponentially [3].

Suppose we have a configurable system where each feature is a binary option that you can either select or deselect. We also define that in this system each feature is completely 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 [9], for reference it is estimated that the universe contains about 10^{79} atoms, which is still less than the number of variants a system with 263 features produces, it is already impossible to use brute-force to analyze such a system, let alone the Linux kernel.

For this reason, we cannot fully explore the entire configuration space and therefore must select a subset that represent the system with a high accuracy. To achieve this state of the art black-box analyzes use sampling strategies to find a suitable subset, such as pair-wise sampling, most-enabled-disabled and random sampling.

MULTICOLLINEAR FEATURES We have already mentioned that features that do not influence each other are called independent features, but configurable systems are not composed of independent features only. If there is a dependence between more than two features, we call these features multicollinear.

The reason why multicollinearity is a problem in a black-box analysis is that we can only observe nun-functional properties during execution and are not able to correctly assign the influence of each feature on the system. One way multicollinearity is introduced into a system is by using alternative groups, since the selection of a feature in the alternative group depends on which features are deselected. [5]

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

Table 2.1: Configuration example illustrating multicollinearity in an alternative group

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 Base feature and 3 features for an alternative A, B, and C (i.e. for each configuration only one of these 3 features can be selected). This now clearly shows the dependence between these features, because in order for feature C to be selected, B and C needs to be deselected, therefore $C = 1 - B - C$ needs to be satisfied.

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

This leads to multiple performance-influcence model that are accurate with respect to the individual measurement, but make completely different statements when compared. The problem here is that since one child of the alternative must be selected, the others must be deselected. Therefore, we cannot correctly infer the influence all features has on to the system, as we can see in our example base is attributed 0, 5 and 8. These values of base or the selected feature can be set in any ratio as long as the sum of the two values is equal to the measured time.

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. [5]

Algorithm 2 Equivalence

```
1: if (c and ¬d)or(¬c and d) then
2:     c, d ← False
```

By extending the example code from Algorithm 1 with an additional condition where $C \equiv D$ holds, so that either C or D can be selected without the other, we insert the code snippet Algorithm 2 after Algorithm 2.

Now, if we select a configuration c that contains the features {{A}, {B}, {C}} we get multiple performance-influcence models:

$$\Pi_1(c) = 1 + 1 \cdot c(C) + 1 \cdot c(D) + 1 \cdot c(C) \cdot c(D)$$
$$\Pi_2(c) = 1 + 0 \cdot c(C) + 0 \cdot c(D) + 3 \cdot c(C) \cdot c(D)$$
$$\Pi_3(c) = 1 + 1 \cdot c(C) + 2 \cdot c(D) + 0 \cdot c(C) \cdot c(D)$$

In $\Pi_1(c)$ and $\Pi_2(c)$, all features are assigned different values from what we would expect when looking at Algorithm 1, while the performance-influcence modelof $\Pi_C(c)$ assigns the expected values.

2.5.2 *Selecting the configuration space*

Software products are constantly being improved, and to add more functionality, the number of optional features is also increased, which also dramatically increases the configuration space, but it has already been shown by Tianyin Xu et. al. [14] that not all features are equally important and that up to 54,1% of features are rarely set by users.

We take advantage of this knowledge to solve the combinatorial explosion problem by using our domain knowledge to extract the most important features we are interested in. This also helps us to mitigate the influence of multicollinear features, as we can ensure that the dependencies between the selected features are as low as possible when selecting the features to include in our configuration space.

2.5.3 *Collecting data*

Having decided which features are of interest to us, we can now turn to the question of to collect our data, which we can then use to create a performance-influcence model.

Since we are using a black-box analysis, our methods are quite limited. As shown in Figure 2.3, we are not able to investigate what the system does in detail and are limited to the properties that we can observe from the outside. In this case, we can measure non-functional properties such as time to completion, energy consumption, memory usage, computational resources used, and basically any value we can observe when looking at the machine the system is running on. If possible, measurements should be repeated at least 30 times to reduce measurement noise during execution [4].

2.5.4 *Creating a Performance-Influence Model using Multiple Linear Regression*

After collecting all of our measurements, we use them to build our performance-influcence modelof the system.

Various approaches are available to build the model. A popular approach is the use of neural networks, which, however, have the significant disadvantage that the model itself loses its explainability and is usually not comprehensible.

For to these reasons, we decided to use multiple linear regression. Compared to a neural network, the linear model we derived is easy for us humans to understand and interpret, since its structure is already very that of a performance-influcence model, and therefore does not require further adaptation.

We use the following formula for linear regression for matrices[6]:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... \beta_n x_n + \epsilon \tag{2.2}$$
$$Y = X\beta + \epsilon$$

$Y = Dependent\ variable$
$X = Independent\ variable$
$\beta = Regression\ coefficient$
$\epsilon = Error$

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 $\beta_k x_i x_j$.

What we are interested in are 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, which for us represents the influence of the base code, meaning the part of the code that is executed regardless of the chosen configuration.

The value of each feature in the matrix is either 1 if the feature is selected, or 0 if it is not selected. If we have numerical features with l different options, we split this 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 [7].

2.5.4.1 *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. 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. [7]

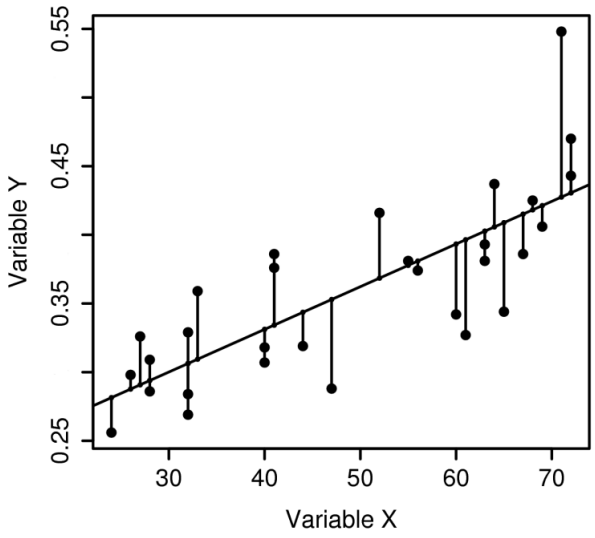

Figure 2.4: Ordinary Least Squares regression model residuals [2]

An illustration of an ordinary least squares estimator for a linear regression model can be seen 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 [7]:

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

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

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

As an example, we can refer to our code from Algorithm 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 2.2: Configuration samples of Algorithm 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 2.5.4.1 we obtain the following results:

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

All values have been rounded to 2 decimal places. We can see that all values have been assigned correctly, except for the base feature, which has been assigned 0. The reason for this is that in every configuration we use, the base feature is active, which is why ordinary least squares assigns the value of the base to the intercept and not β_1.

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

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

We are aware that if our configurations contains multicollinearity then the ordinary least squares estimator is unreliable. To check for multicollinearity we use the variance inflation factor (VIF), where a VIF factor of 0 indicates that there is no multicollinearity in our configurations and the thresholds of 5 and 10 indicate moderate and highly problematic multicollinearity respectively. [5]

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^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. [5]

2.6 WHITE-BOX MODEL

On the contrary to the black-box the white-box model requires access to the system itself. We are aware of the inner workings of the system, and given the system an input we can observe how the system uses the input, which functions are called, in what capacity a feature influence the program flow.

We use these newly available information to formulate a model that differs from our black-box approach, instead of only measuring the time the system needs to finish the process and use the input data to infer in what capacity each feature influences the system.

2.6.1 *Disadvantages of White-Box*

When using white-box model we clearly see how a feature interacts with another feature, due to that we do not need to sample our configuration space like we do in the black-box, meaning we do not face the problem of combinatorial explosion. Neither do we have to handle multicollinearity features differently, since we can see in what extend they influence each other. The example from 2.5.1 would be no problem for the white-box model since we are aware that c and d do not interact with one another and can therefore assign them the precise amount of time they spend in their code region respectively, whereas the black-box model needs to infer this information.

With the surplus of information, the white-box model faces different challenges. First and foremost, to analyze larger systems we need a robust strategy and in depth code comprehension.

2.6.1.1 *Analyzing Strategies*

Analyzing programs is a highly complex topic in itself, it is not a trivial task to use a program to run a analysis over a different system. In our case, we first need to find out which parts of the code corresponds to which feature.

To solve this problem multiple solutions have been proposed, Weber et. al. [13] uses a profiling approach, to generate performance-influence models that depict configurability on a method level, to achieve this they first used a JProflier a coarse-grained profiler to learn a performance influence model for every method that have been learned successfully. To identify the methods that are hard to learn they use a filtering techniques, afterwards using KIEKER a fine-grained profiler to learn these methods.

Velez et. al. introduced us to ConfigCrusher [11] a white-box analysis, that uses a static data-flow analysis to see how features influence variables and the control-flow of the system. In addition, ConfigCrusher uses three insights about configurable systems, from previous works, namely irrelevance, orthogonality and low interaction degree. Irrelevance, is used to identify the features that are relevant for the data-flow of the system, and by doing so reducing the amount of configurations necessary to analyze the system. Orthogonality, is used to identify features that do not influence each other, and thus can be measured together. Low interaction degree, is used to identify the relevant feature interaction, since only few features interact with another, ConfigCrusher focuses on those configurations with interacting features to reduce the amount of configuration that need to be sampled.

Siegmund et. al. introduced us to Comprex [12] which uses a dynamic taint analysis to identify how and to what degree features influence the control-flow of the given system. To reduce measurement costs they introduce two techniques, namely compression and composition. Compression is used to reduce the number of configurations necessary to analyze the system, by simultaneously analyzing regions that are independent of another, therefore they can use a single configuration to analyze different features. They take advantage of the fact that performance-influcence modelcan be build compositionally, by generating a performance-influcence model for each region separately and afterwards compose all the local performance-influcence modelinto a model for the whole system.

Both Comprex and ConfigCrusher use the techniques of compression and composition, the major difference between both analysis methods is Comprex uses a dyanmic taint analysis, whereas ConfigCrusher uses a static data-flow analysis.

After figuring out which parts of the code corresponds to what feature we still need to instrumentalize the code

2.6.2 *Indepth Code comprehension*

3 EXAMPLE CHAPTER

This chapter gives you some examples how to include graphics, create tables, or include code listings. Examples on how to cite papers from the literature can be found in Chapter 6.

3.1 ACRONYMS

This template makes advantage of the acronyms package to support acronyms. The first occurence of an acronym is replaced by its definition (e.g., Integrated Development Environment (IDE)). All other occurences are replaced by the acronym (IDE). The glossaries package also supports plural—IDEs.

3.2 GRAPHICS

In Figure 3.1, we give a small example how to insert and reference a figure.

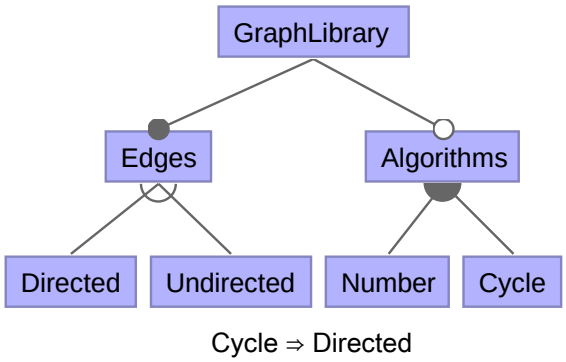

Figure 3.1: A feature model representing a graph product line

3.3 TABLES

Table 3.1 shows the result of a simple tabular environment.

Table 3.1: Mapping a feature model to a propositional formula

Group Type	Propositional Formula
And	$(P \Rightarrow C_{k_1} \wedge \ldots \wedge C_{k_m}) \wedge (C_1 \vee \ldots \vee C_n \Rightarrow P)$
Or	$P = C_1 \vee \ldots \vee C_n$
Alternative	$(P = C_1 \vee \ldots \vee C_n) \wedge \text{atmost1}(C_1, \ldots, C_n)$

3.4 CODE LISTINGS

In Listing 3.1, we give an example of a source code listing.

Listing 3.1: Java source code

```
class A extends Object {
        A() { super(); }
}
class B extends Object {
        B() { super(); }
}
class Pair extends Object {
        Object fst;
        Object snd;
        Pair(Object fst, Object snd) {
                super(); this.fst=fst; this.snd=snd;
        }
        Pair setfst(Object newfst) {
                return new Pair(newfst, this.snd);
        }
}
```

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] Andrea Arcuri and Lionel C. Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, pp. 1–10. DOI: 10.1145/1985793.1985795. URL: https://doi.org/10.1145/1985793.1985795.

[5] 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.

[6] 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.

[7] 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.

[8] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990, pp. 35–37. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.