Measuring and Predicting Non-Functional Properties of Customizable Programs

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Diplom Informatiker Norbert Siegmund

geb. am 19.08.1981 in Aschersleben

Gutachter:

Prof. Dr. Gunter Saake,
Prof. Dr. Don Batory,
Prof. Dr. Christian Lengauer

Ort und Datum des Promotionskolloquiums: Magdeburg, 27.11.2012

Siegmund, Norbert:
Measuring and Predicting Non-Functional Properties of Customizable Programs
Dissertation, Otto-von-Guericke-Universität
Magdeburg, Germany, 2012.

Customizable programs enable users to generate tailor-made program variants from a common code base. To generate a variant, users select features that satisfy their functional and non-functional requirements. Finding features that satisfy functional requirements is straightforward, whereas finding a feature selection that fulfills non-functional requirements, for example, minimizing response time or footprint, is a challenge. Knowing in advance which feature selection yields the best non-functional properties is difficult, because a direct measurement of all possible feature combinations is infeasible: 33 optional and independent features yield a configuration for each human on the planet, and 265 optional features yield more configurations than estimated atoms in the universe.

In this thesis, we present a technique to predict a variant's non-functional properties based on selected features. To this end, we determine the influence of each feature on a non-functional property by measuring two variants that differ only in a single feature. The difference of both measurements is the impact of the differing feature on a non-functional property.

We evaluate our approach with three series of experiments for the non-functional properties footprint, main-memory consumption, and performance using real-world programs from different domains (e.g., databases, operating systems) and implemented with different languages and techniques. We observe that prediction accuracy depends on whether there are feature interactions. A feature interaction exists if the simultaneous presence of two features lead to an unexpected behavior of a variant, whereas their individual presences do not.

To detect feature interactions, we developed two approaches. One requires manual analysis and domain knowledge, but achieves high accuracy. The other approach automatically detects interactions of black-box programs, but at the cost of accuracy. We evaluate their effectiveness based on the non-functional properties footprint and main-memory consumption. Depending on the approach and non-functional property, the average accuracy lies between 89 % and 99 %. We further found in our analysis that feature interactions are not evenly distributed among all features and feature interactions occur in patterns.

Base on these insights, we propose a two-step approach that automatically detects relevant feature interactions to improve prediction accuracy for black-box programs. First, we find only the features that interact. Second, we detect combinations of the identified interacting features that actually cause a feature interaction. To this end, we use three heuristics based on the insights of the previous evaluation. We evaluate this approach for performance with six real-world programs from different domains and vendors. We improve our prediction from an average accuracy of 79 % to 95 % when using all heuristics. Since a error rate of 5 % lyes with the measurement error, our predictions are nearly perfect. Hence, in this thesis, we developed a *scalable and efficient* method to *accurately* predict *different* non-functional properties of customizable *black-box* programs.

Acknowledgements

When writing a PhD thesis, the scientific work is only half of what you need to successfully complete. You need motivation, discipline, and foremost family, friends, and colleagues on whom you can rely and who support you. Without them, finishing this thesis would be impossible.

First of all, I thank my beloved wife Janet. I had the luck to meet my wife at work. Her diligence in what she was doing inspired and motivated my own work. Without Janet, who steadily pushed me, this thesis would be impossible. She motivated me when times were bad (especially after rejections) and shared happy moments. Thank you for your support and love. I thank my parents, my sister, and my whole family who always supported me in what I am doing. I could and can always count on them.

In the same way, I want to thank my friends Stephan, Jens, Ronny, Nancy, Galina, Marko, Michael, Martin, Mario, and all who I have not mentioned here. When writing a thesis, you need a clear mind and for that, you need distraction to recharge your battery. Thanks for many enjoyable days.

I want to thank my advisor Gunter Saake, who gave me the freedom that I needed to write this thesis and who was always there when funding was problematic. He supported me in visiting the University of Texas at Austin, which was a life-time experience. Furthermore, he always has good advice about my work and future directions.

I also thank Sven Apel and Christian Kästner as my advisors. I learned so much from them, especially during our long and fruitful discussions and when writing a paper together. Sven and Christian are not only advisors, but also friends and colleagues. They were never condescending, but treated me as equal.

Don Batory gave me the opportunity to visit him together with my wife. The months at Austin were awesome. I learned many things about scientific working from Don and experienced a different culture that widened my horizon. Thanks for this life-time experience. I also want to thank Christian Lengauer who always supported and advised me.

Last but not least, I want to thank my working group for a wonderful time. Especially, I want to thank Marko Rosenmüller, with which I had so many fruitful discussions that helped solving plenty scientific problems on our white board. I already miss that. Thanks to all who helped me during my thesis.

Contents

List of Figures xiii

List of Tables xvi

Abbreviations xvii

1. Introduction 1

1.1. Motivation . . . 1
1.2. Contributions . . . 6
1.3. Key Assumptions . . . 7
1.4. Research Method . . . 8
1.5. Outline . . . 9

2. Customizable Programs 13

2.1. Non-Functional Properties . . . 13
2.2. Customization Mechanisms . . . 14
2.2.1. Compile-Time Customization . . . 14
2.2.2. Load-Time Customization . . . 15
2.2.3. Run-Time Customization . . . 16
2.3. A Feature-Oriented Perspective . . . 16
2.3.1. Software Product Line Engineering . . . 17
2.3.2. Feature Models . . . 19
2.3.3. Terminology . . . 21

I. Prediction 25

3. Incorporating Non-Functional Properties in Program Derivation 27

3.1. Problem Analysis . . . 27
3.1.1. Late Requirements . . . 27
3.1.2. Unknown Influence of Features on Non-functional Properties 28
3.1.3. Black-Box Approach . . . 29
3.1.4. Measurement Effort and Accuracy . . . 29
3.1.5. Measurement of Non-Functional Properties . . . 30
3.1.6. Initial Problem Statement . . . 30
3.2. Classification of Non-Functional Properties . . . 31
3.3. Incorporating Non-Functional Properties in Program Derivation . . . 35

3.4. Related Work . . . 39
3.4.1. Models of Non-functional Properties . . . 39
3.4.2. Program-Derivation Approaches . . . 40
3.5. Summary . . . 41

4. A Feature-Centric Prediction Model **43**
4.1. Feature-Composition Model . . . 43
4.2. Prediction Model . . . 45
4.3. Relating Feature Model to Prediction Model . . . 48
4.4. Related Work . . . 52
4.5. Summary . . . 54

II. Measurement **55**

5. Overview of Measurement Strategies **57**
5.1. Direct Measurement . . . 58
5.2. Family-Based Measurement . . . 60
5.3. Feature-Wise Measurement . . . 61
5.4. Comparison of Strategies . . . 63
5.5. Related Work . . . 64
5.6. Summary . . . 64

6. Feature-Wise Measurement **67**
6.1. Feature-Wise Measurement . . . 67
6.1.1. Computing Feature Terms . . . 68
6.1.2. Algorithms and Realization . . . 75
6.2. Evaluation . . . 82
6.2.1. Experiments . . . 83
6.2.2. Footprint . . . 87
6.2.3. Main-Memory Consumption . . . 91
6.2.4. Performance . . . 95
6.2.5. Discussion: The Influence of Feature Interactions . . . 99
6.2.6. Extended Problem Statement . . . 102
6.2.7. Threats to Validity . . . 103
6.3. Related Work . . . 104
6.4. Summary . . . 106

7. Feature Interactions **109**
7.1. Introducing Feature Interactions . . . 109
7.1.1. Example: Footprint Feature Interaction. . . . 110
7.1.2. Example: Performance Feature Interaction. . . . 111
7.1.3. Order of Interactions. . . . 112
7.1.4. Causes of Feature Interactions . . . 113

7.2. Detection Approaches 114
7.3. Evaluation 115
7.3.1. Experiments 116
7.3.2. Footprint 118
7.3.3. Main-Memory Consumption 120
7.3.4. Discussion and Analysis of Measurement Approaches 124
7.3.5. Threats to Validity 132
7.4. Analysis of Feature Interactions 133
7.5. Extended Problem Statement 141
7.6. Summary 141

8. Automated Feature-Interaction Detection **143**
8.1. Introducing Deltas 143
8.2. Determining Deltas 145
8.3. Feature-Interaction Detection 149
8.3.1. Detecting Interacting Features 149
8.3.2. Identifying Feature Combinations Causing Interactions 152
8.3.3. Realization 154
8.4. Evaluation 157
8.4.1. Discussion 162
8.4.2. Threats to Validity 164
8.5. Related Work 165
8.6. Summary 165

9. Concluding Remarks and Future Work **169**
9.1. Conclusion 169
9.2. Contributions 171
9.3. Future Work 173
9.3.1. Feature Libraries 173
9.3.2. Workload-Aware Prediction of Non-Functional Properties . . 173
9.3.3. Software Measures and Non-Functional Properties 175
9.3.4. Self-Adaptive Systems 176

A. List of Case Studies **177**
A.1. Case Studies for Footprint 177
A.2. Case Studies for Main-Memory Consumption 191
A.3. Case Studies for Performance 197

Bibliography **201**

List of Figures

1.1. Research Method. 9

2.1. Domain and application engineering phases in software product line development including requirements specification [Czarnecki and Eisenecker, 2000]. 17
2.2. Program-derivation process. 20
2.3. Feature model of a DBMS product line. 21

3.1. Classes of non-functional properties. The highlighted class denotes the most challenging class for measurement. 32
3.2. Feature model of a customizable DBMS with annotated non-functional properties. 2PC: Two-phase commit protocols. 33
3.3. Big picture of incorporating non-functional properties into program derivation including measurement and prediction. 36

4.1. Mapping feature selection of abstract model to other representations. 45
4.2. Product-line model of Berkeley DB with assigned non-functional properties. Footprint represents measured binary size per feature. The up-arrow visualizes an improvement for a qualitative property. . . . 51

6.1. Process of computing all features' terms for the prediction model. . . 68
6.2. Sample customizable DBMS program. The root denotes the concept. 2PC: two phase commit protocol. 69
6.3. Feature model after approximating the influence of each feature on footprint. 76
6.4. Algorithm to compute set of configurations that have to be measured to compute all feature terms. 78
6.5. Algorithm to obtain a valid minimal configuration from a CSP solver for a given partial feature selection. 79
6.6. Building the set of equations based on measurements. 80
6.7. Experimental design with two phases. 84
6.8. Box plot (left) and violin plot (right) of ZipMe's prediction error rate for footprint. 86
6.9. Q-Q plot of prediction and estimated footprint of ZipMe. 87
6.10. Error rates in percent of all customizable programs for footprint. SNW: SensorNetwork. 90
6.11. Error-rate distribution of predicting Violet's footprint. 91

6.12. Error rates in percent of all customizable programs for main-memory consumption. 95
6.13. Error rates in percent of all customizable programs for performance. 99
6.14. Violin plots of average prediction error rates of the three non-functional properties. 101

7.1. C++ code of a customizable list with two features: *PrintList* and *PrintElement*. We show the feature model in the upper right corner. 111
7.2. Recap: Experimental setup of feature-wise measurement. 117
7.3. Experimental design with interaction-wise (IW) and pair-wise (PW) measurement. 117
7.4. Error-rate distribution of predicting footprint with interaction-wise measurement. 120
7.5. Error-rate distribution of predicting footprint with pair-wise measurement. 120
7.6. Error rates in percent of all customizable programs for main-memory consumption using pair-wise measurement. 123
7.7. Error-rate distribution of Violet for the three measurement approaches. 125
7.8. Error-rate distributions of different measurement approaches for LLVM. 125
7.9. Measured and predicted footprint in KB of Berkeley DB (compiled as static link library) using different approaches. 126
7.10. Measured and predicted footprint in KB of Linux kernel using different approaches. 128
7.11. Conceptual relation between number of measurements and error rate of predictions. 131
7.12. Recap: Threats to validity of feature-wise measurement. 132
7.13. Distribution of pair-wise interactions of SQLite's features (footprint experiment). 135
7.14. A large number of features in SQLite do not interact (footprint experiment). 136
7.15. Measured and predicted footprint in KB of Violet using different measurement approaches. 137
7.16. Partial product-line model of Violet including the mapping between features and implementation units. 138
7.17. Pattern of the occurrence of non-functional feature interactions. Orange: first-order interactions; blue: second-order interactions; yellow: third-order interaction. 140
7.18. Insights regarding the nature of feature interactions. 142

8.1. Recap: Selecting three features requires the approximation of seven terms. 144
8.2. Measuring deltas for features and interactions. We omitted the subscript "$_{Perf}$", because of space constraints. 147

8.3. Recap: Specified requirements for a prediction approach that satisfy the thesis goals. 150
8.4. Implication chains with interacting features. 156
8.5. Recap: Experimental setup for performance. 158
8.6. Error-rate distributions for feature-wise measurement and our three heuristics: pair-wise, higher-order, and hot-spot. 161
8.7. Comparing percentage of measurements with average error rates of predictions for each heuristic and customizable program. 163
8.8. Recap: Threats to validity of feature-wise measurement. 164

9.1. Initial results to predict a variant's non-functional properties depending on workload parameters. 175

A.1. Feature model of LinkedList. 178
A.2. Feature model of Prevayler. 179
A.3. Feature model of ZipMe. 180
A.4. Feature model of PKJab. 181
A.5. Feature model of SensorNetwork. 183
A.6. Feature model of Violet. 185
A.7. Boolean constraints of Violet. 186
A.8. Feature model of Berkeley DB. 187
A.9. Feature model of SQLite. 189
A.10. Feature model of selected features of the Linux kernel. 190
A.11. Feature model of Curl. 191
A.12. Feature model of LLVM. 192
A.13. Feature model of x264. 193
A.14. Feature model of Wget. 195
A.15. Feature model of Berkeley DB for main-memory consumption and performance measurements. 195
A.16. Feature model of SQLite for main-memory consumption and performance measurements. 196
A.17. Feature model of Berkeley DB Java version. 198
A.18. Feature model of Apache. 199

List of Tables

3.1. Measurement scales according to Stevens [1946] with an exemplary classification of non-functional properties. Note that metric encompasses ratio and interval scales. Std. dev.: Standard deviation, . . . 32

5.1. Relating direct measurement to goals. 60
5.2. Relating family-based measurement to goals. 61
5.3. Relating feature-wise measurement to goals. 63
5.4. Comparision between the three measurement strategies and a brute-force approach. 64

6.1. Set of configurations to determine feature terms. All measured values are in KB. 70
6.2. Description of experiment variables. Indep: independent; dep: dependent; n: number of features. 85
6.3. Overview of the customizable programs used in the evaluation of footprint prediction. 89
6.4. Overview of mean prediction error rate and measurement effort for footprint. Relative measurement effort compared to all valid configurations. Std. Dev.: Standard deviation. 90
6.5. Overview of customizable programs used to predict main-memory consumption. CC:= conditional compilation; CP: command-line parameter. 93
6.6. Overview of mean prediction error rate and measurement effort for main-memory consumption. Relative measurement effort to all valid configurations. Std. Dev.: Standard deviation. 94
6.7. Overview of sample programs used in the evaluation of performance. 97
6.8. Overview of mean prediction error rate and measurement effort for performance. Relative measurement effort compared to all valid configurations. Std. Dev.: Standard deviation. 98

7.1. Error rates in percent of footprint predictions using the approaches (Appr.): feature-wise (FW), interaction-wise (IW), pair-wise (PW), brute force (BF). Mean: mean error rate, Std: standard deviation. . 121

7.2. Error rates in percent of predicting main-memory consumption of all customizable programs using the approaches (Appr.): feature-wise (FW), pair-wise (PW), brute force (BF). Mean: arithmetic mean error rate of predictions, Std: standard deviation of predictions. 122
7.3. Evaluation of feature-wise (FW), interaction-wise (IW), and pair-wise (PW) measurement and overview of research questions and experiment results. Effort means measurement effort. Q1-Q3 refer to the research questions given in the experiment description. 129

8.1. Evaluation results for customizable programs; approaches (Appr.): feature-wise (FW), pair-wise heuristic (PW), higher-order heuristic (HO), hot-spot heuristic (HS), brute force (BF). Mean: mean error rate of predictions, Std: standard deviation of predictions. 159

Abbreviations

DBMS	Database management system
SNW	Sensor network
Q-Q	Quantile-Quantile
BF	Brute-force approach
FW	Feature-wise measurement
IW	Interaction-wise measurement
PW	Pair-wise measurement
HO	Higher-order heuristic
HS	Hot-spot-feature heuristic

1. Introduction

We develop means to measure and predict non-functional properties, such as performance and footprint, of customizable programs. In this way, we enable stakeholders to select appropriate customization options that optimize a program for their specific functional and non-functional requirements.

1.1. Motivation

A typical process of software development is to improve *non-functional properties* – such as performance, energy consumption, and footprint – of a program as it is being designed or developed. We review how non-functional properties are optimized in traditional software development and discuss limitations of conventional software development regarding applicability for different application scenarios and optimization goals. We demonstrate how customizable programs overcome these limitations, but give users the burden to find a suitable selection of customization options that optimize non-functional properties. Consider the embedded database system SQLite: If users want to find the selection of customization options with the smallest footprint and fastest performance, they are confronted with 2^{88} possibilities; a hardly solvable problem up to this time.

In traditional software development, developers capture requirements of stakeholders and implement an according program. There are two approaches to develop a program with respect to the specified requirements: specialized programs and general-purpose programs.

Specialized Programs. When developing *specialized programs*, we consider only a single application scenario. This allows us to tailor a program according to the functional requirements of this application scenario. This tailoring of functionality enables us to implement resource-efficient software, because a program contains only the actually needed functions.

Another benefit of specialized programs is that we can optimize certain non-functional properties that are important in the specific application scenario. To this end, developers use tools – such as the non-functional requirements framework [Chung et al., 1999], i* framework [Yu, 1997], and KAOS [van Lamsweerde,

2001] – to make design decisions during development that optimize certain non-functional properties of the final program.

The drawback of specialized programs is the limited functionality such that we can use specialized programs only in the intended scenario. When vendors want to support additional application scenarios, developers have to reimplement their changes on new versions of programs – possibly starting from scratch. This is especially the case when other non-functional properties become important. For instance, Windows and Linux cannot be used for real-time applications. Hence, specialized operating systems – such as eCos – or extensions to existing systems, which reimplement large portions of the kernel – such as Windows RTX, RTLinux, and Xenomai – were developed. Since different requirements regarding non-functional properties can be contradicting (e.g., optimize performance vs. maximize reliability vs. maximize availability), specialized programs have only a limited applicability.

General-Purpose Programs. The opposite to specialized programs represent *general-purpose programs*. A general purpose program covers application scenarios of a whole domain. For example, the *database management system (DBMS)* Oracle provides a wide variety of functions, such as XML storage, relational storage, transaction management, and data warehousing. In this context, Stonebraker et al. [2007] coined the expression "one-size-fits-all", meaning that a single program covers the whole database domain. However, he and others emphasized that "one-size-fits-all" is a bad idea with respect to non-functional properties.

General-purpose programs satisfy a wide range of functional requirements, but this comes at the cost of a limited applicability regarding non-functional properties. Requirements on non-functional properties imposed by the used hardware constrain the applicability of a general purpose program. For instance, installing Windows 7 on a smart phone will fail, because Windows 7 targets desktop computers with a powerful hardware, and using a full-fledged DBMS (such as Oracle) on a sensor is infeasible due to resource constraints. Hence, in spite of its usually large applicability, a general purpose program is often limited to powerful hardware.

Another drawback is the waste of resources: Functionality that is never executed remains in the program. This unnecessary functionality increases binary size, which may be important for embedded systems, and consumes main memory for data that is never accessed. Moreover, internal program functions may increase in complexity and size, because a variety of use cases have to be supported all at once. This complexity degrades performance and security, since code must be executed that has either no effect or is not of interest to the user.

Since there is often a trade-off between different non-functional properties (e.g., footprint vs. performance vs. energy consumption), general-purpose programs cannot be optimized for a specific property, because it would mean to sacrifice applicability in application scenarios that require other properties to be optimized.

Stonebraker et al. [2007] have shown that specialized programs (e.g., XML and document databases) outperform general-purpose programs by orders of magnitudes better performance, but only for certain scenarios.

Hence, vendors face the dilemma to develop either a specialized program that is highly optimized, but has only a limited scope, or a general purpose program that can be used in many scenarios, but sacrifices optimization potentials. Program customization can solve this problem.

Customizable Programs

Customization is not new. Already the first operating-system tools allowed users to specify customization options (in terms of command-line parameters) to customize the program behavior. For instance, the 1969-introduced Unix command CP, which copies files, had five options (e.g., recursive copy). These early customization options focus on functional tailoring. The growing demand for new and more sophisticated functionality resulted in more complex software and in the development of diverse customization mechanisms [Rabkin and Katz, 2011]. Today, we use compile-time (e.g., conditional compilation), load-time (e.g., configuration files and command-line parameters), and even run-time mechanisms (e.g., dynamic product lines) to tailor a program to our needs. All these techniques have in common that we select appropriate customization options – here called *features* – to obtain the desired program – called a *variant*. Features are mapped to implementation units to (de-)activate a certain function or to set some parameters.

Similar to general-purpose programs, customizable programs cover a whole domain instead of a specific application scenario. This leads to the situation that vendors have to cope with very different non-functional requirements to not lose optimization possibilities. For example, customers of a DBMS have completely different non-functional requirements when they use a particular variant in different application scenarios, such as mobile devices, parallel computers, or desktop computers. For instance, the footprint of a DBMS variant has to be minimized for an embedded system, a variant for real-time systems must provide a deterministic response time, and a DBMS variant for a mobile device requires minimized energy consumption. If developers face contradicting requirements, they often develop alternative features to provide different variants that are functionally equal, but satisfy different non-functional requirements. For instance, a customizable DBMS may provide alternative buffer-manager features: one minimizes working-memory consumption and another feature optimizes performance.

Finding an Optimal Variant

If a customizable program has many features, it is rarely clear which feature selection gives rise to which non-functional properties. That is, how should we customize an embedded database (e.g., SQLite) to yield a variant with a footprint lower than 200 KB and a response time of less than one second?

A major problem is that customizable programs often have many and partially unknown features for which users do not know how the features satisfy their requirements. Hence, users specify only a *partial* feature selection. They know which features provide functionality that is required for their application scenario, but which of the remaining features provide an additional benefit for their use case remains unclear, because the influence of a feature selection on non-functional properties could not be quantified so far.

Consider the Linux kernel to make the above problem clear, because it represents a common case of customization problem [Hubaux et al., 2012]. Linux contains about 8 000 features. For a single person, it is not possible to know what each of these features do and how these features influence other features [Hubaux et al., 2012]. The website `linux.com` lists the top five deployment mistakes for Linux. The top one mistake addresses the non-functional properties performance and user-experience:

> ''*For mission critical server systems, you need to make sure that you can handle peak loads and ensure uptime. This means doing extensive load testing before you deploy Linux servers to see whether you need heftier hardware, configuration changes, etc. For user systems, you need to make sure that there are no unpleasant surprises when the systems are put in front of real users who aren't already Linux experts.*''[1]

Users can specify only features that realize their functional requirements, e.g., by selecting hardware drivers in Linux. However, which memory strategy, file system, or compiler optimization is additionally needed when, for example, Linux should be deployed on a web server, remains unknown. Users expect that remaining variability should be bound such that they obtain an optimal and valid kernel. Whether a feature selection is valid defines a variability model (e.g., a *feature model*). But which additional features should we select to, for example, minimize binary size or optimize performance when it is used in an automotive system?

Answering this question is far from trivial. A correct answer requires a vendor to measure non-functional properties of all of these variants. This measurement process is usually costly and time-consuming, because even programs with a limited variability (i.e., only few features exist), can have millions of possible variants. With an increasing number of features, vendors face an exponential explosion of the variant space. There can be hundreds of features resulting in myriads of configurations:

[1] Source: `http://www.linux.com/news/technology-feature/security/383997-top-five-linux--deployment-mistakes` Accessed on: August 20th, 2012

33 optional and independent features yield a configuration for each human on the planet, and 265 optional features yield more configurations than there are estimated atoms in the universe. To find the feature selection with the best performance for a specific workload requires an intelligent search; brute-force is infeasible. Hence, rather than measuring each variant, we have to *predict* the optimal feature selection. This prediction, however, requires knowledge about how features and feature combinations influence non-functional properties. To the best of our knowledge, there is no method that gives us this information in the necessary detail and accuracy.

Goal of this Thesis

Our goal is to enable users to derive nearly-optimal variants of customizable programs with respect to measurable non-functional properties. Instead of measuring each variant, we *predict* a variant's non-functional properties based on user-selected features. That is, we aggregate the influence of each selected feature on a non-functional property to compute the properties of a specific variant. To this end, we measure non-functional properties *per feature*. We compile and measure two variants and approximate values per feature from the delta between these variants. That is, we predict what influence the selection of a single feature will have on non-functional properties of the final program. We show that for programs with n features, already $n + 1$ measurements can lead to acceptable predictions, which can be improved further with more measurements.

The accuracy of predictions depends on many factors. There are non-functional properties, such as performance, that are program wide: They emerge from the presence and interplay of multiple features. For example, database performance depends on whether a search index or encryption is used and how both features operate together. We show how we can determine how the combined presence of two features influences non-functional properties. Two features interact if their simultaneous presence in a configuration leads to an unexpected behavior, whereas their individual presences do not [Calder et al., 2003b, Nhlabatsi et al., 2008].

We present a prediction model that captures the influence of features and feature interactions on non-functional properties. We propose three heuristics to automate the detection of feature interactions. An important goal is that our predictions and measurements are applicable for black-box programs. That is, we develop an approach that is independent of programming language, implementation techniques, and customization mechanisms. Furthermore, our approach does not rely on any domain knowledge or implementation artifacts. Hence, we aim at treating a customizable program as a black box.

We evaluate our prediction model in several experiments with different non-functional properties, such as footprint, main-memory consumption, code metrics, and performance. We use real-world programs – such as Linux, SQLite, Berkeley DB,

and Apache – coming from different domains and vendors and implemented with varying programming languages (C, C++, Java) and customization techniques (e.g., condition compilation, command-line parameters). We show that an average accuracy of 95 % for performance is possible when interactions are known. Furtermore, we demonstrate that we can predict footprint of nine programs with an accuracy of 99.8 %, on average. With these predictions, users can find nearly-optimal variants, for example, using constraint-satisfaction problem solvers.

1.2. Contributions

We enable stakeholders to find a nearly-optimal variant for their application scenario that has an optimized performance, a minimized binary size, or reduced main-memory consumption. To this end, we propose, implement, and evaluate a novel approach that quantifies non-functional properties of features and feature interactions. We make the following contributions:

1. **Feature-wise Measurement of Non-Functional Properties.** We propose a technique that enables users to measure the influence of features on measurable non-functional properties. Although many programs were customizable for decades, there is no generally applicable approach yet that quantifies the influence of customization options on non-functional properties. Two major problems prevented solutions in this area: how to quantify the influence of individual features and how to manage the huge configuration space. We propose to use the delta (or difference) of two measured variants that differ only in a single feature to approximate this feature's influence on non-functional properties.
2. **Prediction of Non-Functional Properties.** We develop a prediction model that incorporates features, feature interactions, and non-functional properties. Although simple in structure, it achieves accurate predictions for customizable programs. Furthermore, we outline how this model can be extended to incorporate also the workload of an application.
3. **Feature-Interaction Detection.** Feature interactions are a major problem when coping with variable software, because they cause unexpected system behaviors. Such an unexpected behavior is especially a concern when predicting non-functional properties and measurement per feature is no longer possible. We provide a method to automatically detect feature interactions using three heuristics. These heuristics define configurations that must be measured to find feature interactions.
4. **Analysis of Customizable Programs and Non-functional Properties.** Our work provides measurement data of real-world programs for the first time. These data lay the foundation with respect to non-functional properties for

other research directions. Currently, researchers from University of Waterloo, University of Passau, and ETH Zurich use our measurement data and approximations to develop new methods in their respective fields. For example, researchers at the University of Waterloo work on multi-objective optimization of software product lines, for which they use our approximations of a feature's influence on non-functional properties.

5. **Black-Box Approach.** Related approaches to prediction have relied on domain knowledge. Especially, the detection of feature interactions usually requires that either the source code of a program or feature specifications are available. We rely only on a description of what features exist and in what format a program expects a feature selection, so that we can automatically produce and measure variants. Hence, we do not rely on special implementation techniques, customization options, programming languages, or application domains. The drawback of a black-box approach is a less precise prediction compared to white-box approaches that can take program internals into account. Our goal is to find a sweet spot between generality (i.e., applicable to black-box programs), measurement effort, and prediction accuracy. We show in our evaluation that we reach a sufficient accuracy in our predictions.

Our aim is to make a real-world impact with this thesis. Hence, we have looked at a real-world problem: finding a suitable configuration that meets functional and non-functional requirements. Choosing appropriate options can reduce hardware costs (e.g., due to minimizing hardware requirements), improve response times, and produce more efficient programs in terms of energy consumption. Furthermore, we selected real-world programs to evaluate our approach under realistic conditions and to provide realistic measurement data to vendors.

1.3. Key Assumptions

Our approach is not applicable to all kinds of programs and non-functional properties, because there can be significant differences in properties that may prevent their values being measured in an automated way. Hence, to clarify which conditions must be met to use our approach, we list our assumptions:

- **Measurable Non-Functional Properties (A1).** There are many kinds of non-functional properties. We divide them into automatically measurable and non-measurable properties. Non-measurable properties are (i) qualitative properties (e.g., reliability in some application scenarios) for which we cannot specify a meaningful measurement approach and (ii) properties that are too expensive to measure or require manual tasks (e.g., human experiments). For instance, user experience, security, and usefulness are non-measurable or very expensive to measure. Hence, we concentrate on measurable non-functional

properties – such as performance and binary size – for which we can automate the measurement process.

- **Automated Variant Derivation (A2).** *Variant derivation* refers to the process of generating a variant based on a feature selection (e.g., by specifying command-line parameters). Our approach requires that a program can be automatically produced given its feature selection. That is, we must be able to generate a variant in an automated manner, because we automate the whole measurement process. Many binary programs that can be executed with some kind of feature selection (e.g., command-line parameters) fulfill this requirement. However, there are customization mechanisms that require a manual preparation to produce a variant. For example, software product lines can provide features that are not yet implemented or need to be manually assembled from components (see application engineering in Section 2.3.1). These programs cannot be compiled and measured automatically, which is a prerequisite for our work.
- **Feature Order (A3).** In this work, we do not consider the ordering at which implementation units are composed. Either the ordering is implicit (e.g., in the case of annotated source code) or the order at which features are composed is explicitly given. That is, a variant has always the same absolute order of features, which is, for example, different in component-based programs.

1.4. Research Method

We follow a constructive research approach, as shown in Figure 1.1. That is, based on the initial problem analysis in Section 3.1, we develop our prediction model. We use this model to estimate non-functional properties based on a feature selection.

Our research started with an initial prediction model (Chapter 3). Next, we evaluated the prediction model in terms of its accuracy. To this end, we conducted several experiments to empirically rate the accuracy of the model for the non-functional properties footprint, main-memory consumption, and performance (Chapter 6). We analyzed the results of the evaluation to identify problems that cause prediction inaccuracies. This resulted in an extension of the problem statement. We refined the prediction model to solve the newly identified problems. Again, we evaluated the accuracy of the refined model by multiple experiments (Chapters 7 and 8). With this method, we improved our model in each step such that our predictions become more accurate and the external validity is increased (i.e., more non-functional properties and programs are evaluated).

In this thesis, we developed the above concepts in three iterations:

1. **Feature-Wise Measurement.** In the first iteration, we propose to measure the influence of a feature based on the non-functional delta of two variants

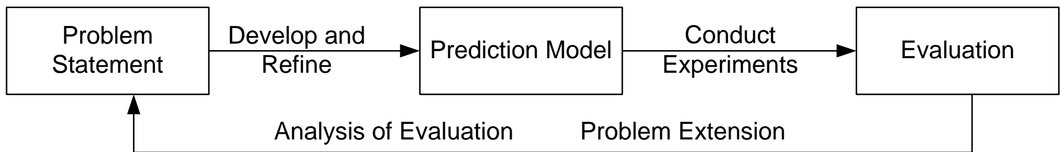


Figure 1.1.: Research Method.

that differ only in the corresponding feature. Since we do not measure features directly, one research question during the first iteration is how to keep the prediction accuracy high with only few measurements. We call this approach *feature-wise measurement* (described in Chapter 6). We conduct experiments with the properties footprint, main-memory consumption, and performance to evaluate prediction accuracy.

2. **Feature Interactions.** Based on (1), we identify that feature interactions cause significant prediction error rates. From this insight, we extend our problem statement, such that we need to determine the influence of feature interactions on non-functional properties. In the second iteration, our first attempt to incorporate feature interactions is to use domain knowledge (called *interaction-wise measurement*) and a naïve detection approach (called *pair-wise measurement*). We describe and evaluate these approaches in Chapter 7.

3. **Automated Detection of Feature Interactions.** We observed in (1) and (2) that interactions are not evenly distributed among all features such that we do not have to measure all feature combinations to find interactions. Furthermore, we identify that higher-order feature interactions (i.e., interactions between more than two features) exist, which can degrade prediction accuracy of pair-wise measurement.

   Since our main goal is to support black-box programs, we cannot rely on domain knowledge and source-code analysis. Hence, we cannot use interaction-wise measurement. Instead, we developed an *automated feature-interaction detection* approach, which uses three heuristics (presented in Chapter 8) based on the observations of our second iteration. We conduct an experiment with the non-functional property performance to evaluate feasibility and accuracy of our automated detection approach.

1.5. Outline

Chapter 2. In the next chapter, we give background about different customization techniques, which we support with our approach. This description helps to judge applicability of our approach compared to other techniques. Furthermore, we define our terminology we used to ease understanding of the remaining thesis. Finally, we

show important concepts of software product line engineering, such as implementation techniques and the feature model, which we use as a central element to choose appropriate configurations for measurement.

Chapter 3. We describe our initial problem statement, why we developed a black-box approach to increase applicability, and why scalability of measurements is crucial for highly customizable programs. We propose a classification of non-functional properties that addresses the problem of measuring very diverse non-functional properties. Based on this classification, we suggest different measurement and configuration strategies, which we outline in Chapter 4. We conclude this chapter by illustrating the big picture of incorporating non-functional properties in the configuration of a customizable program. In particular, we explain when each class of non-functional properties should be measured and configured.

Chapter 4. We present our prediction model by introducing the concept of feature and feature-interaction terms. We infer the concept of terms in a feature selection from a feature-composition model and map it to the composition of the computational effects of selected features on non-functional properties. Furthermore, we explain the relationship between feature model (as a mean to configure a program) and prediction model. In this context, we propose an extension to feature models, called product-line models, to be able to represent feature interactions, which map to feature-interaction terms in the prediction model. Furthermore, we describe how non-functional properties can be assigned to features in the product-line model. We optionally support the definition of implementation units, which help in identifying feature interactions.

Chapter 5. We explain and compare three strategies to measure and quantify the influence of features on non-functional properties. These strategies are: direct measurement, family-based measurement, and feature-wise measurement. We show that feature-wise measurement supports measurement of most non-functional properties and is implementation independent. For these reasons we present feature-wise measurement in Chapter 6.

Chapter 6. We present our main approach to quantify the influence of features on non-functional properties. We show how to derive and measure suitable variants and how to infer from these measurements the impact of a feature on a non-functional property. We evaluate this approach with three series of experiments using the non-functional properties footprint, main-memory consumption, and performance. We show that feature-wise measurement achieves a good accuracy for some non-functional properties, but when features interact, we observe high error rates.

Chapter 7. We propose two approaches to determine the influence of feature interactions on non-functional properties with the goal to improve prediction accu-

racy. The first approach (interaction-wise measurement) uses domain knowledge and source-code analysis to manually define feature interactions. The second approach (pair-wise measurement) defines an interaction between each pair of features. We evaluate both approaches and extend our problem statement with the findings.

Chapter 8. We present an approach that automatically detects feature interactions without the use of domain knowledge or code analysis. We propose a two-step technique, in which we first detect features that interact at all, and, second, we find combinations of these features using three heuristics that actually cause a feature interaction. We evaluate measurement effort and prediction accuracy of our three heuristics with six real-world programs for the non-functional property performance. We found that we can predict different non-functional properties for black-box programs with an acceptable accuracy by finding the sweet spot between measurement effort, generality of the approach, and prediction accuracy.

Chapter 8. We summarize our results and give future research directions for measuring and optimizing non-functional properties.

2. Customizable Programs

We give background about customization mechanisms to show which customizable programs we can measure and predict with our approach. Furthermore, we present our terminology, which we use throughout the thesis. We conclude this chapter by presenting feature models, which are a central element for our approach. Next, we review definitions of the term non-functional property.

2.1. Non-Functional Properties

In literature, there are controversial discussions about the terms non-functional properties, non-functional requirements, and quality attributes. There are over 25 definitions for these terms and even surveys that reflect and comment these definitions to add their own definition at the end [Glinz, 2007, Chung and do Prado Leite, 2009]. To give an impression, we present some of these definitions:

- *"Describe the nonbehavioral aspects of a system, capturing the properties and constraints under which a system must operate."* [Anton, 1997]
- *"The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability."* [Davis, 1993]
- *"A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior."* [Wiegers, 2003]
- *"A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property."* [Robertson and Robertson, 1999]

However, we identify a consensus that a non-functional property does not relate to the functionality of a variant, but to a quality or behavioral attribute. Hence, not to give a 26th definition of non-functional properties, we use this consensus to express non-functional properties. In our work, we measured several kinds of properties, such as *footprint* (as binary size of a program), *main-memory consumption* (as peak memory consumption of a program), and *performance* (as response or execution time of a program). We further give a classification of non-functional properties in Section 3.2 and describe related models in Section 3.4. Furthermore, we consider a non-functional requirement as a qualitative or quantitative requirement on a specific

non-functional property, such that a requirement usually constrains the number of valid programs.

2.2. Customization Mechanisms

Users can customize programs with many different techniques and at different times. For example, they can specify command-line options at program start to dynamically tailor a program to their current needs. Moreover, they can also generate and construct a tailor-made program specifying which implementation units have to be compiled. In the following, we provide an overview of different customization mechanisms, because they affect how non-functional properties emerge and how we have to measure them. We divide them into compile-time, load-time, and run-time customization times, because these different customization times allow us to measure different non-functional properties, which affect the design of a general measurement approach.

2.2.1. Compile-Time Customization

Compile-time customization refers to techniques with which stakeholders tailor programs through compilation. That is, a stakeholder specifies a configuration to define which code must be compiled. For example, a preprocessor removes code from a source-code file that is not annotated (e.g., using `#ifdef` statements) with user-defined flags. Hence, code remains in a file only when it was annotated with defined preprocessor flags. Also, code that was not annotated remains in a file.

Other techniques use composition. That is, a developer specifies a feature selection, which maps to modules that have to be assembled and compiled. Prominent techniques in this area are aspect-oriented [Kiczales et al., 1997] and feature-oriented programming [Prehofer, 1997, Batory et al., 2004], as well as program generators using components. Depending on the concrete implementation technique, measuring non-functional properties can be challenging. For example, we can easily measure performance or footprint of components, because we can compile and execute them individually. But this means that we measure only a subset of the behavior, because we cannot measure how components work together.

In contrast to components, feature modules, as used in feature-oriented programming, have no explicit interface and may not even provide a cohesive implementation. Instead, feature modules are often entangled with other features, such that the implementation of a single feature can cut across several classes introduced by other features. Measuring execution time in such an entangled scenario is complicated, because the program flow may frequently visit several features, so that an isolated

measurement is not possible. Hence, a general applicable measurement approach has to consider also such techniques.

Compile-time customization is the only technique that influences the non-functional property binary size. That is, when stakeholders want to minimize binary size of a program, they have to apply compile-time customization. Moreover, when variants are sold as source code (e.g., as libraries), code metrics may become important to judge (at some degree) the code quality. Improving code metrics via customization is only possible for compile-time customization. That is, the source code of a variant is assembled, but not yet compiled.

2.2.2. Load-Time Customization

When customizing a program at load time, we tailor the behavior of a program dynamically with configuration options [Rabkin and Katz, 2011]. Variability is encoded using condition statements (e.g., `if` statements), which change the program flow depending on the given configuration. This kind of customization affects only non-functional properties that emerge at runtime. For example, the binary size of a program will not change no matter what the configuration looks like, but the main-memory consumption and performance can change substantially. Hence, when customizing a program at load-time, we can only influence properties that emerge from the program's execution behavior.

A traditional method to customize a program at load-time is the use of command-line parameters or configuration files. A program is called with a specific set of flags. These flags determine the executed functionality (e.g., /r for recursive search), quality aspects (e.g., /c=1 for a high compression rate in the data-compression program RAR[1]), and the workload (e.g., which folders to analyze). Usually, each of these options influences non-functional properties. For example, selecting a high compression rate increases processing time.

With an increasing complexity of programs and the desire for high customization, command-line options have reached their limits in terms of manual specification. Instead of specifying parameters at each program start, we can use configuration files to customize programs with hundreds of parameters, such as a web server. A configuration file (e.g., an ini-file in Windows) typically stores a set of key-value parameters. A program reads at start the configuration file and stores the values internally. When the program flow reaches a point in which a configuration option defines the following execution path, the program accesses its repository and evaluates the corresponding parameter. There are a number of ways to customize a program at load-time. For example, we can define key-value parameters within the Windows registry and read these values in our program.

[1] Source: http://www.winrar.de [Accessed: December 6th, 2012]

An important prerequisite for our approach is that all customization techniques can be used programmatically (A2). That is, we can *generate* a certain configuration and start the program accordingly. If there is user interaction involved (e.g., a customization option is selected only in a user-interface dialog), we cannot support this customization technique with our approach, because we aim at measuring non-functional properties in an automated manner.

2.2.3. Run-Time Customization

In contrast to load-time customization, we do not customize at program start, but when the program is running. Hence, run-time customization can trigger an adaptation of the program at run-time.

Currently, we do not support measurement and prediction of non-functional properties of programs that are customized at run-time. However, this is only a technical issue, as we experimentally showed that determining non-functional properties is possible at run-time [Rosenmüller et al., 2011b]. Adaptive systems provide variability to customize programs at run-time. Such approaches are often based on components [Oreizy et al., 1999]. Developers define adaptation rules, which trigger the actual adaptation process [Floch et al., 2006, Garlan et al., 2004]. For example, they change the currently executed component architecture by connecting new components.

A novel customization concept represents dynamic software product lines. Run-time customization is performed via a reconfiguration of a product line at run-time [Alves et al., 2009, Hallsteinsen et al., 2008, Rosenmüller et al., 2011b]. That is, the same concepts that we rely on for compile-time customization are lifted to run-time configuration. For example, feature models are used to reconfigure these programs, which allows for consistency checks of run-time adaptations [Cetina et al., 2009]. Hence, we argue that our approach is applicable also for run-time customization.

2.3. A Feature-Oriented Perspective

In this thesis, we address different types of customization mechanisms. Although each mechanism has a different notation about how to specify customization options, we consider all of them as features. The process of selecting features is a central topic of software product line engineering and therein well understood. This is why we use product-line terminology in our work and this is why we describe concepts of software product line engineering in the following to ease comprehending the remaining thesis. We continue with a description of feature models that we use to structure features and derive valid variants. We conclude with a description of the terminology we use

throughout the thesis such that the reader has a glossary about the meaning of the various terms we use.

2.3.1. Software Product Line Engineering

The manufacturing industry use product lines since the beginning of the 20th century. Besides cost-reduction and a decreased time-to-market, a product line provides another major benefit: *customization*. Instead of producing a single product for all customers, customers specify their desired features so that a tailor-made product is manufactured. Although customized, these products are all related; build up from a common set of assets. Software engineers transferred this concept to software development. A *software product line* is a set of related programs sharing a common code base [Clements and Northrop, 2002]. We distinguish programs of a product line in terms of *features*, in which a feature is a stakeholder-visible characteristic [Kang et al., 1990].

In software product line engineering, we differentiate between *domain engineering* and *application engineering* [Czarnecki and Eisenecker, 2000, Pohl et al., 2005] as illustrated in Figure 2.1. The goal of the domain-engineering phase is to capture and model necessary features based on an analysis of the domain and to implement these common features. During application engineering, users specify concrete requirements that are specific for their application scenario. Based on these requirements, they select features to produce a tailor-made variant. In the following, we describe each phase in detail.

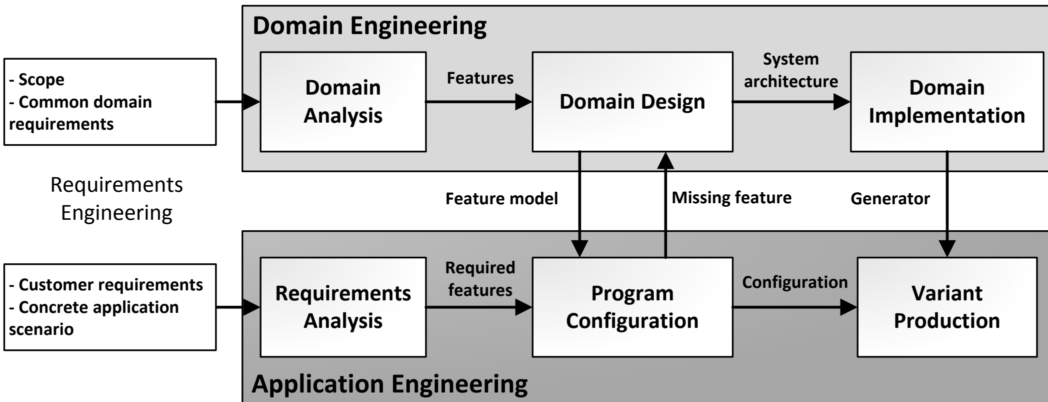


Figure 2.1.: Domain and application engineering phases in software product line development including requirements specification [Czarnecki and Eisenecker, 2000].

Domain Engineering

Domain engineering splits into three tasks: *analysis*, *design*, and *implementation* [Czarnecki and Eisenecker, 2000].

- **Domain Analysis.** A domain engineer analyzes functional and non-functional requirements that are important for an entire domain (i.e., not necessarily for a single application scenario). *Feature-oriented domain analysis* is a technique to capture commonalities and differences of requirements in terms of *features* [Kang et al., 1990]. Common requirements lead to features that are central for reuse in different programs of a product line, whereas different requirements lead to features that distinguish programs. Feature-oriented domain analysis describes a way to structure features of a domain in a feature model, which we describe in Section 2.3.2.
- **Domain Design.** Product-line architecture has to reflect the diverse requirements that are identified during domain analysis. It must support variability such that different programs can be generated. The aim of this task is to specify the used techniques to realize variability (e.g., with preprocessor or components) and to specify in which form the identified features can be implemented. For example, it may be suitable to implement several features within a single component to reduce complexity of implementation.
- **Implementation.** Finally, based on the architecture, developers implement all common features. Depending on the vendor, not all features may be implemented at this phase. For instance, a feature that is required only seldom may be implemented when the first variant requires it. Nevertheless, typically almost the whole implementation work is done in the domain-engineering phase, affecting the point at which we can quantify the influence of features on non-functional properties.

Domain engineering is in contrast to conventional software development, in which concrete requirements are specified *before* development. For example, a DBMS product line can provide features for in-memory and persistent-storage support. Although both features have contradicting goals (i.e., performance vs. reliability), they both are useful for specific scenarios (e.g., an in-memory variant for a web browser and a persistent variant for an e-mail client). Developers implement such alternative features to satisfy different and even incompatible goals.

Application Engineering

For each variant, application engineers analyze requirements of the concrete application scenario and map them to a selection of features of the product line. This set

of selected features, called a *configuration*, satisfies all requirements.[2] By mapping features to implementation units, a generator produces a program based on a configuration. This process is called *program derivation* and described in detail next. Depending on the used implementation technique, a generator may be a compiler including a preprocessor or a composition-based tool, such as FeatureHouse [Apel et al., 2012] and AHEAD [Batory et al., 2004].

Program Derivation. Program derivation is the process of obtaining a variant from a customizable program. This process contains of several steps, as we illustrate in Figure 2.2. A stakeholder defines functional and non-functional requirements on a program. Functional requirements can often be mapped directly to a set of features. This results in a *partial configuration* – also called *partial feature selection*. Usually, only a subset of features is required, because customizable programs cover a whole domain rather than a specific application scenario. Nevertheless, the remaining variability has to be bound. This leads to the situation that stakeholders have to decide which additional features to select and which not. Furthermore, there can be alternative features that provide the same functionality, but optimize different non-functional properties. This decision, however, is not trivial and has a crucial influence on non-functional properties. So, which remaining features should be selected to, for example, minimize binary size or optimize performance, remains unclear. This thesis focuses on this step to give feedback to the user which features influence non-functional properties to what extent in order to derive an optimized variant.

After binding the remaining variability, the configuration is mapped to implementation artifacts (e.g., components, command-line parameters, or preprocessor flags). With this mapping, we produce a variant. There are a large number of tools supporting program derivation [Antkiewicz and Czarnecki, 2004, Czarnecki et al., 2004, Batory, 2005]. A prominent example is the Linux *kbuild* tool.[3] Kbuild encompasses a Linux variability model, a configuration language (kconfig), and a production tool. It resolves dependencies between features and produces a valid kernel based on a partial configuration. However, it does not support means to specify non-functional requirements.

2.3.2. Feature Models

A feature model specifies all valid configurations of a customizable program. It was introduced by Kang and others in 1990 as a way to model the result of a domain analysis [Kang et al., 1990]. Since its publication, many researchers proposed extensions to feature models (see Schobbens et al. [2006] and Benavides et al. [2010]

[2]If requirements cannot be satisfied (e.g., because functionality is missing or non-functional requirements cannot be fulfilled), new features or alternative implementations have to be developed.
[3]http://kernel.org/doc/Documentation/kbuild/

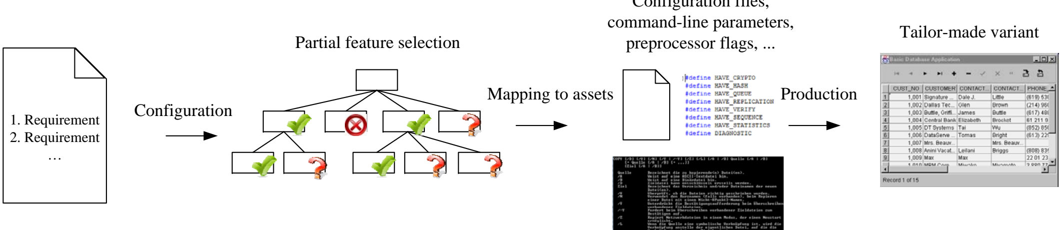


Figure 2.2.: Program-derivation process.

for a survey), such as cardinality-based feature models [Czarnecki et al., 2004]. Next to conceptual extensions, there are various approaches that specify alternative representations of feature models, such as graphically (e.g., feature diagrams), textual [Rosenmüller et al., 2011c], or as propositional formulas [Batory, 2005].

A feature diagram has a hierarchical structure beginning with a root node, which represents the domain concept. In Figure 2.2, we visualize the feature model of a customizable DBMS using a feature diagram. We use a DBMS as a running example. A feature model defines four basic relationships between features (see Figure 2.3):

- **Mandatory.** A mandatory feature must always be selected when its parent feature is selected. If its parent feature is the root node, then all configurations must contain this feature. Often, core functionality is modeled as a mandatory feature. For example, feature *base* in Figure 2.2 is mandatory, because it implements basic storage functionality that is common in all variants.
- **Optional.** An optional feature represents a variability point in a feature model. We can decide whether to select this feature. For instance, we can choose between a DBMS variant with Transaction support (by selecting feature Transaction in Figure 2.2) or without. Optional features double the number of configurations, which leads to an exponential increase.
- **Alternative Group.** Alternative features cannot occur in the same configuration. We have to select exactly one (no more and no less) feature from a group of alternative features. In our sample feature diagram, features *Btree* and *Hash* are alternative features.
- **Or Group.** In contrast to an alternative group, we can select more than one feature in an or group. Still, we have to select at least one feature. When selecting feature *Transaction*, we can select either one of the features *Logging* and *2PC* or both. In cardinality-based feature models, we can specify a range to define a minimal and maximal boundary of selected features within a group.

In addition, feature models support the definition of cross-tree constraints (e.g., feature *InMemory* requires feature *RSA*) or, more general, the definition of arbitrary propositional formulas, which we support with our approach.

Using propositional formulas has the advantage that we can encode a whole feature model into a single formula and apply different reasoning techniques [Batory, 2005, Benavides et al., 2005]. For example, we can use a satisfiability solver to verify whether a feature selection is valid. We can also enumerate all valid configurations or detect dead features, for which their selection always results in an invalid configuration.

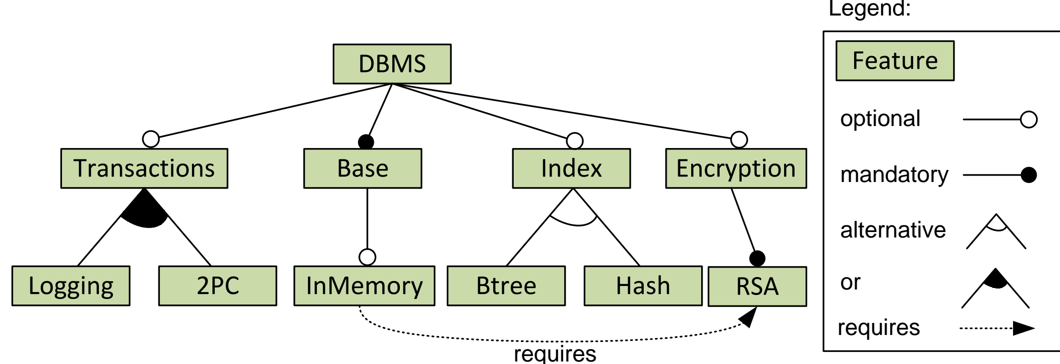


Figure 2.3.: Feature model of a DBMS product line.

2.3.3. Terminology

In the following, we present the terminology, derived from software product line engineering, to ease understanding of the remaining thesis. Although some terms presented below are introduced and defined later, we list all of them in one spot to provide a glossary for the reader.

Feature (aka. Customization Option). In literature, there are different definitions regarding the term *feature*. Czarnecki and others define a feature as [Czarnecki and Eisenecker, 2000]: "*a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept.*", whereas Kang and others defines a feature as a *user-visible* characteristic of a system [Kang et al., 1990]. Batory and others focus on functionality and define a feature as an increment in functionality of a program [Batory et al., 2004].

In this thesis, a feature is a *configuration option* provided by a customizable program. We do not distinguish between different kinds of customization mechanisms. That is, a feature may include or remove code during compilation, (de-)activate a function at load-time, and change the behavior of a program at run-time. When a user can customize a program with an option, then we consider it as a feature and we determine its influence on non-functional properties.

Feature Interaction. The term *feature interaction* has different meanings depending on the context. In a broad scope, two features interact if their simultaneous presence in a configuration leads to an unexpected behavior, whereas their individual presences do not [Calder et al., 2003b, Nhlabatsi et al., 2008]. But what does unexpected behavior mean for non-functional properties? It means that we predict non-functional properties differently than what we actually measure, but only for a certain feature combination.

To clarify the meaning of a feature interaction, consider the following example for performance: When determining performance of feature *Compression*, which compresses data, we measure a value of 20 s to pass a benchmark. Furthermore, we measure performance of feature *Encryption*, which encrypts data, and observe 40 s to pass the same benchmark. Now, we expect that when selecting both features in combination, we measure a performance of 60 s, because we first compress data (taking 20 s) and afterwards encrypt this data (taking 40 s). However, when we actually execute the benchmark, we measure only 50 s rather than the expected 60 s. The reason for this difference is a feature interaction between features *Encryption* and *Compression*, because feature *Encryption* encrypts already compressed data, which has a significant smaller size and therefore requires less time. We introduce feature interactions in full detail in Chapter 7.

Throughout this paper, we use # as the interaction symbol (described in Section 4.2). That is, when features a and b interact, we denote the interaction between both features with $a\#b$.

Interacting Features. *Interacting features* are features that participate in a feature interaction. That is, features a and b are interaction features if the combination of both features lead to unexpected change of a variant's non-functional properties (i.e., there is an observable feature interaction $a\#b$). We explain how to detect interacting features in Chapter 8.

Feature Model. A *feature model* describes all valid configuration of a customizable program [Kang et al., 1990]. It captures features and relationships among them. We described feature models in Section 2.3.2 in detail.

Configuration (aka. Feature Selection). A *configuration* is a valid feature selection according to the constraints and relationships specified in a feature model. To derive a valid feature selection, we use feature models. We denote configurations by specifying a set of selected features. For example, to denote a configuration with features a and b, we write: $C = \{a, b\}$.

Partial Configuration (aka. Partial Feature Selection). A partial configuration is a feature selection, in which not the complete variability of a customizable program is bound. That is, there are some customization decisions left. Since not all features are selected or deselected, a partial configuration may be invalid according to the constraints in a feature model.

Variant (aka. Product). A *variant* is the final and fully customized program that a stakeholder actually uses. Hence, it is a concrete instance of a customizable program according to a given configuration. That is, a configuration maps to a specific variant and vice versa.

Configuration Space. The *configuration space* enumerates all valid configurations of a customizable program. Hence, it is the search space in which we want to find a near-optimal variant.

Variant Production. *Variant production* refers to the process of generating a variant. How this production is realized depends on the used customization and implementation techniques. For example, variant production can be compiling a program with a specific set of preprocessor flags. It also can be the generation of command-line parameters or configuration files.

Program Derivation (aka. Product Derivation). *Program derivation* describes the complete process of obtaining a variant. That means, it includes the selection of features to yield a valid configuration and the subsequent variant production process. We described this process in Section 2.3.1 in detail.

Near-Optimal Solution. A *near-optimal solution (or variant)* is a configuration (or variant) that is close to the optimal configuration (or variant) for a given non-functional requirement. Although we aim at finding the best configuration for a certain non-functional requirement (e.g., to produce the fastest variant), we cannot give a guarantee, because our approach uses heuristics to decide which features and feature interactions to measure and we can only approximate the influence of features on non-functional properties. Hence, we cannot claim to find always the best configuration. However, we usually find a configuration that is nearly-optimal, as we will show in our evaluations.

Initial Feature Set. An *initial feature set* is a set of features that build a valid configuration and has a minimal number of features. We use this set as a base for further measurements to determine non-functional properties of features. We introduce the initial feature set in Chapter 6.

Implementation Unit. An implementation unit is a piece of code that realizes a feature's functionality. For example, a component may be an implementation unit for a feature. We do not limit implementation units to have a one-to-one mapping to features. That is, a single implementation unit may realize multiple features and multiple features may map to the same implementation unit. We describe this concept in Section 4.3.

Variability. *Variability* describes the quantity of customization possibilities provided by a program. That is, a program with high variability allows us to derive many variants, whereas limited variability reduces the number of producable variants.

Part I.

Prediction

3. Incorporating Non-Functional Properties in Program Derivation

This chapter shares material with the following papers:

- "Measurement and Optimization of Non-functional Properties" in *APSEC'08* [Siegmund et al., 2008b],
- "SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines" in *Software Quality Journal* [Siegmund et al., 2012c], and
- "Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties" in *VaMoS'08* [Siegmund et al., 2008a].

We present the big picture of deriving a variant optimized for specific non-functional properties. To this end, we propose different classes of non-functional properties that determine when to use different measurement strategies, which we describe in the next chapter. Finally, we describe the overall process of deriving optimized variants to provide an orientation at which phase our approach should be used. We begin with a problem analysis, in which we outline challenges to incorporate non-functional properties in program derivation.

3.1. Problem Analysis

In the following, we present several problems that arise when deriving near-optimal variants: *late requirements*, *unknown influence of features on non-functional properties*, *black-box approach*, *complexity and effort*, and *diversity of non-functional properties*. We present these problems to clarify why we cannot use existing approaches and why we have to develop new measurement strategies to quantify the influence of features on non-functional properties.

3.1.1. Late Requirements

In traditional software engineering, domain engineers specify requirements on software before development. Functional requirements are usually easy to describe and implement, because there is often a straight-forward way to verify whether the functional requirement is realized. However, when it comes to non-functional requirements, it is often not clear how to implement a complex program consisting of multiple components with response times of less than one second. Developers have

to make design decisions to satisfy non-functional requirements (e.g., implementing an authentication system to improve security). Since there are trade-offs between different non-functional properties, developers need a model to structure and reason about design decisions.

Prominent examples of models to guide developers during software development with non-functional requirements are the non-functional requirement framework [Chung et al., 1999], the i* framework [Yu, 1997], and the goal model used in KAOS [van Lamsweerde, 2001]. Unfortunately, we cannot use such models when it comes to customizable programs.

To realize customizable programs, developers face the problem that requirements of concrete customers are specified *after* development. That is, vendors have to consider a spectrum of non-functional properties during development and may implement several alternative algorithms. This prohibits using existing models. A major reason for these late requirements is that a customizable program targets a whole domain, instead of a single application scenario. As an example, consider the Apache web server, which we can use for different non-functional requirements. We can host personal web pages with characteristics, such as low access rates and low security requirements, and large commercial portals (e.g., YouTube.com) with completely other characteristics, such as performance, reliability, and security. Heterogeneous and sometimes contradicting requirements need a customizable code base rather than a single tailored one.

The same problems hold for other domains. For example, the main task of database and web-server administrators is to find the program configuration that is optimal to non-functional requirements. Unfortunately, we often cannot rely on such expert knowledge or other domain experience. The only resource of information is to measure the non-functional properties of the program itself. Hence, our approach targets programs after development. We do not want to change a program (e.g., the code base) to optimize certain non-functional properties, but to find a near-optimal configuration.

3.1.2. Unknown Influence of Features on Non-functional Properties

To incorporate non-functional properties in program derivation, we need to know to what extent a feature influences a non-functional property. That is, we need a concrete value per feature to enable users to find a suitable configuration.

This problem of how to quantify the influence of features on non-functional properties is fundamental in practice. Let us have a look at a representative case study: *SQLite* is a customizable DBMS deployed on over 500 million systems.[1] Although it targets embedded systems and thus has already a small footprint, its developers pro-

[1]Source: http://sqlite.org [Accessed: August 28th 2012]

vide further customization options to reduce the binary size of the compiled DBMS. However, they can neither provide values to which degree a deactivated feature reduces binary size, nor what influence a deactivation has on other non-functional properties, such as performance. The website states only: “the library size can be less than 300KiB, depending on compiler optimization settings” and “If optional features are omitted, the size of the SQLite library can be reduced below 180KiB.” For many application scenarios, users need more exact information than “less than” or “can be reduced below”, because they would need to generate and measure variants as long as they find a variant that meets their requirements and even if they find a variant, they do not know whether this is a near-optimal variant for their use case or variant that sacrifices large optimization potentials. Determining the influence of a feature on non-functional properties is a major problem that we aim to solve with this thesis.

3.1.3. Black-Box Approach

Instead of measuring non-functional properties, we can develop a prediction model that captures the program behavior. These models are called white-box approaches, because they require knowledge about the program internals. White-box approaches are, however, often not applicable. From a vendor’s point of view, they require domain knowledge, which may not be available, because it got lost during development. Furthermore, creating such models is a time-consuming and complex task, which increases costs. Developers have to be familiar with concrete prediction approaches, such as Markov models [Happe, 2005] and petri nets [Kounev and Dutz, 2009].

From a customer’s point of view, developing a white-box model requires not only domain knowledge, but also the source code of the respective program, which may not be available. Furthermore, constructing a model as a user is risky, because users may invest a lot of time and effort only to identify that no variant satisfies their requirements. Hence, to maximize the applicability of a prediction approach, we have to develop a *black-box approach* that does not require availability of source code and domain knowledge, but provides a sufficient accurate prediction of non-functional properties.

3.1.4. Measurement Effort and Accuracy

The goal of a customizable program is to support many application scenarios by providing a large number of customization options. The drawback of an increased variability is the huge configuration space. Each optional and independent feature doubles the number of variants; 265 optional and independent features yield more configurations than there are estimated atoms in the universe.

SQLite, for instance, consists of 85 compiler flags (i.e., features) to generate different variants.[2] When all features are optional and can be arbitrarily configured, there are 2^{85} different variants. To measure footprint of a single variant, we have to compile the corresponding code, which takes approximately five minutes. Measuring all 2^{85} variants would take approximately $3.6 * 10^{20}$ years, a billion times longer than our universe exists. Obviously, a customer cannot find the optimal variant with a brute-force approach and no prediction approach can guarantee to find *the optimal* variant. Scalability is henceforth a crucial factor of a practical solution and prediction can only provide a near-optimal solution.

3.1.5. Measurement of Non-Functional Properties

Since we target black-box programs, we must *measure* non-functional properties. Hence, we need to determine how to measure an individual property. Not all non-functional properties can be measured and an even smaller subset can be measured with the same technique. For example, properties based on code metrics require a measurement tool that reads the source code of programs and outputs the analyzed metrics *per feature*. Measuring performance requires a running variant with a certain workload, for example, a benchmark; for binary size, we need to compile a program, measure the binary files (e.g., executable files or class files), and aggregate the measurements. Other non-functional properties cannot automatically be measured at all. For example, measuring usability, user experience, and security may not be possible in an automated manner, because we cannot define a meaningful metric that quantitatively describes a program or we cannot exclude user interaction from the measurement (e.g., human experiments to measure user experience). Hence, we need to differentiate between different types of non-functional properties, so that we can define for which types which measurement and configuration techniques are suitable. Moreover, we can set the focus of our approach to only those properties that are measurable in an automated manner.

3.1.6. Initial Problem Statement

We summarize the above problems in our initial problem statement and subsequently infer requirements for a prediction approach to overcome these problems.

Of course, many application scenarios require precise estimates about the non-functional properties of a variant to be able to determine which one is the best (R1). Unfortunately, we cannot use traditional techniques to optimize programs for non-functional properties, because explicit requirements are usually not known before development and even contradicting requirements must be supported by a

[2] We count all compiler flags given on SQLite's website as well as additional flags that allows us to define important static parameters, such as different cache and page sizes.

customizable program. Also, we cannot develop a white-box prediction model, because these models require domain knowledge and availability of source code, which limits applicability – however, our main goal is to support any customizable program (R2). A possible solution is the direct measurement of a variant to identify whether it satisfies all requirements. Unfortunately, the huge number of possible variants makes a direct measurement not feasible. Hence, we can measure only a small number of variants to quantify the influence of individual features on non-functional properties to enable the accurate predictions (R3).

Based on our initial problem statement, we define the following three requirements for a prediction approach that satisfies our thesis goals:

- **(R1)** Accurate predictions of non-functional properties.
- **(R2)** Black-box approach.
- **(R3)** Determine the influence of individual features on non-functional properties.

Our solution to the previously described problems is to determine the influence of each feature on non-functional properties with only few measurements using heuristics and, based on this, predict the properties of a variant.

3.2. Classification of Non-Functional Properties

Based on the problem analysis, we discovered that we need different means to treat non-functional properties during measurement and program derivation. The classification proposed in the section is driven by the need to determine which non-functional properties can be measured in an automated manner and which non-functional properties allow us to quantify the influence of features on a metric scale to enable the computation of an optimal configuration. In Figure 3.1, we illustrate our three different classes: *non-measurable properties*, *measurable properties per feature*, and *measurable properties per variant*. In our context, measurable means *practically* measurable. That is, although we can measure a non-functional property, such as user friendliness, we classify it as a non-measurable property, because we neither can automate the measurement process nor define a meaningful metric that quantifies the influence of features on a metric scale.

The theory of measurement describes different levels (nominal, ordinal, and metric) of how measured values can be interpreted [Stevens, 1946]. That is, the theory defines valid operations and valid statistical tests for each scale. This is important in our context, since we want to compute an optimized variant using numbers on a metric scale. In Table 3.1, we present an overview of Stevens' measurement scales including exemplary properties that belong to different scales. Note that the classification of Table 3.1 may differ depending on the application scenario and customiz-

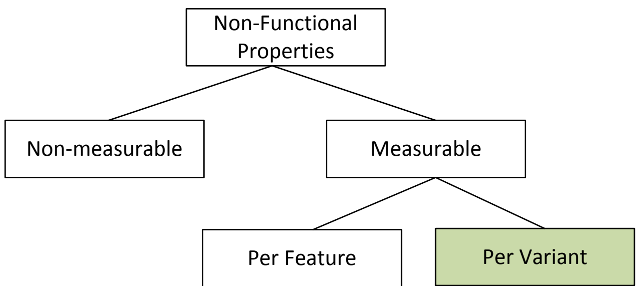


Figure 3.1.: Classes of non-functional properties. The highlighted class denotes the most challenging class for measurement.

able program. For example, we cannot predict security of a configuration, because we cannot quantify the influence of a feature on a metric scale. We can only rank different features according to their qualitative influence on security.

Scale	Operations	Statistics	Non-Functional Properties
Ordinal	equality (=), order (<)	median, percentile, mode, chi-squared	availability, reliability, security, user-friendliness, trustability, usability, integrity, completeness, user-experience, adaptability, interoperability, modularity
Metric	addition (+), multiplication (∗)	mean, std. dev., variation coefficient	footprint, performance, energy consumption, maintainability, accuracy/resolution of data, response time, resource behavior, bandwidth

Table 3.1.: Measurement scales according to Stevens [1946] with an exemplary classification of non-functional properties. Note that metric encompasses ratio and interval scales. Std. dev.: Standard deviation,

In the following, we explain each class in detail and use the feature model of a customizable DBMS depicted in Figure 3.2 as an example.

Non-Measurable Properties. There are non-functional properties that can be described only qualitatively using an ordinal or nominal scale (i.e., there is no metric from which we can retrieve quantifiable measures). For example, we can define that features *Transactions*, *Logging*, and *2PC* improve reliability of a DBMS, because logging data and transactions allow us to recover data after a system crash (see Figure 3.2). We can assign such qualitative statement to features (i.e., 'feature *Logging* improves reliability'). Since ranking is a valid operation for values on an ordinal scale, a domain expert can rate features according to their influence on a non-functional

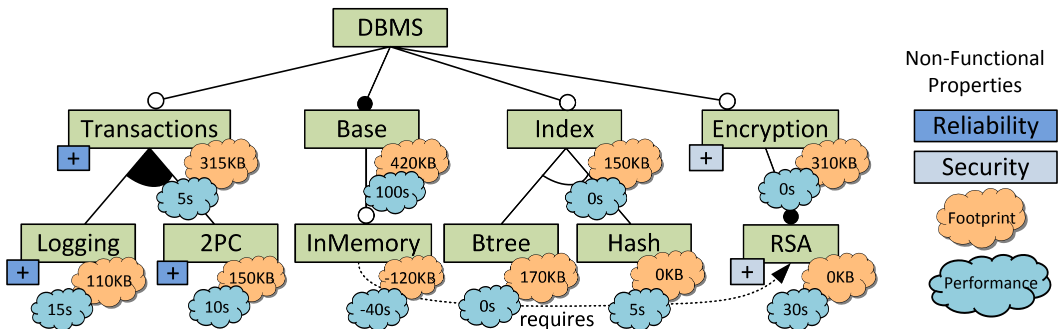


Figure 3.2.: Feature model of a customizable DBMS with annotated non-functional properties. 2PC: Two-phase commit protocols.

property. For example, we can even assign numbers to features such that a feature with high number improves a certain non-functional property more than a feature with a low number.

We consider also non-functional properties as non-measurable if there is no method that allows us to automate measuring the property. Since we cannot programmatically assign values to features, a stakeholder has to manually define the corresponding values to features. As an example, consider user friendliness. It needs large and costly user studies to quantify the influence of different features on user friendliness. The evaluation of such a study is usually done manually. Hence, a property, such as user friendliness, is out of scope, because we want to fully automate the measurement of non-functional properties. Since for qualitative properties, too, there is no measurement involved, we do not consider this class in the remaining thesis, but can support their configuration during program derivation.

We propose to optimize non-measurable properties with three techniques: (a) show the ranking of features assigned to qualitative properties that can satisfy non-functional requirements to the user (e.g., hint which features qualitatively improve a certain property), (b) automatically select features with a positive influence, and (c) avoid the selection of features with a negative influence during the computation of a near-optimal variant. For non-functional properties for which a stakeholder has manually measured and assigned metric values, we suggest to include them for an automated computation as we use it for the following two classes.

Measurable Properties Per Feature. This class contains only properties that can be measured on a metric scale with a user-defined metric (i.e., either customers or vendors provide suitable metrics). In addition, we require that we can either measure these properties directly for an individual feature or infer the results of the measurement of a variant to single features (i.e., a family-based approach).

The reason for defining this class is that we can use more efficient measurement techniques than for the class of measurable properties per variant. That is, the worst-case measurement complexity is $O(n)$, because we can measure each feature directly. Hence, this class represents no challenge for stakeholders to determine the influence of a feature on a non-functional property.

Depending on the implementation technique, we can measure, for example, footprint of a feature directly [Siegmund et al., 2008b]. If a feature maps one-to-one to a code unit (e.g., a component) that can be separately compiled, we can automatically measure each feature's footprint directly. We extract the results from these measurements and assign them to their corresponding features as shown in Figure 3.2, in which we assigned the measured footprint to each feature.

During program derivation, we suggest to find an optimized configuration by stating an objective function. For example, an objective function may minimize footprint of a variant. By using a constraint-satisfaction-problem solver that allows us to define objective functions, we compute, based on a partial feature selection, which (alternative) features should be selected to minimize footprint. For example, we select feature *Hash* rather than feature *Btree* to minimize the footprint of a variant (cf. Figure 3.2).

In Chapter 5, we outline how to measure features individually using the direct-measurement and family-based measurement.

Measurable Properties Per Variant. Some non-functional properties have either no meaning for single features or we are not able to break down measured non-functional properties of a variant to the individual influences of present features. For example, we cannot measure performance of a feature individually, because a feature's performance depends on other features of a variant and we usually cannot execute a feature in isolation. Hence, these properties are measurable only per variant. Considering the exponential number of variants, properties of this class raise the challenge how to find near-optimal variants for these kinds of properties.

Although it was stated before that breaking down properties of a variant to single features is not possible [Sincero et al., 2010], we propose a novel approach in the remaining thesis that makes it possible to infer the influence of individual features from the measurement of only few variants. Hence, similar to measurable properties per feature, we assign concrete values to features as we have shown for performance in Figure 3.2. This way, we can even compute a near-optimal configuration for this kind of properties and enable a multi-objective optimization with non-functional properties of different classes.

Note that the classification of a specific non-functional property depends on the customizable program and the application scenario – it is not general. This means that the same property can be located in different classes for different programs or

domains. Reasons for different classifications are, for example, different viewpoints and interpretations of stakeholders for the same property, different metrics and measurements. Also the domain of a customizable program may change the category of a property. For instance, in a web-service program, security may be measured via an intrusion-detection system resulting in quantifiable measures. In another scenario, security can only be qualitatively specified (e.g., with weak, medium, and strong security), like it is done in Windows 7.[3]

3.3. Incorporating Non-Functional Properties in Program Derivation

In this section, we propose a holistic approach to the optimization of non-functional properties in the program-derivation process. By holistic, we mean that we support the measurement and optimization of desired non-functional properties during program derivation. We support different classes of non-functional properties, which we described in the previous section. Here, we show how these different classes of non-functional properties are treated during program derivation and highlight the steps to derive a near-optimal variant.

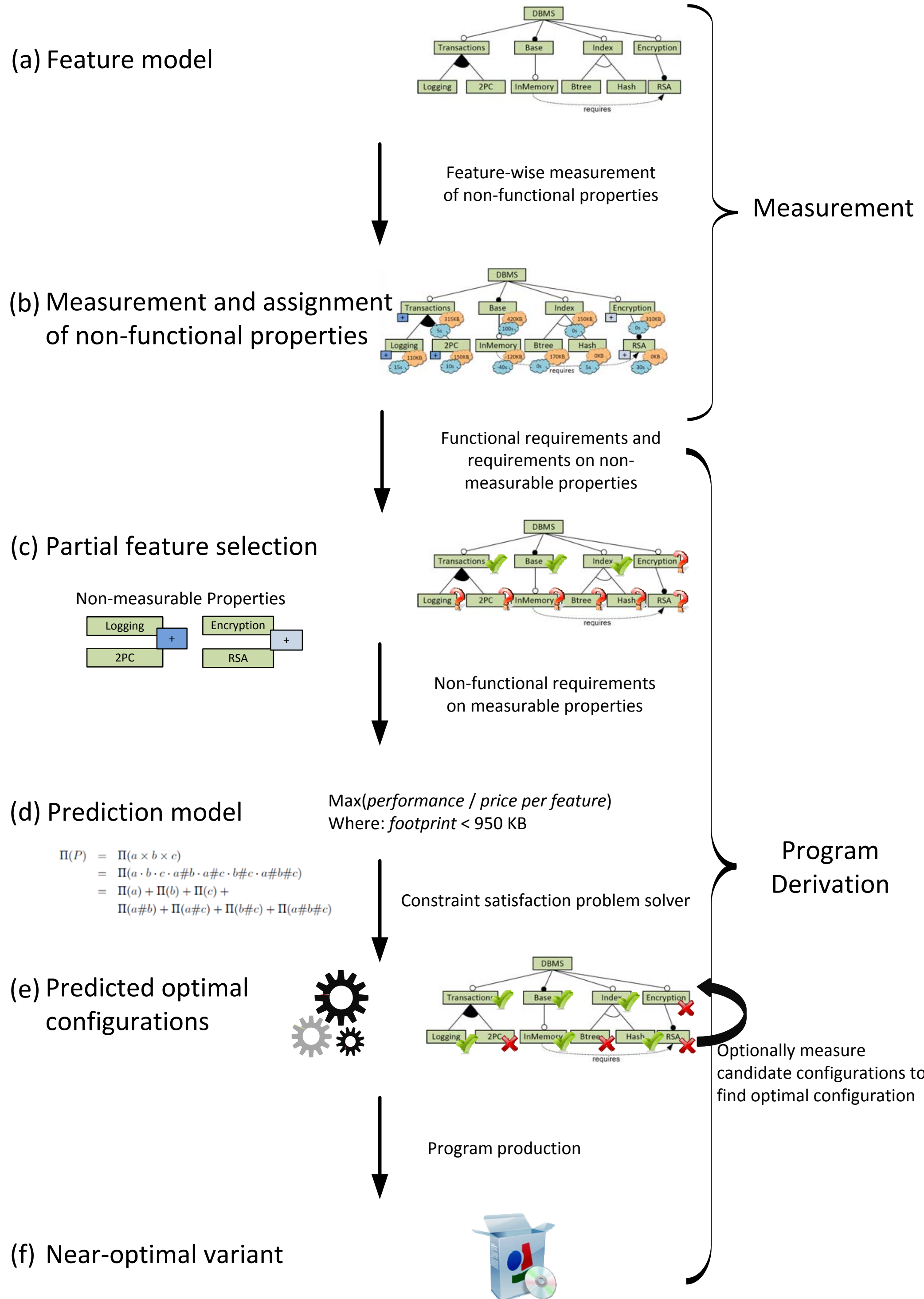
In Figure 3.3, we show the big picture of how to incorporate non-functional properties in program derivation. It consists of six steps:

- (a) Create a feature model, which describes features and variability of the customizable program (by vendor)
- (b) Assign non-measurable properties to features and measure the remaining properties per feature (by domain expert and vendor)
- (c) Define functional and non-functional requirements to obtain a partial feature selection (by customer)
- (d) Build the prediction model and state an objective function that expresses non-functional requirements (by domain expert and customer)
- (e) Obtain multiple configurations that are near-optimal solutions by predicting their non-functional properties and, optionally, identify the best configuration by measuring these configurations again (by vendor)
- (f) Produce the variant that corresponds to the chosen configuration (by vendor)

We describe in the following each of the steps in detail.

(a) Feature Model. The initial task to enable program derivation is to create a feature model (as shown in Figure 3.3). We use feature models to configure a variant.

[3]Source: `http://www.microsoft.com/de-de/security/pc-security/windows7.aspx` [Accesed: August 8th 2012]



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

Figure 3.3.: Big picture of incorporating non-functional properties into program derivation including measurement and prediction.

Hence, vendors of a customizable program use their domain and implementation knowledge to extract all customization options and develop an according feature model including all constraints and relationships among features. Usually, vendors develop a feature model before implementation, such that this development artifact already exists. For customizable programs that were developed traditionally (i.e., without dividing development in domain and application engineering as described in Section 2.3.1), this task may require some effort, because domain knowledge may be lost during development, such that not all relationships and constraints between features are known.

(b) Measurement and Assignment of Non-Functional Properties. The raw feature model builds the base to manually assign non-measurable non-functional properties. That is, a domain expert ranks features according to their qualitative influence on non-functional properties, as we describe in Section 3.2. The result is an enriched feature model, which we use to determine the influence of features on measurable non-functional properties either via a measurement per feature or a measurement per variant. The vendor has to perform these measurements only once per customizable program for non-functional properties that are independent of the program's workload (e.g., footprint).

If we measure non-functional properties that depend on a certain workload (e.g., performance of a DBMS depends on the type and quantity of queries as well as on the amount and diversity of stored data), we can either use a standard benchmark as it is done in many domains, such as databases and video encoding, or the realistic workload is given by a concrete customer. Since customers have to specify their requirements anyway, they can also specify their actual workload to measure non-functional properties in a realistic environment. These measurements need to be repeated for each customer, but produce more realistic statements about the influence of features on non-functional properties. Since we aim at conquering the exponential effort to measure customizable programs and automate the measurement process, we provide a practical solution that allows vendors without user-interaction to measure non-functional properties with customer-defined workloads in hours (or few days for highly variable programs).[4]

(c) Partial Feature Selection. The actual program derivation begins with the feature selection based on functional requirements. That is, customers select those features that provide the functionality needed in their application scenario. Also in this step, customers select features that satisfy requirements on qualitative non-functional properties. For example, if customers require high reliability of a DBMS variant, they select those features that were assigned with having a positive effect on reliability (e.g., *Transactions*). The result of this process is a partial feature selection.

[4]Note that our current and future work targets to extend our approach to measure a feature's influence on a variable workload. We outline this work in Section 9.3.

That is, some variability is bound, but there are decisions left to be selected to optimize the variant regarding non-functional properties.

(d) Prediction Model. In this step, we build our prediction model (presented in Chapter 4) based on the measurements and partial configurations. The prediction model allows us to reason about a variant's non-functional properties without the need to actually produce and measure it. This is the key benefit that makes our approach feasible in practice.

Furthermore, the customer defines an objective function that represents the customer requirements on measurable non-functional properties (i.e., the optimization goal). For example, if customers are interested in the best trade-off between performance and footprint and have a limited budget, they may define an objective function like this:

$$max(\frac{performance}{1000 * footprint}) \mid Price \leq 1000\,\$ \tag{3.1}$$

Specifying appropriate objective functions based on customers non-functional requirements is out of scope of this thesis. However, there is a large community that focuses on quantifying requirements – that is, making requirements explicit, such that we can use them for optimization.

(e) Predicting Optimal Configuration. Since the computation of finding an optimal configuration is NP-hard [White et al., 2009], we may be able to give only an approximately good feature selection. In this thesis, we use constraint-satisfaction-problem solver to compute an optimized feature selection. There are other approaches that approximate in less time near-optimal solutions [White et al., 2009]. Since our contribution is not the computation of the optimal configuration, but producing the necessary input with a suitable accuracy for such approaches, we are not limited to a specific technique.

The result of this prediction is either a single configuration or a list of candidate configurations that all may be suitable for the customer. If the customer requires a high reliability of finding the optimal variant and satisfying hard non-functional constraints (e.g., a variant must not exceed a certain response time), we can produce and measure the variants that correspond to the candidate configurations to completely omit approximation errors of the prediction process.

(f) Near-Optimal Variant. The last step is to produce the variant that corresponds to the selected configuration in the previous step. Again, we cannot guarantee that we actually found the optimal variant. We described in Section 2.2 different types of customization techniques. Depending on the type, the variant production differs.

For example, we map a configuration to a set of preprocessor flags and compile the corresponding variant or we map a configuration to a customized configuration file. At the end of this process, customers or users receive a variant that satisfies their functional requirements and is optimized to desired non-functional properties.

In the next chapter, we explain how we compute a variant's non-functional properties based on the influences of individual features. That is, we introduce our prediction model which we use as an input for a constraint-satisfaction-problem solver to compute a near-optimal variant.

3.4. Related Work

In the following, we present related work regarding our classification of non-functional properties and our program-derivation approach.

3.4.1. Models of Non-functional Properties

Quality models are used to bridge the gap between metrics and quality attributes. They describe which metrics allow us to measure the quality of which attribute. Furthermore, they model relationships between properties and ease the specification of non-functional requirements [Deissenboeck et al., 2009]. Since we aim at measuring and predicting non-functional properties, these quality models can be used as a foundation to identify which metric is suitable for which property. A quality model is defined by the ISO/IEC 14598 International Standard (standard for information technology - Software product evaluation - Part 1: General overview):
"The set of characteristics and the relationships between them which provides the basis for specifying requirements and evaluating quality." [International Organization for Standardization (ISO), 1999]

There are a number of non-functional properties including their classification described in the literature, for instance, McCall's quality model [Mccall et al., 1977], Boehm's quality model [Boehm et al., 1978], and the ISO 9126 quality model [International Organization for Standardization (ISO), 2001]. These models have a distinct purpose. For example, McCall's quality model bridges the gap between a customer's quality perspective and a developer's view on quality attributes. Hence, McCall describes factors based on an external view of software and quality criteria that describe the internal view of a software. A developer can use this model to derive suitable metrics (e.g., error tolerance and accuracy) to improve a quality factor (e.g., reliability). Boehm's quality model is a hierarchical model to refine and further specify characteristics from which a property is composed [Boehm et al., 1978]. For example, maintainability is refined to understandability, which in turn is refined to conciseness. Hence, Boehm qualitatively defines software quality with a given

set of metrics. In contrast to the mentioned models, our purpose is to classify non-functional properties, such that we can choose proper measurement techniques in the context of customizable programs. Some non-functional properties can be described only qualitatively, whereas other properties can be represented with metric-based values, so we cannot use the same optimization technique for all properties.

There are a number of approaches that target the development of programs with desired non-functional properties. These approaches, such as the non-functional requirement framework [Chung et al., 1999], i* framework [Yu, 1997], and KAOS [van Lamsweerde, 2001], are originally intended to help developers with design decisions to develop a software considering non-functional requirements. For customizable programs, software artifacts are usually already implemented when new customers derive a variant, but decisions regarding desired non-functional properties can be made during the variant-derivation process. Hence, these frameworks target traditional monolithic programs, in which non-functional properties must be optimized before and during development, whereas we concentrate on customizable programs and on the optimization after development.

3.4.2. Program-Derivation Approaches

The vast majority of variant-derivation tools focuses on reducing the complexity of the configuration process and supporting the user with advanced user interfaces during feature selection [Batory, 2005, Antkiewicz and Czarnecki, 2004, Czarnecki et al., 2004, Botterweck et al., 2007, Rabiser et al., 2007]. These tools often use satisfiability solvers or Prolog (e.g., in pure::variants[5]) to verify a configuration against the constraints of a feature model.

As we explained before, we use a constraint-satisfaction-problem solver to compute an optimized variant. There are also some approaches that allow a user to optimize the feature selection with regard to a specific non-functional property. Benavides et al. [2005, 2007] presented a technique based on constraint-satisfaction-problem solvers to find an optimal variant. The solver evaluates values attached to features in the feature model and then computes an optimal configuration for a small number of features. Their studies show that with an increasing number of features, the computation time exponentially grows. White et al. [2007, 2009] extended the program-derivation process by enabling the definition of resource constraints (i.e., non-functional requirements). Moreover, they propose a solution based on filtered Cartesian flattening to approximate a nearly optimal variant for even large-scale feature models. However, although required, both approaches do not present a way how to obtain the influences of features on non-functional properties. In contrast, we provide a holistic concept of including the measurement of non-functional properties in the whole program-derivation process. Furthermore, since we do not focus

[5]Source: `http://www.pure-systems.com/pure_variants.49.0.html` [Accessed: August 9th 2012]

on how to compute the optimal configuration (we provide the necessary input and use a standard solver), we can integrate their solutions in our holistic approach.

3.5. Summary

We presented our initial problem statement, in which we outlined that quantifying a feature's influence on non-functional properties is problematic in terms of diversity of non-functional properties and measurement effort. We explained why we target black-box programs to increase applicability. Furthermore, we argued that we have to measure non-functional properties after program development and not before, as it is done for specialized programs, because customizable programs are developed to cover application scenarios of a whole domain, such that concrete requirements of customers are often known only after development.

We proposed to classify non-functional properties into three classes: non-measurable properties, measurable properties per feature, and measurable properties per variant. We explained that by using these classes, we address the problem of measuring very diverse non-functional properties. We argued that we cannot target all kinds of non-functional properties, because we either cannot provide a meaningful metric to quantify the influence of features on a non-functional property or the measurement effort is too high and cannot be automated (e.g., in the case of human experiments). Hence, we specified how to measure and configure properties of the respective classes.

Finally, we presented the big picture of incorporating non-functional properties in the configuration of a customizable program to yield optimized variants. We explained six important steps of the program-derivation process: (a) feature-model creation, (b) measurement and assignment of non-functional properties, (c) specification of a partial feature selection based on functional requirements and requirements on qualitative non-functional properties, (d) building the prediction model to estimate non-functional properties of different configurations and specifying an objective function for later optimization, (e) computing different configurations according to the objective function, and (f) producing the variant that corresponds to the best found configuration.

With the big picture in mind, we describe our prediction model in the next section. We show how features and their influences are represented in the prediction model and how we can map this model to the derivation of an actual variant.

4. A Feature-Centric Prediction Model

This chapter shares material with the following papers:

- "Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties" in *VaMoS'08* and
- "Predicting Performance via Automated Feature-Interaction Detection." in *ICSE'12* [Siegmund et al., 2012a].

This chapter presents our prediction model, which receives a configuration as an input and estimates non-functional properties of its corresponding variant. In the big picture, we are located at the step, when a customer has made his partial feature selection and now wants to know what the non-functional properties of a corresponding variant are. Our prediction model is based on a feature-composition model, which we introduce with its important properties first. Afterwards, we describe the mapping between the feature-composition model and our prediction model. In this way, we introduce the concept of *terms*, which describe the impact of features and feature combinations (i.e., feature interactions) on a variant's non-functional properties.

Furthermore, we present how different non-functional properties can be supported via different mappings (i.e., homomorphisms), such that we can tailor the computation of a prediction to different properties. Afterwards, we describe how the model relates to feature models, because feature models are the central model to compute valid configuration and to visualize the variability of a program. Finally, we present an extension to feature models to support the definition of feature interactions and implementation units. This is useful to map all elements in the prediction model to elements in the extended feature model and vice versa.

4.1. Feature-Composition Model

We predict a variant's non-functional properties based on the corresponding feature selection. Hence, our prediction model relies on the feature composition. To this end, we use a recent model of feature composition by Batory et al. [2011]. This model treats programs as a sequential composition of features. If program P consists of features a, b, and c, we write: $P = a \cdot b \cdot c$, where $\cdot$ denotes the associative and commutative composition of features. That is, evaluating $a \cdot b \cdot c$ generates P. Lowercase letters (e.g, a) are *terms* representing either features or feature interactions. Capital letters denote compositions of one or more terms.

Feature interactions play a central role in this model. To work correctly, a program requires not only the implementation units that correspond to selected features, but also interaction units that ensure that features operate together in a desired way (e.g., interaction unit a#b denotes the interaction between features a and b); these interaction units are called lifters [Prehofer, 1997] or derivatives [Liu et al., 2006]. Consider the classic example from fire and flood control [Lee et al., 2004]. In this example, we have a flood-control (fc) sensor working with a fire-alarm (fa) sensor. If only one of fc or fa is present, the behavior is unambiguous: Water is turned on when fire is detected and turned off when a flood is detected. When fc and fa are both present, we observe an interaction $fc\#fa$ that turns water off after the fire sensor turned water on to prevent a fire. Obviously, we do not get our desired result, because a house burns down when both features are present. In code, we make this interaction explicit, such that we control this interaction with an appropriate behavior (i.e., never turn water off when fire is detected). Nevertheless, the interaction at a behavioral level is present whether we handle it or not.

As a result, a stakeholder wants not only the composition of implementation units that correspond to selected features, but also all necessary interaction units. [Batory et al., 2011] introduce the concept of a cross-product ($\times$) that maps a given feature selection a, b to the composition of its corresponding implementation units and interactions:

$$a \times b \quad = \quad a\#b \cdot a \cdot b \tag{4.1}$$

where features a and b are composed with their interaction unit $a\#b$.

[Batory et al., 2011] showed that the $\times$ operation is associative and commutative:

$$\begin{aligned} \mathit{Commutativity} &: \quad a \times b \quad = \quad b \times a \quad (4.2) \\ \mathit{Associativity} &: \quad (a \times b) \times c = a \times (b \times c) \quad (4.3) \end{aligned}$$

Interaction units are composed by the changes needed to modify the participating feature such that they work together correctly. Also for the interaction composition, commutativity and associativity holds:

$$\begin{aligned} \mathit{Commutativity} &: \quad a\#b \quad = \quad b\#a \quad (4.4) \\ \mathit{Associativity} &: \quad (a\#b)\#c = a\#(b\#c) \quad (4.5) \end{aligned}$$

Commutativity means that there is no order in which an interaction unit changes implementation units of a feature. Hence, the interaction unit abstracts from a certain implementation technique, in which ordering may matter. This way, an interaction unit comprises all permutations of interactions: $a\#b$ comprises interaction units $a\#b$ and $b\#a$. This already justifies the associativity of the # operation. There are also additional axioms, such as distributivity, that are not relevant for our work.

We use this terminology to specify feature selections, incorporate feature interactions, and relate these units to the prediction of non-functional properties.

4.2. Prediction Model

We consider the previous model as an abstract and uniform way to represent features and feature combinations of a customizable program. Our main idea is that we use the same model to not only describe functionality of a program, but also other characteristics of features, such as non-functional properties. From this model, we map a feature selection (i.e., terms of the abstract model) to distinct representations as shown in Figure 4.1. For example, we use the abstract model as a mapping to different implementation artifacts, such as components or feature modules. We denote a mapping from terms (i.e., features and feature interactions) in the abstract model to their corresponding implementations with the operator τ. That is, $\tau(a)$ represents the implementation unit of feature a.

By using these mappings, we make domain knowledge explicit. For example, we can use the mapping from a feature selection to its implementation units to identify which features map to the same implementation unit. For instance, features a and b map to the same component τc. Note that τc has no unambiguous mapping to a single feature (i.e., there is no feature c). Therefore, we write τc, instead of $\tau(c)$ to distinguish from a one-to-one mapping. In other words, τc comprises $\tau(a) \cdot \tau(b)$, which can be rewritten as $\tau c = \tau(a \cdot b)$. This explicit mapping helps us in identifying feature interactions for different non-functional properties. We describe in Chapter 7 feature interactions in detail.

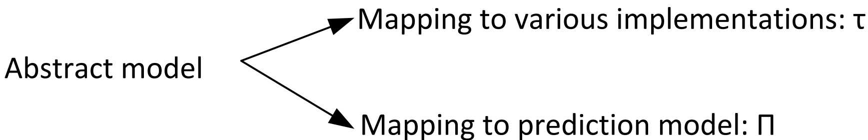


Figure 4.1.: Mapping feature selection of abstract model to other representations.

In contrast to the original feature-composition model, we map terms in a configuration to the impact of features and feature interactions (i.e., feature combinations) on non-functional properties, denoted with Π. That is, when selecting feature a and b, we map them to their influences to non-functional properties $\Pi(a)$ and $\Pi(b)$. Note that we specify the concrete non-functional property as follows: $\Pi_{Perf}(a)$ indicates the influence of feature a on performance and $\Pi_F(a)$ indicates the footprint (binary size) of feature a. We explain in Chapter 6 how to measure these terms.

From Equation 4.1, we know that when selecting multiple features, we have to consider in addition to the feature terms also feature-interaction terms. That is, non-functional properties of a variant with features a and b are not only caused by the individual influences of both features, $\Pi(a)$ and $\Pi(b)$, but also how these two features work in combination: $\Pi(a\#b)$. That is, interaction $\Pi(a\#b)$ describes how features a and b operate together at the level of non-functional properties, for

example, how performance changes when features *Btree* and *Transaction* work in combination.[1]

The benefit of using the feature-composition model is that it clearly states which terms exists for a configuration. This allows us to *selectively* determine which terms to measure to predict a variant's non-functional properties. If we determine all terms that exist for a customizable program, we would determine the influence of all feature combinations on non-functional properties. Clearly, this approach does not scale, because there is an exponential number of terms. However, we can identify which terms may be beneficial to determine to achieve a near-optimal prediction. That is, we can specify heuristics to determine only a subset of terms, which have a measurable effect on non-functional properties. This way, we can omit measurements without sacrificing too much accuracy in predictions. We present in Chapters 6-8 means to identify those relevant terms in order to find the best trade-off between measurement effort and prediction accuracy.

Since we consider terms as impacts on non-functional properties, we have to specify how to aggregate these influences of each feature on a non-functional property. In Equation 4.1, terms represent implementation units such that the aggregation function is the feature composition operation $(\cdot)$. Considering non-functional properties, we need other aggregation functions. We use different mappings to translate the feature-composition model to our prediction model. In the following equation, we define a mapping that uses the addition to predict a non-functional property:

$$\begin{aligned} \Pi(a \times b) &= \Pi(a \cdot b \cdot a\#b) \\ &= \Pi(a)\Pi(\cdot)\Pi(b)\Pi(\cdot)\Pi(a\#b) \text{ // } \textit{Homomorphism} \\ &= \Pi(a) + \Pi(b) + \Pi(a\#b) \end{aligned} \tag{4.6}$$

We predict non-functional properties of a program with features a and b with $\Pi(a \times b)$ using the influences of both feature terms $\Pi(a)$ and $\Pi(b)$ and the feature-interaction term $\Pi(a\#b)$. Hence, our prediction model requires us to know how each feature influences a non-functional property and, optionally, how the combination of features affect a non-functional property, whose effect may be negligible.

An aggregation function may differ depending on the non-functional property, the customizable program, how non-functional properties are measured, and which metric is used to quantify a property. For example, aggregating maintainability of features using the cyclomatic-complexity metric requires a *max* aggregation function [Mccall et al., 1977]. That is, the worst-case complexity of any selected feature

[1] We assume that the composition order has no effect on non-functional properties. This might be different for some implementation techniques, such as components, in which the composition changes performance and other properties. As a prerequisite, we consider that there is an absolute order at which features are composed or executed.

defines the overall complexity of a variant. We define the mapping to predict cyclomatic complexity (CC) using the maximum as the aggregation function as follows:

$$\begin{aligned} \Pi_{CC}(a \times b) &= \Pi_{CC}(a \cdot b \cdot a\#b) \\ &= \Pi_{CC}(a)\ \Pi_{CC}(\cdot)\ \Pi_{CC}(b)\ \Pi_{CC}(\cdot)\ \Pi_{CC}(b) \\ &= Max(\Pi_{CC}(a), Max(\Pi_{CC}(b), \Pi_{CC}(a\#b))) \end{aligned} \quad (4.7)$$

Feature-Interaction Terms. Our prediction model includes feature-interaction terms to quantify the behavior of how features operate together. If a feature combination has a relevant, measurable effect on a non-functional property, we have to consider and include this term to improve prediction accuracy. We use a basic result that follows from 4.1 and 4.7. If we can measure the influence of a feature on a non-functional property (e.g., $\Pi(a)$ and $\Pi(b)$) and the non-functional properties of a variant that includes both features, $\Pi(a \times b)$, we can compute the value of $\Pi(a\#b)$:

$$\begin{aligned} \Pi(a\#b) &= \Pi(a \times b) - \Pi(a) - \Pi(b) \quad (4.8) \\ &= \Pi(a) + \Pi(b) + \Pi(a\#b) - \Pi(a) - \Pi(b) \\ &= \Pi(a\#b) \end{aligned}$$

Unfortunately, the number of feature-interaction terms grows exponentially with the number of features in a configuration. Until now, we have considered feature interactions only between pairs of features, but their number grows exponentially with the number of selected features. For example, to predict variant's P properties with three features, we have to determine seven terms (three feature terms and four feature-interaction terms):

$$\begin{aligned} \Pi(P) &= \Pi(a \times b \times c) \\ &= \Pi(a \cdot b \cdot c \cdot a\#b \cdot a\#c \cdot b\#c \cdot a\#b\#c) \\ &= \Pi(a) + \Pi(b) + \Pi(c) + \\ &\quad\ \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c) \end{aligned} \quad (4.9)$$

The possible exponential number of terms raises the question how to find only the *relevant* feature-interaction terms. Relevant means that we need to consider only those terms for prediction that have a measurable and observable effect on a variant's non-functional properties. Finding only relevant feature-interactions is far from trivial. We will discuss in the following chapters how we determine these interaction terms and at what cost in terms of measurement effort.

The simplicity of the prediction model is the key to be feasible for highly customizable programs, because it allows us to increase the number of measurements if needed. Overall, we achieve the following benefits:

- **Scalable Prediction.** In our prediction model, we need to quantify at least the feature terms, which requires a linear number of measurements. We can increase prediction accuracy by measuring more variants to determine the influence of feature-interaction terms on non-functional properties. This process can be stopped when a sufficient accuracy is reached or the measurement effort becomes too high.
- **Reduction of Terms.** If we identify that a feature term has no effect on a certain non-functional property (e.g., a command-line parameter has no effect on the binary size of a program), we can remove this term from the model and all feature-interaction terms in which this feature participates. This can result in considerably fewer measurements for particular non-functional properties.
- **Generality.** The model allows us to predict all quantifiable non-functional properties for which we can define a meaningful mapping between feature composition and prediction of non-functional properties. That is, if we can specify a sound aggregation function, we can use this model. Examples include addition and multiplication for performance, energy and memory consumption, max, min, and average for software measures, etc.
- **Extendability.** Since all terms in our prediction model are an abstract representation, we can map them to cost functions instead of concrete values to predict non-functional properties with a variable workload. Moreover, we can also model other variable factors, such as the environment, the operating system, and different hardware configurations as additional terms. The prediction model stays the same, but we may have to combine multiple aggregation functions for a single non-functional property. For example, if we predict performance, we may sum up the performance influences of all features and feature-interaction terms. Additionally, we use multiplication to predict performance for different CPUs.

In the following, we describe the mapping of a configuration in the feature model to a configuration in the prediction model. We need this mapping to account for dependencies and relationships among features, to derive only valid variants.

4.3. Relating Feature Model to Prediction Model

So far, we ignored dependencies between features, as documented in feature models. However, to derive a valid variant, we have to consider them. Furthermore, these dependencies specify which terms in the prediction model can exist in the same configuration. Hence, we draw the relationship between a feature model, which expresses all valid variants, the prediction model, which takes a valid feature selection and estimates non-functional properties of the corresponding variant, and the implementation units, which realize a feature's functionality. To establish this relationship, we propose an extension to feature models, called *product-line model* [Siegmund et al., 2008a], which we already used in Chapter 3. Our product-line model captures

non-functional properties of features, differentiates between different classes of properties (which we described in Section 3.2), and models implementation units as well as feature interactions. In the following, we describe the new concepts and explain the benefits of having them integrated in a single model.

Implementation Units. Implementation units realize a feature's functionality. It is an abstract representation to support any implementation technique. For example, an implementation unit can be a component, a feature module, or even an `if`-branch in normal program code, which is executed when a certain command-line parameter is defined. Implementation units realize a feature's functionality. Since we target a black-box approach, we do not require to model them, but we can use this information to determine which feature-interaction terms might be relevant for measurement.

The integration of implementation units into the feature model requires two conditions. First, implementation units can only be child elements of features or other implementation units. An implementation unit cannot be a root node or a parent of a feature, because features are defined during the domain analysis, which precedes the implementation phase (the feature model written once is solely extended but not changed). Second, we need an additional relation to represent the interaction between implementation units (i.e., derivatives).

A further benefit of capturing implementation units is to establish a mapping between features, which might be represented in a stakeholder-convenient way, and the implementation units that a program generator needs to know to generate a corresponding variant. For example, we may model a set of command-line parameters as implementation units as the realization of a certain domain feature. We map features to implementation units using τ, as we already illustrated. There are different cases for this mapping:

- One-to-one: Features a and b map to their implementation units $\tau(a)$ and $\tau(b)$.
- N-to-one: Features a and b map to the single implementation unit $\tau(a \cdot b)$.
- M-to-n: Features a and b map to the implementation units $\tau(a) \cdot \tau(b) \cdot \tau(a\#b)$.

Note that there is no case in which a single feature maps to several implementation units that realize only a single feature's functionality (i.e., there is no one-to-n mapping), because we comprise multiple physically existing implementation artifacts (e.g., multiple pieces of code that belong only to one feature) to a single abstract implementation unit. The reason is that it does not make any difference for program production and computation of non-functional properties whether a feature is realized by multiple pieces of code when they all compiled at once. It makes only a difference when these artifacts are simultaneously used by other features, because this causes different expectations of a variant's non-functional properties for different feature selections.

Non-Functional Properties. The product-line model supports the assignment of qualitative properties (with ordinal values) and quantifiable properties (with actually measured metric values). We do not have to differentiate between feature-wise and variant-wise quantifiable properties, because the main difference between these classes of properties is how to measure them and not how to configure and model them. Hence, this extension to feature models is similar to the one of Benavides et al. [2005] with the addition of qualitative properties. The mapping of assigned non-functional properties to our prediction model is that the quantifiable property defines which mapping and therefore which aggregation function to use. The mapping of features to their concrete influence on non-functional properties is realized by an assignment of their corresponding impacts: $\Pi_F(a)$ represents feature a's assigned influence on footprint. Qualitative properties are not mapped to the prediction model, but configured directly as we described in Section 3.3.

Feature Interactions. As an important extension to feature models, we introduce the concept of *feature interactions* in our product-line model [Siegmund et al., 2008a]. We map interactions between features to our prediction model as feature-interaction terms, which are equivalent to the abstract model. Since we do not consider implementation units in our prediction model, we cannot map these interactions to the prediction model as they have no directly corresponding term (i.e., $\tau(a\#b)$ has no representation in the prediction model). However, we compute – using the mapping from features to implementation units – at which feature combinations the interaction of the implementation unit would occur. That is, if $\tau(a\#b)$ or $\tau(a \cdot b)$ exists, we mark the interaction term $\Pi(a\#b)$ in the prediction model as a relevant term. That is, we expect that this interaction has an influence on non-functional properties. Hence, the product-line model allows us to specify which terms in the prediction model need to be quantified. Note that it is also possible that features interact without having an interaction at the level of source code. Hence, this is only one method to identify a subset of all possible feature interactions.

Example. In Figure 4.2, we show a concrete example of the product-line model of Berkeley DB's C version.[2] We show the measured footprint of each feature (described in Section 4) and also an assigned price for features for illustration purposes. That is, when selecting feature *Hash*, we map it to the terms $\Pi_{Price}(Hash) = 125\,\$$ and $\Pi_F(Hash) = 113\,KB$ in the prediction model.

Furthermore, we defined two qualitative properties, security and reliability, and highlight which features have a positive influence on these properties. For example, feature *Verification* has a positive effect on reliability for a DBMS variant. As

[2] Note that the model does not correspond exactly to the official version: `http://oracle.com/technetwork/products/berkeleydb/`. For instance, we model an additional Btree implementation and assigned a price per feature only for illustration of the concepts.

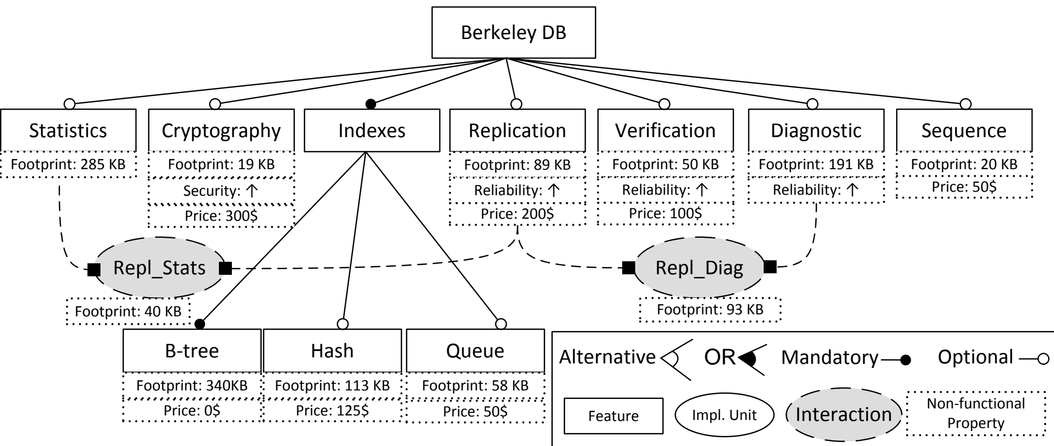


Figure 4.2.: Product-line model of Berkeley DB with assigned non-functional properties. Footprint represents measured binary size per feature. The up-arrow visualizes an improvement for a qualitative property.

explained before, we do not map qualitative properties to the prediction model, because we require metric values for our prediction.

In our example, we model two feature interactions, *Repl_Stats* and *Repl_Diag*, which increase footprint of a variant when the corresponding features are selected in combination. That is, we map the feature interaction *Repl_Stats* to $\Pi_F(\mathit{Repl\#Stats}) = 40\,KB$ in the prediction model. This interaction does not have a corresponding mapping to an implementation unit (i.e., $\tau(\mathit{Repl\#Stats})$ does not exist), because the selection of both features, *Replication* and *Statistics*, does not require to select an additional distinct implementation unit. That is, there is no additional module that needs to be defined for a program generator to produce the corresponding variant. In the case of Berkeley DB, this feature interaction is caused by a nested `#ifdef` statement. That is, just by defining $\tau(\mathit{Replication})$ and $\tau(\mathit{Statistics})$ additional code is included in a variant.

To reduce complexity of the product-line model, we omit implementation units if there is a one-to-one mapping between a feature and its implementation unit.

When we want to predict the footprint of a variant V with features *Statistics (s)*, *Replication (r)*, *Indexes (i)*, and *Btree (b)*, we yield the following prediction equation using the product-line model of Figure 4.2:

$$\begin{aligned}
\Pi_F(V) &= \Pi_F(s \times r \times i \times b) \\
&= \Pi_F(s) + \Pi_F(r) + \Pi_F(i) + \Pi_F(b) + \\
&\quad \Pi_F(s\#r) + \Pi_F(s\#i) + \Pi_F(s\#b) + \\
&\quad ... \text{ // } \textit{all interaction terms} \\
&= \Pi_F(s) + \Pi_F(r) + \Pi_F(b) + \Pi_F(s\#r) \text{ // } \textit{only relevant terms} \\
&= 285\,KB + 89\,KB + 340\,KB + 40\,KB \\
&= 754\,KB
\end{aligned}$$

Hence, we predict that the given equation yields a footprint of 2,214 KB. In this example, we removed all terms that are either unknown (e.g., we do not know about the influence of most of the interactions on footprint) or zero (i.e., in the case of feature *Index*). To know which terms are relevant, we must determine them in the measurement phase, which precedes the program-derivation phase (cf. Section 3.3). We show in the following chapters how to determine which terms affect non-functional properties and to what extent.

4.4. Related Work

Model-based predictions are common [Balsamo et al., 2004, Witten and Frank, 2005]. For example, linear and multiple regression explore relationships between input parameters and measurements. Based on a regression model, different estimation methods (e.g., ordinary least squares) can be used to predict performance for specific input parameters. Bayesian (or belief) networks are used to model dependencies between variables in a network [Jensen and Nielsen, 2007]. They are used to learn causal relationships and hence may be applicable to detect feature interactions. Furthermore, machine-learning approaches can be used to find the correlation between a configuration and a measurement (e.g., *canonical correlation analysis* [Mardia et al., 1980]). It uses dataset pairs to identify those linear combinations of variables with the best correlation. *Principal component analysis* [Hotelling, 1933] finds dimensions of maximal variance in a dataset that can also be used to detect interactions. Ganapathi et al. [2009] provides an analysis for different machine-learning approaches in the context of performance prediction of database queries, which is an important non-functional property.

The feasibility of model-based approaches depends on the application scenario and program to be analyzed. Our work differs in that it offers a general way to produce accurate predictions independent of the application scenario and non-functional property. By using terms (i.e., feature terms and feature-interaction terms) in our

prediction model, we abstract from a concrete realization. That is, we can set different approaches on top of this model (e.g., heuristics to determine which terms have to be quantified) to reduce measurement effort.

There are a number of approaches that predict output parameters for given input parameters using statistical learning methods. Using measurements as a training set and the configuration for input parameters, we can define this prediction as a nonlinear regression problem, for which we can use hierarchical clustering approaches to map measurement results to certain feature combinations [Hastie et al., 2009]. Furthermore, we can use classification and regression trees for the prediction step [Breiman et al., 1984]. The idea is to measure combinations of feature and store their measured non-functional properties. When predicting a configuration, we search for the most similar feature combination stored in the model and predict a corresponding value. However, many learning techniques considerably depend on the quantity and quality of the training sets and overfitting is a serious practical problem [Mitchell, 1997]. We overcome the problems above by providing a holistic approach that combines measurement and prediction. That is, our prediction model specifies the terms that have to be determined and in the measurement process, we specify and measure the configurations to determine the terms (i.e., we define the training set). Furthermore, we can quantitatively describe what each feature and each feature interaction contribute to a non-functional property, which is usually not possible with this granularity for statistical learning approaches.

Krogmann et al. [2010] combine monitoring data, genetic programming, and reverse engineering to reduce the number of measurements to create a platform-independent behavioral model of components. For a platform-specific prediction, they use bytecode-benchmark results of concrete systems to parameterize the behavior model. Happe et al. [2011] present a compositional reasoning approach, based on the *Palladio component model*. The idea is that each component specifies its resource demands and predicted execution time in a global repository. We predict not only performance, but all measurable non-functional properties independently of the used programming language, implementation technique, and availability of bytecode.

Other approaches usually require either expert knowledge to, for example, specify components' performance behavior [Bertolino and Mirandola, 2003, Aigner et al., 2003], or use connectors and meta-programs to monitor the program flow [Woodside and Litoiu, 2008]. We concentrate on end users or stakeholders that have no domain knowledge available and cannot intercept or monitor an application (e.g., a user that installs a Linux kernel instead of a kernel developer). Hence, we provide a general approach to predict non-functional properties of customizable black-box programs.

Also in this vein, model-driven-engineering-based work uses feature models to customize or synthesize performance models (e.g. [Tawhid and Petriu, 2011]). This line of research requires up-front and detailed knowledge of domain-specific perfor-

mance modeling, where tuning predictions for accuracy can be difficult. Our approach avoids these problems by directly measuring non-functional properties.

4.5. Summary

We presented our prediction model based on a model of feature composition. We describe how to map this model to the prediction of different non-functional properties by means of different mappings. We introduced the concept of feature terms and feature-interaction terms as central elements for the prediction.

We explained the relationship between a feature model and the prediction model, such that, based on a valid configuration, we can predict in advance the expected non-functional properties. Next, we proposed an extension to feature models, called product-line model, to represent feature interactions, which map to feature-interaction terms in the prediction model, in feature models. Furthermore, we described how non-functional properties can be assigned to features in the product-line model and how the optional definition of implementation units helps in identifying possible feature interactions.

This concludes Part I in this thesis. We showed the big picture of how to derive an optimized variant from a customizable program. We explained the challenges to enable the incorporation of non-functional properties in program derivation (e.g., unknown influence of features on non-functional properties and exponential number of variants) and proposed a new classification of non-functional properties to define suitable configuration and measurement techniques. We proposed our prediction model, which uses terms to express the influence of features and feature combinations on non-functional properties. We explained how these terms and the prediction model map to feature models, which are commonly used for configuration as they describe all valid variants.

Part II.

Measurement

5. Overview of Measurement Strategies

This chapter shares material with the following papers:

- "Measurement and Optimization of Non-functional Properties" in *APSEC'08* [Siegmund et al., 2008b] and
- "SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines" in *Software Quality Journal* [Siegmund et al., 2012c],

In Part 2, we show how to determine the terms in the prediction model. That is, we show how we can approximate the influence of individual features on non-functional properties. In Chapter 6, we present our initial approach that determines only the terms that correspond to features in the prediction model. In Chapter 7, we extend this approach by determining terms that correspond to (a) known feature interactions and (b) all pair-wise interactions. Finally, we propose an automated feature-interaction-detection approach in Chapter 8, which determines terms in the prediction model based on heuristics. We developed these heuristics from the observations and insights we gained by analyzing the evaluation of the prior approaches.

The main challenge to enable the prediction of a variant's non-functional properties is how to measure the impact of each feature on non-functional properties. In this chapter, we define goals that a measurement strategy has to fulfill for a practical usage in the context of customizable programs. Afterwards, we explain three measurement strategies, *direct measurement*, *family-based measurement*, and *feature-wise measurement*. We rate their feasibility according to our defined goals. We developed these measurement strategies, because they correspond to general strategies of analyzing variable software [Thüm et al., 2012]. Finally, we compare the three strategies and conclude that feature-wise measurement satisfies our defined goals.

Measurement Goals. To enable the prediction of non-functional properties, we need a measurement strategy that measures the influence of individual features on non-functional properties. We define several goals that a strategy has to accomplish. These goals are driven by the necessity (a) to make the measurement feasibility for highly variable programs, (b) to increase the applicability and generality regarding supported non-functional properties and our targeted black-box approach, and (c) to enable accurate predictions. In the following, we explain each goal in detail:

1. **Measurement Effort.** Measurement effort is an important factor that specifies whether a measurement strategy is feasible for highly variable programs.

Since we can produce millions of variants with only few independent and optional features, a feasible strategy should require only a linear number of measurements with respect to the number of features.

2. **Prediction Accuracy.** The more accurate we measure the influence of individual features, the more accurate are our predictions. Hence, an important goal is that the measurement itself quantifies the true influence of a feature, such that the measurement is not disturbed by other factors, for example the presence of other features or compiler optimizations.

3. **Applicability.** To increase applicability, a measurement strategy must not require implementation artifacts. Otherwise, we could not support black-box programs, which is, however, one of our overall goals in this thesis. Furthermore, a strategy must not depend on a certain implementation technique to not limit applicability. For example, if we can measure performance only for customizable programs implemented with components, but not for preprocessor-based programs, then our goal of a broad applicability is not accomplished.

4. **Generality.** We aim at measuring all measurable non-functional properties. That is, a measurement strategy must not exclude some non-functional properties that are, in general, measurable, but not with this strategy. Hence, we target a general solution.

5. **Realistic Environment.** Our last goal is that a measurement strategy has to be applicable in a realistic environment to produce realistic measurement results. That is, when measuring a program, we should measure it as if it were practically used. For example, measuring binary size or performance of a generated program without compiler optimization is not realistic, because, in practice, we would use compiler optimizations, which can substantially change the influence of features on non-functional properties. From a mathematical point a view, in practice, we measure the following properties: $\Pi_{opt}(a \cdot b) = \Pi_{opt}(a)\ \Pi_{opt}(\cdot)\ \Pi_{opt}(b) \neq \Pi_{unopt}(a \cdot b)$. Hence, the prediction of an optimized variant might be inaccurate.

We use measurement effort and prediction accuracy in the remaining thesis as key factors to quantitatively evaluate and rate an approach. Applicability, generality, and realistic environment are mainly discussed in this section, because they are qualitative factors, which we can use to quickly rule out inappropriate measurement strategies. Based on these goals, we propose and compare different measurement strategies.

5.1. Direct Measurement

Direct measurement means that we measure not a whole variant, but features directly. That is, we perform our measurements on the corresponding implementation

units. Since we know the mapping between features and implementation units from the product-line model, we know which implementation units have to be measured for which feature. Hence, the general approach is as follows: $\Pi(a) = \Pi(\tau(a))$. We can assign the measured result either to the implementation unit or to the corresponding feature, if a one-to-one mapping is present.

***Definition* 1.** *Direct Measurement*: A measurement of a customizable program is called a direct measurement if (a) it operates only on the features' individual artifacts (i.e., measurement of a feature's source code or object files) and (b) the measurement is performed for each feature and physically existing feature interaction (e.g., derivatives or nested `#ifdefs`) in isolation.

The benefit of a direct measurement is that we need only a linear number of measurements. We need to measure each feature only once to quantify its influence on a non-functional property. Furthermore, the analysis of the measurements is simple, because each measurement result maps directly to a feature. This is different to measuring a variant, because we would have to extract the individual influence of a feature, which might be difficult. Another benefit is that a direct measurement is fast, because we often do not have to generate and even execute a corresponding variant. Finally, when a customizable program changes (e.g., additional features are added), we need to measure only the added implementation units and not all features again. Hence, we conclude that the measurement effort is low (goal 1) and this strategy is optimal for non-functional properties of the class measurable properties per feature (cf. Section 3.2).

Unfortunately, the direct measurement strategy has significant limitations. First, we can measure only a subset of all quantifiable non-functional properties. That is, we cannot measure properties that emerge in a generated or running program, because we analyze only an individual feature at a time. Hence, we cannot perform measurements for the properties of the class measurable properties per variant (e.g., performance, main-memory consumption, and energy consumption). We conclude that generality is limited (goal 4).

Second, this strategy requires that features either are implemented in a modular way, such that we can perform our measurements on each module, or that features map one-to-one to produced artifacts (e.g., object files), which we can measure individually. This precondition limits applicability, such that we cannot measure already compiled customizable programs. This excludes black-box programs, which rely on load-time or run-time customization techniques, such as command-line parameters or configuration files. In a black-box program, we cannot measure implementation units of features individually, because we have no information about the mapping from features to implementation units and we do not have access to implementation units. That is, features a and b map to $\tau(a \cdot b)$ and not to $\tau(a) \cdot \tau(b)$. Hence, we can only determine $\Pi(\tau(a \cdot b))$, but we need to determine $\Pi(\tau(a))$ and $\Pi(\tau(b))$. We conclude that also applicability is limited (goal 3).

Third, we cannot quantify the influence of feature interactions on non-functional properties if they are not implemented as separate implementation modules or if we cannot identify the produced artifacts that emerge due to a feature interaction. Finally, a direct measurement can degrade accuracy of predictions, because to be able to measure certain properties, such as binary size, we may need to construct an artificial environment (e.g., generating interfaces to be able to compile individual modules). This, however, may prohibit optimizations, such as method inlining by compilers, which cause non-realistic measurement results. Using these artificial measurements for predicting production systems can cause significant inaccuracies. We conclude that both goals, prediction accuracy (goal 2) and realistic environment (goal 5), are not accomplished by this strategy. We summarize our findings in Table 5.1.

1. Effort	2. Accuracy	3. Applicability	4. Generality	5. Environment
satisfied	not satisfied	not satisfied	not satisfied	not satisfied

Table 5.1.: Relating direct measurement to goals.

5.2. Family-Based Measurement

Family-based measurement is a strategy in which we measure all features at once by evaluating valid feature combinations given by a feature model. Hence, we measure the influence of all features in a single step and infer afterwards how each feature contributes to the overall measured property. To this end, we produce a variant that contains all features: $\tau(a \cdot b \cdot ...)$ and extract with specialized measurement techniques the individual influence of features: $\tau(\Pi(a) \cdot \Pi(b) \cdot ...)$. Hence, family-based measurement results in a white-box approach. Possible techniques are symbolic execution [King, 1976], program simulators [Parnas, 1972], or monitoring and profiling approaches.

An analogy how such a measurement can be realized for variable software provide approaches that ensure type safety in software product lines [Apel et al., 2010a]. In an automated analysis, a program performs reference checks on an abstract syntax tree that encodes valid feature combinations. If such a check identifies that a valid variant exists in which a method call cannot be referenced, because a feature is missing, then an error is given, such that either the feature model must be changed to require existence of the missing feature or the code must be fixed. Regarding non-functional properties, such a variability-aware analysis might also be implemented, but it is unclear how non-functional properties, such as performance, can be measured, because probably all feature combinations must be executed in a single (or few) run.

***Definition* 2.** *Family-Based Measurement*: A measurement of a customizable program is called a family-based measurement if it (a) operates on all features at once and (b) maps the result based on the encoded variability to the individual influences of all features.

The advantage of this strategy is that we need to measure all implementation units only once. That is, performing a measurement has a constant complexity. We conclude that the measurement effort is very low (goal 1).

The disadvantages of the family-based measurement are similar to the direct measurement approach. In most cases, we cannot execute the variant containing all features, because a customizable program usually has alternative features. This, however, prohibits measurement of non-functional properties that emerge only in running variants (i.e., it excludes the class measurable properties per variant). Hence, generality is limited (goal 4).

Furthermore, we need special implementation techniques that allow us to produce a variant that consists of all features including alternatives and to manipulate the program code to include monitoring capabilities as well as variability information from a feature model. Hence, this measurement strategy relies on a white-box approach rather than on a black-box approach, such that applicability is limited (goal 3).

A further problem is the difficulty to identify a feature's influence on non-functional properties in this variant. We need means to extract this information, which in turn disturb accuracy of measurements (goal 2). The family-based strategy has the same drawbacks as the direct measurement regarding feature-interaction detection (we can measure only explicitly implemented interactions) and an artificial measurement environment, which decreases prediction inaccuracies for the production system (goal 5). We relate our findings to the defined goals in Table 5.2. Although there are a number of drawbacks, this approach is promising for white-box programs in future work, such that we can improve accuracy and generality.

1. Effort	2. Accuracy	3. Applicability	4. Generality	5. Environment
satisfied	not satisfied	not satisfied	not satisfied	not satisfied

Table 5.2.: Relating family-based measurement to goals.

5.3. Feature-Wise Measurement

Feature-wise measurement is a strategy in which we measure non-functional properties of two valid variants that differ in a single feature. We interpret the delta of these two measurements as the influence of the respective feature on non-functional

properties. For example, to determine the influence features a and b on performance, we measure the following variants:

$$\begin{aligned}
\Pi_{Perf}(a) &= \Pi_{Perf}(a \times \emptyset) - \Pi_{Perf}(\emptyset) \\
&= 40s - 0s \\
&= 40s \\
\Pi_{Perf}(b) &= \Pi_{Perf}(a \times b) - \Pi_{Perf}(a) \\
&= 90s - 40s \\
&= 50s
\end{aligned}$$

On the right side of the equations are only two measured variants. These variants differ only in the feature to be measured. This results in a worst-case complexity of $O(n)$. We describe this approach in Chapter 6 in detail. To quantify the influence of feature interactions on non-functional properties, we generate a variant that consists of the combination of features that cause an interaction and compare the measurement of this variant against our prediction. We interpret the delta of this comparison as the influence of the feature interaction on non-functional properties.

***Definition* 3.** *Feature-Wise Measurement*: A measurement of a customizable program is called feature-wise measurement if (a) it operates only on produced variants and (b) uses the results of two variants that differ in a single feature to compute the influence of the respective feature on non-functional properties.

Feature-wise measurement has a number of benefits. First, we need only a linear number of measurements to quantify the influence of all features on non-functional properties. That is, we need two measurements per feature to determine the delta. Second, this approach is scalable in the number of measurements. From our prediction model in Section 4.2, we know that there are an exponential number of terms. With this approach, we can measure these terms directly to, for example, improve prediction accuracy, and to find a sweet-spot between measurement effort and accuracy (goals 1 and 4).

Third, with feature-wise measurement, we measure only the produced variants. That is, we measure black-box programs with standard measurement techniques, which do not need to consider the context of variable software. This maximizes applicability, because this strategy is independent of how a program is customized or implemented (goal 3). Furthermore, measuring variants directly means that we can measure all kinds of measurable non-functional properties. Hence, this strategy maximizes also generality (goal 4).

Finally, we can measure variants under realistic conditions. That is, we can turn all compiler optimizations on, we do not have to extract information from a runtime environment (e.g., no need to profile a Java program with the Java runtime

environment), and we can run the variant on the customer's system when measuring non-functionality properties. We conclude that goal 5 is satisfied.

The main drawback of feature-wise measurement is the required measurement effort when many feature interactions exist (i.e., if we have to determine many terms of the prediction model). As we show in Chapter 7, we need to determine all relevant feature-interaction terms to enable a precise prediction. We conclude that measurement effort and prediction accuracy depend on each other in this strategy (summarized in Table 5.3). Hence, we have to find a sweet-spot between both of them. In Chapter 8, we show how to find this sweet-spot using three heuristics.

1. Effort	2. Accuracy	3. Applicability	4. Generality	5. Environment
partly satisfied	partly satisfied	satisfied	satisfied	satisfied

Table 5.3.: Relating feature-wise measurement to goals.

5.4. Comparison of Strategies

In Table 5.4, we provide a comparison of the different measurement strategies, including a pure brute-force approach. The best trade-off between measurement effort, prediction accuracy, applicability, and supported non-functional properties provides the feature-wise measurement. Although the brute-force approach seems to be another good option, it is not feasible for highly customizable programs because of an exponential number of variants that would be needed to measure. Here, feature-wise measurement clearly outperforms brute force.

Direct measurement has the benefit of less measurement effort. Although it has only limited applicability, it is reasonable to use this technique whenever a non-functional property can be measured with this approach. As we have shown in previous work [Siegmund et al., 2012c], a possible use case are code metrics, such as cyclomatic complexity [McCabe, 1976]. For this metric, we need to measure only each feature's implementation units to determine their complexity value.

The main drawback of family-based measurement is the use of white-box measurements. Since a major contribution in this thesis is to give a general applicable solution, family-based measurement is not feasible. However, we believe that with further research, many of the drawbacks (e.g., regarding accuracy and generality) can be eliminated. Hence, we see more potential in this solution than for direct measurement.

We summarize that for the defined goals, feature-wise measurements seems to be the only appropriate solution. Again, we selected these strategies based on more general analysis strategies for variable programs and because of the necessity of

Strategy	1. Effort	2. Accuracy	3. Applicability	4. Generality	5. Environment
Direct	+	−	−	−	−
Family-based	+	−	−	−	−
Feature-wise	+−	+	+	+	+
Brute force	−	+	+	+	+

Table 5.4.: Comparision between the three measurement strategies and a brute-force approach.

determine a feature's influence on non-functional properties, which, for example, excludes statistical learning-based approaches. Hence, our strategies are a representative selection. In the remaining thesis, we describe feature-wise measurement, explain how to realize it, evaluate its effort and accuracy, and propose a solution to find a sweet-spot between measurement effort and prediction accuracy.

5.5. Related Work

In recent years, researchers developed different approaches to analyze variable software, especially in the area of software product lines, such as type checks [Apel et al., 2010a, Kästner et al., 2012], theorem proving [Bruns et al., 2011], and model checks [Kishi and Noda, 2006, Apel et al., 2011]. Thüm et al. [2012] provide an overview of these analyses. The main contribution, however, is that they give a classification about how these analyses are applied to software product lines. That is, should we analyze all variants, specific variants, or should we analyze only the features instead of programs? For this reason, Thüm and others suggest three categories of analysis strategies: *product-based*, *family-based*, and *feature-based*.

In a parallel line of research, we developed our strategies to measure non-functional properties of customizable programs. We call our strategies *direct measurement*, which refers to a feature-based analysis, *family-based measurement*, which corresponds to a family-based analysis, and *delta-wise measurement*, which is a form of an optimized product-based analysis. Hence, both strategies are related, but we have specific goals, such as generality, efficiency, and accuracy of predictions, in mind, whereas Thüm discusses more general properties (e.g., scalability) that are applicable for all kinds of analyses and not specific for non-functional properties.

5.6. Summary

We defined five goals (measurement effort, prediction accuracy, applicability to black-box programs, generality to measurable non-functional properties, and usage

in realistic environments) that a measurement strategy in the context of customizable programs has to satisfy. We outlined three strategies based on analysis strategies of software product lines to measure non-functional properties of features.

We presented the direct measurement strategy, which measures non-functional properties of a feature's implementation unit. That is, the approach is applicable only if we have access to the source code and if the mapping between features and implementation units is known (i.e., it prohibits a black-box approach).

The second strategy is family-based measurement. We showed that the complete customizable program is measured once and afterwards, the measurement results are mapped back to individual features. This strategy also requires the presence of source code or the ability to instrumentalize the code with variability information, which makes it possible to map measurements to individual features. However also for this strategy, a black-box approach is not possible.

Finally, we outlined the concept of feature-wise measurement. We measure produced variants, which maximizes applicability (e.g., we can measure black-box programs). To this end, we compute the difference of measured non-functional properties of two variants that differ only in a single feature to determine the influence of the corresponding feature on a non-functional property. We compared all three strategies regarding our defined goals. We concluded that feature-wise measurements represent the best trade-off between these goals.

6. Feature-Wise Measurement

This chapter shares material with the following papers:

- "Scalable Prediction of Non-Functional Properties in Software Product Lines." in *SPLC'11* [Siegmund et al., 2011],
- "Predicting Performance via Automated Feature-Interaction Detection." in *ICSE'12* [Siegmund et al., 2012a], and
- "Scalable Prediction of Non-Functional Properties in Software Product Lines: Footprint and Memory Consumption." in *IST'12* [Siegmund et al., 2012b]

In this chapter, we present our approach to measure a feature's influence on non-functional properties. We concentrate only on feature terms in our prediction model and call this approach *feature-wise measurement*. We omit measuring feature-interaction terms, because we want to find out how accurate our predictions are with as few measurements as possible. That is, we define $a\#b = \emptyset$, such that $\Pi(a\#b) = 0$. Hence, we extend a configuration for feature-wise measurement as follows: $\Pi(a \times b) = \Pi(a) \cdot \Pi(b) \cdot \Pi(a\#b) = \Pi(a) \cdot \Pi(b) \cdot \Pi(\emptyset) = \Pi(a) \cdot \Pi(b)$. Furthermore, we want to identify the effects of feature interactions for different non-functional properties. An important aspect of our approach is to determine which configurations need to be measured. Hence, we explain how to identify these configurations and how to compute the influence of all feature terms. We evaluate prediction accuracy and measurement effort with several experiments by means of three non-functional properties: performance, footprint (binary size), and main-memory consumption.

6.1. Feature-Wise Measurement

The main idea of feature-wise measurement is simple: measure two variants that differ in a single feature and interpret the difference of these measurements as the influence of the differing feature on non-functional properties. This simplicity is the key to conquer the exponential complexity of measurements and to achieve a wide applicability in terms of supported non-functional properties and customization techniques. Since we measure actually generated variants, we are not limited to specific implementation techniques, and require neither domain knowledge nor the source code of the customizable program. With few measurements (linear complexity in terms of number of features), we can predict non-functional properties of all configurations (exponential in the number of features). Although the approach is simple, it yields surprisingly good results, as we show in our evaluation.

We explain in Chapters 7 and 8, how feature interactions influence prediction accuracy and how interactions can be detected and quantified with respect to their influence on non-functional properties.

6.1.1. Computing Feature Terms

In this section, we describe how to determine the feature terms. That is, the configurations that must be measured to quantify the influence of each feature on a non-functional property. First, we describe the general concept of our approach. Next, we explain why and how we build a set of equations from the measurements. Afterwards, we explain algorithms necessary to extract the approximations of each feature's influence on a non-functional property from a (minimal) set of variants. We start with a description of our notation that we use to express configurations, measurements, and approximations of a feature's influence on a property. For illustration, we use the property footprint in the examples.

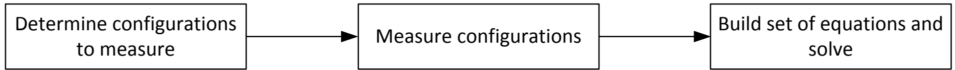


Figure 6.1.: Process of computing all features' terms for the prediction model.

The overall process is as follows: First, we determine configurations that we have to measure to approximate the influence of each feature on a non-functional property (as depicted in Figure 6.1). Second, we measure these configurations. Third, we build a set of equations from these measurements. Each feature represents a variable in these equations. A single equation is therefore the set of selected features that equals to the measurement result. By solving this set of equations, we compute for each feature the influence on the measured non-functional property. Next, we describe this process in detail.

We have to keep in mind that all configurations must be valid, because we must be able to generate the corresponding variants to measure them. Thus, we use feature models as a base to determine necessary configurations. In Figure 6.2, we show a feature model of a customizable DBMS, which we use as example. If we want to approximate the influence of feature *Encryption*, we have to determine two configurations that differ in the presence of feature *Encryption* only. We select and measure configurations $C_1 = \{Base, Encryption, RSA\}$, where the measured footprint of C_1 is $\Pi_F(C_1) = 730\,\text{KB}$, and $C_2 = \{Base\}$, where $\Pi_F(C_2) = 420\,\text{KB}$. Note that we have to select feature *RSA* in C_1, too, because it is a mandatory child feature of *Encryption* and must therefore always be selected in combination with *Encryption*. From these two configurations, we determine the delta, which is 310 KB and interpret this as the influence of features *Encryption* and *RSA* on footprint: $\Pi_F(Encryption \times RSA) = 310\,\text{KB}$. Since there is no variant that allows us to select *Encryption* without

RSA (consequently, there is no need to measure these features individually), we successfully determined *Encryption*'s influence on footprint.

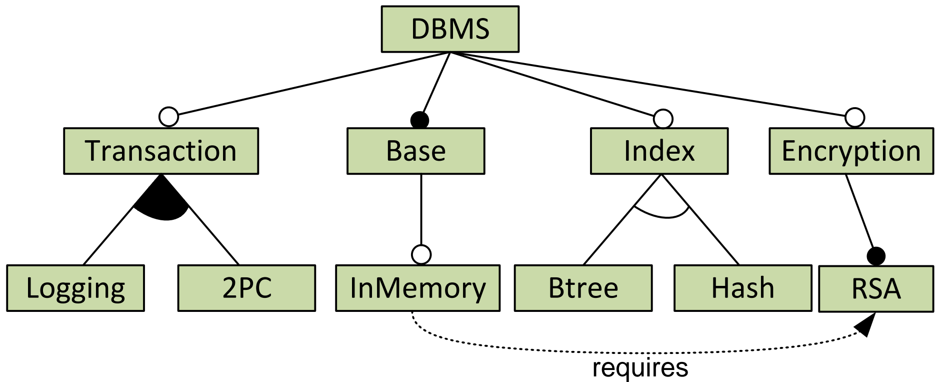


Figure 6.2.: Sample customizable DBMS program. The root denotes the concept. 2PC: two phase commit protocol.

Although the main idea is to determine only two configurations per feature, we already observed in the above example that this is not always possible. Feature *RSA* cannot be measured in isolation, because we cannot measure *Encryption*'s influence without *RSA*. This and other constraints are defined in feature models either by common relationships or by propositional formulas. In the following, we show how to determine necessary configurations for the most common relationships. This is sufficient, because it is possible to translate propositional formulas into feature models with these relationships, as shown by Czarnecki and Wasowski [2007]. Furthermore, we show in Chapter 8 how to completely abstract from feature models and their relationships – for now, we keep them to describe the basics of our approach.

In Table 6.1, we summarize all configurations that we need to determine all feature terms of the DBMS example (Figure 6.2) for the property footprint Π_F (for illustration purpose). To distinguish the two configurations we need to measure for each feature, we call the configuration with the feature to be approximated *feature variant* and the configuration without this feature *delta variant*.

Note that although we show in the following individual equations that compute the influence of a feature on a non-functional property, we do not perform this computation in isolation per feature, but collect all measurements to build a set of equations. As said before, in an equation, each feature is represented by a single variable and the sum of all variables equals to the measurement result. Then, we solve this set of equations to approximate the influence of all features in one step. This has several benefits, as we discuss in Section 6.1.2.

(1) Initial Feature Set. There are customizable programs that always require selecting some features to derive a valid configuration (e.g., we may have to choose

Feature	Feature Variant	Π_F	Delta Variant	Π_F
Base	P_1 ={Base}	420	∅	0
Encryption	P_2 ={Base, Encryption, RSA}	730	P_1 ={Base}	420
RSA	P_2 ={Base, Encryption, RSA}	730	P_2 ={Base}	420
Index	P_4 ={Base, Index, Hash}	570	P_1 ={Base}	420
Btree	P_3 ={Base, Index, Btree}	740	P_1 ={Base}	420
Hash	P_4 ={Base, Index, Hash}	570	P_1 ={Base}	420
InMemory	P_5 ={Base, InMemory, Encryption, RSA}	610	P_2 ={Base, Encryption, RSA}	730
Transactions	P_6 ={Base, Transactions, Logging}	845	P_1 ={Base}	420
Logging	P_6 ={Base, Transactions, Logging, 2PC}	995	P_7 ={Base, Transactions, 2PC}	885
2PC	P_6 ={Base, Transactions, Logging, 2PC}		P_8 ={Base, Transactions, Logging}	845

Table 6.1.: Set of configurations to determine feature terms. All measured values are in KB.

between alternative features). In such a case, we cannot determine a configuration that differs only in these features, because we always have to select them. To overcome this problem, our first task is to measure the influence of an *initial feature set* on a non-functional property. This initial feature set acts as the base configuration for all features that have no parent feature. Since there is no variant with fewer features, the minimal variant is the empty set, for which each non-functional property is zero. Hence, we interpret the influence of the initial feature set on a non-functional property as the measured value of the corresponding variant. For example, feature *Base* must always be selected in our sample program (see Figure 6.2):

Feature	Feature Variant	Delta Variant	Feature Term	Result
Base	$\Pi_F(P_1) = 420\,\text{KB}$	$\Pi_F(\emptyset) = 0\,\text{KB}$	$\Pi_F(Base) = 420\,\text{KB}$	Base 420KB

The corresponding equation is:

$$\begin{aligned}\Pi_F(Base) &= \Pi_F(P_1) - \Pi_F(\emptyset)\\ &= 420\,\text{KB} - 0\,\text{KB}\\ &= 420\,\text{KB}\end{aligned}$$

(2) Optional. In an optional relationship, it is not required to select the child feature. Hence, we generate two variants: The first contains only the parent feature.[1] And, in the second, we additionally include the optional child feature. In our sample program, feature *Encryption* is an optional feature. Since it has no parent feature,

[1] Of course, we have to include all necessary features to derive a valid variant, e.g., all mandatory features.

the initial feature set is considered as the root feature and acts as the parent feature. Based on the computed set of configurations in Table 6.1, we measure the following variants:

Feature	Feature Variant	Delta Variant	Feature Term	Result
Encryption	$\Pi_F(P_2) = 730\,\text{KB}$	$\Pi_F(P_1) = 420\,\text{KB}$	$\Pi_F(Enc) = 310\,\text{KB}$	Encryption 310KB

With these measurements, we compute the term of feature *Encryption*:[2]

$$\begin{aligned}\Pi_F(\mathit{Encryption}) &= \Pi_F(P_2) - \Pi_F(P_1) \\ &= 730\,\text{KB} - 420\,\text{KB} \\ &= 310\,\text{KB} \\ &= \Pi_F(\mathit{Encryption} \times \mathit{RSA}) // \; \textit{per defintion}\end{aligned}$$

where P_1 represents the variant that we measure for *Encryption*'s parent feature. Note that $\Pi_F(\mathit{Encryption})$ does not contain the influence of feature *Encryption* solely, but also the influence of feature *RSA* and their feature interactions. This is because feature *RSA* is a mandatory child feature of *Encryption* and thus must always be selected in combination. We explain this behavior next.

(3) Mandatory. A mandatory relationship enforces that, whenever the parent feature is selected, we must also select its child feature. As a consequence, we cannot measure the parent feature's influence on a property without measuring the influence also of all mandatory children. Hence, we set the value of the child feature to zero (i.e., $\Pi(\mathit{Child}) = 0$) and the value of the parent term to computed delta (e.g., 310 KB for feature *Encryption*).

When working with feature models, a preprocessing step often takes place, which builds *atomic feature sets* [Segura, 2008]. An atomic feature set is a set of features, which are always selected in combination. That is, we compose multiple features that must always be selected in combination into a single feature. This reduction of the number of features has benefits for feature-model analysis, because we have to consider less features without sacrificing any information or variability [Thüm et al., 2011]. In our case, features *Encryption* and *RSA* are always selected in combination. Hence, they build an atomic set not only regarding their code base, but also regarding their influence on non-functional properties. When a stakeholder selects the parent feature of a mandatory relationship during configuration, we show

[2] Again, we omit all feature-interaction terms, because in the feature-wise measurement, we do not consider feature interactions. We explain how to incorporate interactions in Chapter 7.

already the aggregated value of both features. This way, stakeholders can easily see the implications of a feature selection, because they usually select features starting from the root node.

Feature	Feature Variant	Delta Variant	Feature Term	Result
RSA	N/A	N/A	$\Pi_F(RSA) = 0$ by definition	RSA 0KB

We define the term for a mandatory relationship as follows:

$$\Pi_F(RSA) \quad = \quad 0 \; // \; \textit{atomic feature set}$$

Although we measured the influence of feature *RSA* already with $\Pi_F(Enc)$, we do not assign values to both features in the product-line model. Instead, we assign always the value zero to a mandatory feature that is not in the initial feature set (i.e., not a root feature), because we approximated the influence of this feature when measuring its parent feature.

(4) Alternative. In an alternative relationship, we cannot select the parent feature of the relationship individually, but measure its value always in combination with its child features. Since we cannot select multiple child features, there is no need to quantify the influence of the parent feature. Any valid configuration that includes a child feature contains also the parent feature. Hence, this is exactly the same case as for mandatory features. The only difference is that we cannot assign the influence to the parent feature, because depending on which child feature is selected, this influence changes.

There are at least two ways to store the influence of feature *Index* and its children *Hash* and *Btree* in the product-line model. In the first variant, we can set $\Pi_F(Index) = 0$ and approximate only the influence of its child features (i.e., we store only $\Pi_F(Hash)$ and $\Pi_F(Btree)$). In the second variant, we set the smallest measured value of the alternative features as the influence of feature *Index*, because selecting *Index* has at least the smallest measured influence on non-functional properties. Furthermore, we set the influences of the alternative features to the difference of the smallest influence and their actually measured influence.

From the above description, we see that there is a unique solution, but lots of different ways to represent them. That is, we can build different, sometimes equivalent, prediction models from the same measurement data. This may lead to prediction errors or at least different prediction outcomes. This is why we give our definitions and describe later their corresponding realization for which we experienced the most accurate predictions.

Feature	Feature Variant	Delta Variant	Feature Term	Result
Index	N/A	N/A	$\Pi_F(\mathit{Index}) = 0\,\text{KB}$ by definition	Index 0KB
Btree	$\Pi_F(P_3) = 740\,\text{KB}$	$\Pi_F(P_1) = 420\,\text{KB}$	$\Pi_F(\mathit{Btree}) = 320\,\text{KB}$	Btree 320KB
Hash	$\Pi_F(P_4) = 570\,\text{KB}$	$\Pi_F(P_1) = 420\,\text{KB}$	$\Pi_F(\mathit{Hash}) = 150\,\text{KB}$	Hash 150KB

In our example, we require two variants $P_3 =\{\mathit{Base, Index, Btree}\}$ and $P_4 =\{\mathit{Base, Index, Hash}\}$. To use the first method to store the influences of alternative features, we define the following equations:

$$\begin{aligned}
\Pi_F(\mathit{Btree}) &= \Pi_F(P_3) - \Pi_F(P_1)\\
&= 740\,\text{KB} - 420\,\text{KB}\\
&= 220\,\text{KB}\\
\Pi_F(\mathit{Hash}) &= \Pi_F(P_4) - \Pi_F(P_1)\\
&= 570\,\text{KB} - 420\,\text{KB}\\
&= 150\,\text{KB}\\
\Pi_F(\mathit{Index}) &= 0\,\text{KB} \ // \ \mathit{by\ definition}
\end{aligned}$$

If we decide for the second method of storing the influences of alternatives, we additionally use the following equations after computing the terms for features *Btree* and *Hash*:

$$\begin{aligned}
\Pi_F(\mathit{Index}) &= \text{Min}(\Pi_F(\mathit{Hash}), \Pi_F(\mathit{Btree}))\\
&= \text{Min}(740\,\text{KB}, 570\,\text{KB}) - 420\,\text{KB}\\
&= 570\,\text{KB} - 420\,\text{KB}\\
&= 150\,\text{KB}\\
\Pi_F(\mathit{Btree}) &= \Pi_F(\mathit{Btree}) - \Pi_F(\mathit{Index})\\
&= 740\,\text{KB} - 570\,\text{KB}\\
&= 170\,\text{KB}\\
\Pi_F(\mathit{Hash}) &= \Pi_F(\mathit{Hash}) - \Pi_F(\mathit{Index})\\
&= 570\,\text{KB} - 570\,\text{KB}\\
&= 0\,\text{KB}
\end{aligned}$$

(5) OR. In contrast to an alternative relationship, in an OR relationship, we can select multiple child features. This raises the problem that we have to quantify

the influence of the parent feature to predict configurations accurately when more than one child feature is selected. For instance, if we approximate the influence of feature *Logging*, we measure not only the footprint of *Logging*. Π_F*Logging*, but also feature *Transaction*'s footprint Π_F*Transaction*. Approximating *2PC*'s influence on footprint is similar. We measure *2PC*'s footprint, but also *Transaction*'s footprint. When we predict the footprint of a configuration with features *Logging* and *2PC*, we take the footprint of feature *Transaction* into account twice.

Feature	Feature Variant	Delta Variant	Feature Term	Result
Transaction	$\Pi_F(P_6) = 995\,\text{KB}$	$\Pi_F(P_1) = 420\,\text{KB}$	$\Pi_F(\mathit{Txn}) = 315\,\text{KB}$	Transaction 315KB
Logging	$\Pi_F(P_6) = 995\,\text{KB}$	$\Pi_F(P_7) = 885\,\text{KB}$	$\Pi_F(\mathit{Log}) = 110\,\text{KB}$	Logging 110KB
2PC	$\Pi_F(P_6) = 995\,\text{KB}$	$\Pi_F(P_8) = 845\,\text{KB}$	$\Pi_F(\mathit{2PC}) = 150\,\text{KB}$	2PC 150KB

We overcome the above problem by additionally measuring one configuration in which two child features are present (P_6). The approach is as follows: We determine the influence of each OR feature by determining two configurations that differ only in the current feature. For instance, configurations P_6 and P_7 differ only in the presence of feature *Logging*. Hence, we interpret the delta of the corresponding measured footprint as *Logging*'s footprint. We yield the following equations:

$$
\begin{aligned}
\Pi_F(\mathit{Logging}) &= \Pi_F(P_6) - \Pi_F(P_7) \\
&= 995\,\text{KB} - 885\,\text{KB} \\
&= 110\,\text{KB} \\
\Pi_F(\mathit{2PC}) &= \Pi_F(P_6) - \Pi_F(P_8) \\
&= 995\,\text{KB} - 845\,\text{KB} \\
&= 150\,\text{KB}
\end{aligned}
$$

To approximate the impact of the parent feature of an OR relationship, we have several possibilities. We can subtract the already approximated influence of features *Logging* and *Base* from measurement $P_8 = (\{\mathit{Base}, \mathit{Transaction}, \mathit{Logging}\})$: $\Pi_F(\mathit{Transaction}) = \Pi_F(P_8) - \Pi_F(\mathit{Base}) - \Pi_F(\mathit{Logging})$. Another possibility is to not use any intermediate result, but calculate *Transaction*'s footprint directly from measurements:

$$
\begin{aligned}
\Pi_F(\mathit{Transaction}) &= \Pi_F(P_7) + \Pi_F(P_8) - \Pi_F(P_1) - \Pi_F(P_6) \\
&= 885\,\text{KB} + 845\,\text{KB} - 420\,\text{KB} - 995\,\text{KB} \\
&= 315\,\text{KB}
\end{aligned}
$$

Here, we use the measured configurations for each OR feature ($\Pi_F(P_7)$ and $\Pi_F(P_8)$) and subtract the measurement of the configuration without *Transaction* ($\Pi_F(P_1)$) and the measurement of the configuration in which both OR features are present ($\Pi_F(P_6)$). This way, we add the influence of *Transaction* twice and subtract it only once. Hence, the result of this computation is the influence of feature *Transaction* on footprint.

(6) Requires. Finally, we also consider cross-tree constraints in the feature model. The *excludes* constraint does not change the computation of a feature's non-functional properties, because it restricts only the number of features, and we already measure a variant with a minimal number of features. In contrast, the requires constraint prohibits the measurement of a single feature. For example, we cannot measure feature *InMemory* without feature *RSA*. In such a case, our approach is to first measure the variant that includes the target of the requires constraint (i.e., feature *RSA* with configuration P_2 is the target of feature *InMemory*). Then, we measure the variant that includes both features of the requires constraint P_5 ={*Base, InMemory, Encryption, RSA*}. Therefore, the delta of both measurements represents the influence of feature *InMemory*.

Feature	Feature Variant	Delta Variant	Feature Term	Result
InMemory	$\Pi_F(P_5) = 610\,\text{KB}$	$\Pi_F(P_2) = 730\,\text{KB}$	$\Pi_F(Mem) = -120\,\text{KB}$	InMemory -120KB

$$
\begin{aligned}
\Pi_F(InMemory) &= \Pi_F(P_5) - \Pi_F(P_2) \\
&= 610\,\text{KB} - 730\,\text{KB} \\
&= -120\,\text{KB}
\end{aligned}
$$

We visualize the result of our computations in Figure 6.3. We are aware of that there might be cycles in a feature model, such that each feature of the cycle cannot be measured without any other feature of the cycle. Hence, approximations of individual features in a cycle cannot be computed, but this is not necessary. Cycles can both be removed by computing the corresponding atomic feature set or by assigning the measured value to one feature of the cycle and set all others to zero.

6.1.2. Algorithms and Realization

To realize feature-wise measurement, we need a constraint-satisfaction-problem solver that allows us to define objective functions. The solver computes two configurations for each feature and determines configurations with a minimal number

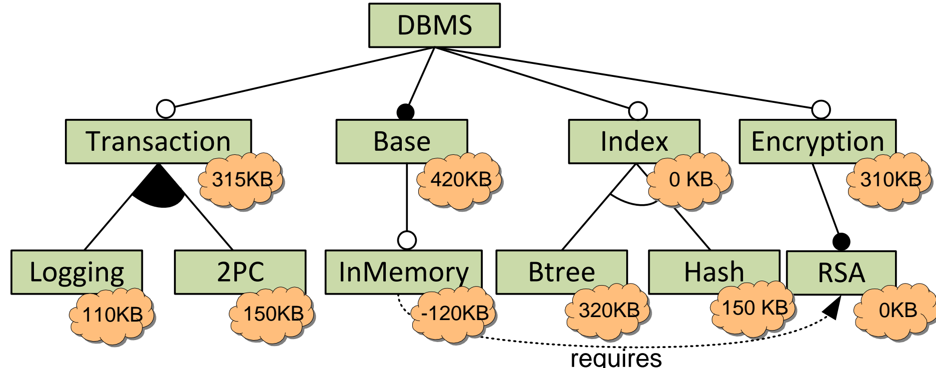


Figure 6.3.: Feature model after approximating the influence of each feature on footprint.

of features, which is the optimization goal and the reason why we cannot use satisfiability solvers. Furthermore, we use a linear-equation solver to compute all feature terms. In the following, we describe our implementations, which we used in our tool *SPL Conqueror*.[3]

Computing the Set of Configurations for Measurement. Measurements can be time consuming and expensive. This is the reason why we aim at further reducing the number of necessary measurements from $2n$ to $n+1$ by reusing already executed measurements. To reach this goal, we use the hierarchical structure of feature models that allows us to reuse variants already defined for the parent feature.

Since a feature model has a hierarchical form, every feature, but the root, has a parent feature. In case we have to select multiple features to obtain a minimal valid configuration, we use the initial feature set as the root node. Beginning with the root node, we traverse the feature tree and add for each feature *a single* configuration to the configuration set. For example, when reaching feature *Encryption* of our sample program, we add configuration $P_2 = \{Base, Encryption, RSA\}$. We add also feature *RSA*, because features *Encryption* and *RSA* are in an atomic feature set. The reused configuration for this feature is the initial feature set. Hence, each newly determined configuration can use the previously defined configuration (e.g., the one for the parent) to compute the difference of its non-functional properties. An exception for this rule is the OR relationship, in which we have to measure an additional configuration to determine the influence of the parent feature of the OR group (as we explained before).

Although the feature model of Figure 6.3 has ten features,[4] we need to measure only eight configurations, because we reuse already measured configurations and

[3]http://fosd.de/SPLConqueror
[4]The root node does not represent a feature.

we can omit measurements due to the mandatory feature *RSA* and the alternative relationship between *Btree* and *Hash*.

In Figure 6.4, we show our corresponding implementation to compute the set of configurations for measurement. The input is a feature model that consists of a list of features and a list of constraints in propositional formulas. The output is a list of configurations, which have to be measured. After initializing all variables, we transform the feature model by computing atomic feature sets (Line 4). This removes mandatory features and can remove circles. In Line 5, we compute the initial feature set, if possible. That is, we check whether we always have to select features to obtain a valid configuration. Since all features of the initial feature set will have the same value (because we cannot measure them individually), we add only a single configuration to the set of configurations to be measured (Line 6).

In Line 9, we begin traversing the feature model.[5] As said before, we handle OR groups differently to compute the feature terms of the OR group's parent feature (Lines 11–14). That is, we add a configuration for the parent of an OR group that contains two additional child features (Line 13 encoded with `f.children`). Of course, we check whether these features can be selected in combination, but this can be easily done using a satisfiability solver. Hence, we omitted this part for brevity.

We compute a valid configuration for a given feature f using a constraints-satisfaction-problem solver. In Lines 13 and 15, we make calls to an intermediate function that actually uses the solver. We explain the details of this function next. The result of function `getConfig(..)` is a valid configuration that contains the given features (first argument), has a minimal number of features, and contains – if possible – the initial feature set (second argument) to enable reuse of configurations. For brevity, we omitted the part at which we translate the first argument of function `getConfig(..)` (in Line 13 and 15) to a list of features. We add the result configuration to the list `configurationSet` and continue with the next feature. After all features have been processed, we remove all duplicates from the list to obtain a set of configurations. Afterwards, we generate the corresponding variants of the configurations.

Function `CSP.getConfig(..)` is responsible for translating a feature model into propositional formula and to use a constraint-satisfaction-problem solver, such that the solver computes a minimal valid configuration. We show the corresponding implementation in Figure 6.5. The function takes a partial configuration (i.e., a list of required features, Line 1), the initial feature set (Line 2), and the feature model (Line 3) as an input. We need the initial feature set to keep a consistent base configuration for all our approximations, which eases a later detection of feature interactions. Furthermore, we need the feature model to translate all constraints of the feature

[5] We do not need a specific traversing order to measure only $n+1$ variants, because we automatically reach every position in the model in which we add a configuration for the parent of the current feature to the configuration set.

```
1   Data: FeatureModel fm
2   Result: List<Configuration> configurationSet
3
4   fm = buildAtomicFeatureSet(fm);
5   List<Feature> initialFeatureSet = getInitialFeatureSet(fm);
6   configurationSet.Add(initialFeatureSet);
7
8   //Run through all features and add a corresponding configuration
9   foreach(Feature f in fm){
10   if(!processedFeatures.Contains(f) && !initialFeatureSet.Contains(f)){
11     if(f.is_OR_Group_Parent()){
12      //add configuration that includes f and two child features
13      configurationSet.Add(CSP.getConfig(f + f.children, initialFeatureSet,fm));
14     }
15    configurationSet.Add(CSP.getConfig(f, initialFeatureSet, fm));
16    processedFeatures.Add(f)\;
17    }
18   }
19   return configurationSet;
```

Figure 6.4.: Algorithm to compute set of configurations that have to be measured to compute all feature terms.

model to boolean terms, which we can put into the solver, so that we derive a valid variant.

Finally, we use a solver with the ability to optimize a solution, because we want to minimize the number of features in each configuration. The reason to keep the number of features minimal is twofold: First, we enable reuse of already measured variants. Second, a minimal number of features also minimizes the number of feature-interaction terms in a variant (see Equation 4.1). Since feature-interaction terms can change the approximated influence of a feature, we aim to exclude this threat to accuracy.

To minimize the number of features in a configuration, we use a solver that takes a partial feature selection and outputs a valid feature selection with a minimal number of features. To enable the minimization, we assign an artificial cost of 1 to each feature as a penalty for the optimizer to not include more features than needed (Line 14 in Algorithm 6.5). Hence, we translated the problem of minimizing the number of features to minimizing the cost of a feature selection. Furthermore, we assign features of the initial feature set a value of -1000 as a reward to include these features in a configuration (Line 12). This has the benefit that we encourage the solver to reuse features that were already measured.

Furthermore, to enforce that the solution of the solver is a configuration in which the required features are present, we define additional constraints that state that a valid solution implies the presence of the required features (the corresponding boolean representation is set to true in Line 24). Finally, we map the result of the solver from its boolean representation back to a feature selection (Line 21).

```
1  Data: Configuration RequiredFeatures
2  Data: Configuration initialFeatureSet
3  Data: FeatureModel fm
4  Result: Configuration configToMeasure
5
6  //Build CSP model, map features to CSP variables,
7  //build cost map to get minimal configuration
8  foreach(Feature f in fm){
9   initialize boolean term t that corresponds to f;
10  add t to CSP solver;
11  if(f in InitialFeatureSet)
12    set cost of t = -1000;
13  else
14   set cost of t = 1;
15
16  get all constraints of f in fm;
17  build boolean terms of constraints;
18  add these terms to CSP model;
19 }
20
21 //Set required features to true in CSP model
22 foreach(Feature f in RequiredFeatures){
23 get boolean term t of feature f;
24 add implication to CSP model: true implies t;
25 }
26
27 //Start optimization and obtain configuration
28 CSP.minimize();
29 foreach(Term t in CSP.Solution){
30  map boolean term t of the solution back to feature f;
31  add feature f to configToMeasure;
32 }
33 return configToMeasure;
```

Figure 6.5.: Algorithm to obtain a valid minimal configuration from a CSP solver for a given partial feature selection.

Computing Feature Terms From Measurements. In the previous paragraph, we explained how to calculate the configurations that we have to measure. Here, we explain how to compute feature terms from these measurements. As said before, we use a set of equations rather than computing the influences separately. The benefits of using such a set of equations are the following: First, measurement errors have a weaker effect on the approximation of a feature's impact, because by solving all equations at once, measurement errors distribute over all measurements. Second, we can already weaken the effect of feature interactions by over-specifying the set of equations. That is, we can use more equations than necessary to approximate the features' influences (e.g., because a feature is present in multiple configurations). To this end, we can define additional variables (in the following called error vector) that account for inconsistencies in solving the equations. The benefits become clear when we describe the realization.

The first task is to build the set of equations. It bases on the set of previously measured configurations. Each configuration maps to a single equation, in which

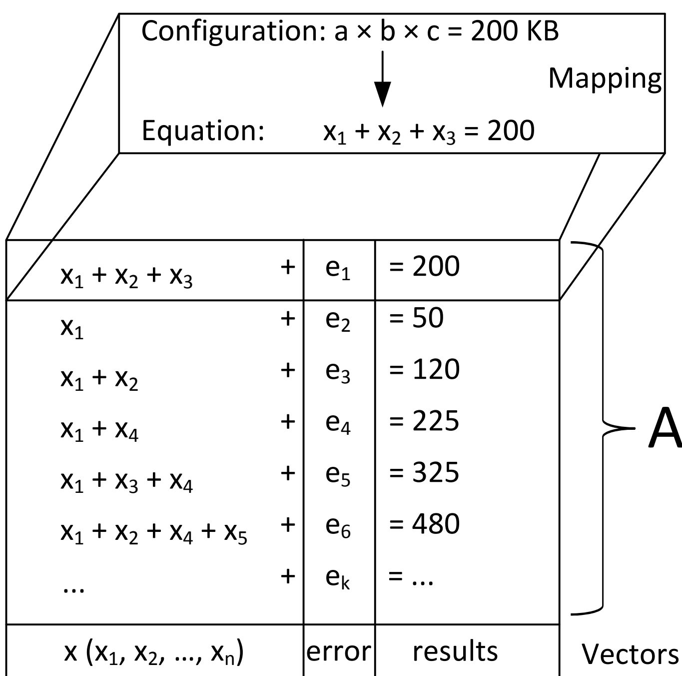


Figure 6.6.: Building the set of equations based on measurements.

features of a configuration are represented by variables (see top of Figure 6.6). That is, we map each feature of a customizable program to a variable x_i and build a vector $x = (x_1, ..., x_n)$, where n is the number of all features. The left part of each equation represents the corresponding feature selection $x_1 \times x_2 \approx x_1 + x_2$. The right part of each equation represents the corresponding measurement result $\Pi(\{x_1 \times x_2\}) = VALUE$. Hence, the mapping is as follows (cf. top of Figure 6.6):

$$\begin{aligned} \Pi(Base \times Encryption \times RSA) &= 730\,KB \\ x_1 + x_2 + x_3 &= 730 \end{aligned}$$

Unfortunately, this mapping is not accurate, because operator $\times$ expands to more than the three variables *Base*, *Encryption*, and *RSA* (i.e., we miss feature-interaction terms). Hence, we may face inaccuracies. To account for interaction and measurement errors, we introduce a variable e_i in each equation, which compensates for these errors. We yield the following equation:

$$x_1 + x_2 + x_3 + e_1 = 730$$

The remaining task is to solve all variables and minimize the error variable at the same time. The underlying technique is a well known *simplex algorithm*. Simplex

allows us not only to solve a set of linear equations, but also minimize the error of the solution. As an input, we need vectors that represent our set of equations. We build a vector *error*, which contains for each configuration the approximation error. Hence, $|error|$ is equal to the number of measured configurations n. Similarly, we build a second vector *results* (with cardinality of n) that stores the measurement results for all configurations (as depicted in Figure 6.6).

Finally, we build a $k \times n$ matrix A that contains for all k measurements the feature selection (i.e., the previously described mapping) and the measured result ($result$ vector). Now, we define the following objective function to compute all feature terms by minimizing the approximation error:

$$\begin{aligned} Minimize &: \quad error \cdot x \\ where & \quad\; Ax = results \mid x_i \geq 0 \end{aligned}$$

Note that although the definition of simplex requires that $results_i \geq 0$ and $x_i \geq 0$, there is a standard way to enable also negative measurements. In short, each variable is split into its positive and negative parts: $x_i = x_{i}_pos - x_{i}_neg$. If a variable is negative, then the negative part is greater than zero and the positive part is zero.

As a result of simplex, we obtain a value for each x_i, which represents the influence of the corresponding feature on a non-functional property. Vector $error$ compensates measurement bias and can be used as a first indicator for possible feature interactions.

Tool Support: SPL Conqueror. We developed the tool *SPL Conqueror*[6] to manage and automate the process of determining and measuring variants and to approximate a feature's influence on non-functional properties [Siegmund et al., 2012c]. The application of SPL Conqueror provides two major benefits compared to a manual approach. First, SPL Conqueror realizes an automated measurement and approximation process that does not require any user interaction (e.g., the measurement process can run over night without monitoring). Second, based on the results of the automated measurement and approximation process, it predicts a variant's non-functional properties almost instantly.

SPL Conqueror maintains a product-line model of the given customizable program. We use SPL Conqueror to determine valid configurations that have to be measured. To support arbitrary programming languages and composition techniques, we abstract from specific implementation techniques and consider a customizable program as a black box. All customizable programs have in common that they require a configuration in a special format to be executed with this given configuration. The measurement process has three steps: (1) generate a configuration in the application-specific format, (2) trigger the generation or execution of the variant, and (3) execute

[6] http://fosd.de/SPLConqueror

a user-defined measurement program, which, in turn, executes the application, measures its non-functional properties, and writes the results in a XML format that can be read by SPL Conqueror.

Each step must be defined in SPL Conqueror, so that the whole measurement process can be automated. SPL Conqueror needs to know in which format a configuration must be generated. For example, for preprocessor-based customization, we generate a `flags.h` file, which contains preprocessor statements (e.g., `#define HAVE_ENCRYPTION` to compile Berkeley DB with encryption support). The remaining task (once per customizable program) is to manually include this `flags.h` file in the compilation process (e.g., in the makefile). We support a wide array of customization techniques, but further techniques can be included:

- **Preprocessor**-based customization is supported via an automated generation of a user-defined header file, which includes the definition of preprocessor flags corresponding to selected features.
- **FeatureHouse** is a language-independent composition tool based on feature-oriented programming [Apel et al., 2009]. It stores configurations in an expression file, in which the selected features are listed.
- **AHEAD** is a composition tool suite for programs and other artifacts based on feature-oriented programming [Batory et al., 2004]. The configuration mechanism is similar to FeatureHouse.
- **FeatureC++** generates C++ programs based on feature-oriented programming [Apel et al., 2005] and uses also expression files with a slightly changed syntax.
- **Configuration files** are used in many programs, such as the Apache web server and the RAR compression library. To use this method with SPL Conqueror, a user specifies the name and path of the configuration file as well as how a selected customization option is specified by the corresponding program. Basically, we define the value for these key-value pairs in the product-line model and generate the according configuration file.
- **Command-line options** represent a common way to customize a program. In this case, triggering the variant production is the process of executing a program with a generated set of command-line parameters. This set of parameters is also derived as key-value pairs from the product-line model. With this technique, we measure only runtime properties.

We implemented all concepts presented in this thesis in SPL Conqueror. Next, we evaluate prediction accuracy and measurement effort of feature-wise measurement.

6.2. Evaluation

Since our approach only predicts non-functional properties and cannot provide precise results, we evaluated accuracy of our approximations with three series of ex-

periments. These experiments help to judge feasibility of feature-wise measurement regarding measurement effort and accuracy. Furthermore, they help identifying for which non-functional property it is important to also identify feature interactions.

The first series of experiments addresses measurement and prediction of property *footprint* (binary size of a program), the second series of experiment concentrates on the *main-memory consumption*, and the third series of experiments targets *performance*. We conducted three experiments to increase external validity of our approach. Furthermore, we want to investigate whether there are commonalities or differences of prediction accuracies among the three non-functional properties. We use the goal question metric (GQM) to evaluate our defined goals and research questions [Basili, 1992]. That is, for each evaluation we state the goal, the according research questions, and the metric to answer the questions.

We demonstrate that our predictions are sufficiently accurate for many real-world scenarios, in which we want to constrain the configuration space or select a nearly-optimal product regarding some non-functional property. We provide a detailed analysis for each program online and here show the aggregated results. Furthermore, we present in the appendix feature models and additional information of all programs used as case studies in this thesis. We refer the interested reader to our Web site for more detailed information and for downloading our tool: `http://fosd.de/SPLConqueror`

6.2.1. Experiments

The goal of our evaluation is to rate the prediction accuracy of feature-wise measurement using the three different non-functional properties: footprint, main-memory consumption, and performance. To this end, we state the following research questions:

- **Q1:** What is the mean error rate of the feature-wise measurement for footprint?
- **Q2:** What is the mean error rate of the feature-wise measurement for main-memory consumption?
- **Q3:** What is the mean error rate of the feature-wise measurement for performance?
- **Q4:** Are there differences in the mean error rates between different non-functional properties?

As a metric for evaluation, we calculate the error rate of our prediction as the relative difference between predicted and actual property: $\frac{|actual - predicted|}{actual} * 100$.

In the following, we describe the parts of experiments that are common for all experiments: experimental design, experimental variables, and analyze procedure. We keep these parts constant to ease comparing error rates for different non-functional

properties. Differences occur only for the experimental material and experimental procedure, which we explain separately for each experiment. We had to change the material, because programs that are customized via command-line parameters or configuration files do not change their footprint. Since we want to measure also programs customized via command-line parameters, we had to select different programs. Furthermore, the measurement of different non-functional properties requires different measurement procedures. For footprint, we must compile a program and measure the size of the generated files. For performance and main-memory consumption, we have to execute a benchmark and measure the run-time behavior. Note that we provide feature models and additional descriptions for all customizable programs in the Appendix.

Experimental Design. Our experimental design consists of two phases: creation of prediction model and evaluation of prediction error rate (see Figure 6.7). The first step is to build the prediction model including all feature terms as described in Section 4.2. To this end, we determine the necessary configurations that we have to measure to compute the influence of each feature. This process follows our given description in Section 6.1.2. The result is a prediction model in which we have quantified all feature terms.

In the second step, we evaluate the error rate of our predictions. To this end, we select the configurations for which we compare our prediction against the actually measured non-functional property. We either measure all configurations of a customizable program or select 100 random variants. The threshold for selecting the random variants is determined by the amount of time it takes to measure all variants. If this is feasible (i.e., all measurements can be done within one week), we measure all configurations; otherwise, we measure 100 variants. We state in the corresponding experiments for which programs we measure 100 variants.

After measurement, we compare the predicted non-functional property against the measured. We calculate a error rate of our prediction as the relative difference between predicted and actual property: $\frac{|actual-predicted|}{actual} * 100$. The metric we use is addition (sum) for all non-functional properties. For evaluation, we compute the arithmetic mean of all error rates and present the distribution of error rates using box plots.

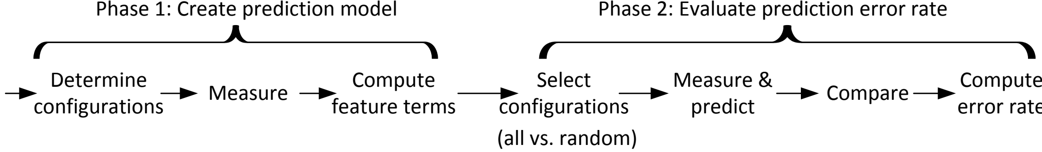


Figure 6.7.: Experimental design with two phases.

Name	Type	Class	Scale Type	Unit	Range/Value
Configuration	indep.	flags, etc.	nominal	N.A.	2^n
Measurement	dep.	measurement.	ratio	KB	≥ 0
Prediction	dep.	prediction	ratio	KB	≥ 0
Error rate	dep.	$\frac{\lvert measurement - prediction \rvert}{measurement} * 100$	ratio	%	≥ 0

Table 6.2.: Description of experiment variables. Indep: independent; dep: dependent; n: number of features.

Variables. The experiment has a single independent variable: configuration. A configuration plays two roles. First, it determines which features are compiled or executed. That is, it influences the measurement result. The measurement result depends on the configuration; it is a dependent variable. Second, a configuration is the basis for our prediction model. From a configuration, we derive which terms we have to determine to compute the prediction for a non-functional property. Hence, prediction also depends on the configuration and is our second dependent variable.

Error rate describes the difference between the predicted and measured non-functional property of a variant. Note, the error rate requires careful interpretation: a base variant or a feature with over proportional influence on the property may distort the error rate. We cannot provide a relative error rate corresponding to some base or minimal variant, because it is not clear what the base or minimal variant is (we would need to measure all variants in the first place).

Analysis Procedure. We analyze the error rate visually using *box plots* [Anderson and Finn, 1996], violin plots, and Quantile-Quantile (Q-Q) plots. A box plot is a graphical method to describe the distribution of data in a comprehensible way. The box of a box plot summarizes 50 % of all data points. The upper and lower borders of a box represent the respective upper and lower quartile of the data distribution. The thick line in the box shows the median value. Finally, whiskers describe the distribution of the remaining data points with the exception of outliers, which are plotted as small dots. We use the statistical tool R to compute box blots.[7] The whiskers extend to the most extreme data point which is no more than 1.5 times the length of the box away from the box. We show in the left part of Figure 6.8 a box plot that illustrates prediction error rates of the customizable program ZipMe. We see that all predictions have an error rate below 1.5 %. The mean prediction error rate is below 0.5 % (thick line).

Although the box plot provides information of how the error rates distribute, we cannot look inside the box. To increase the granularity of showing the error-rate distribution, we can use violin plots. Similar to box plots, violin plots allows us to

[7] `http://www.r-project.org/`

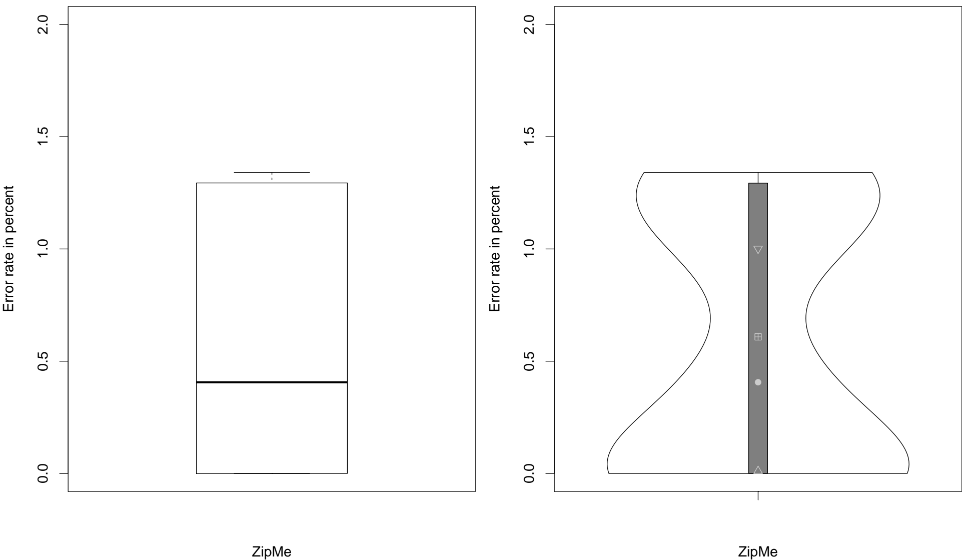


Figure 6.8.: Box plot (left) and violin plot (right) of ZipMe's prediction error rate for footprint.

analyze data distributions visually. The difference is that a violin plot additionally shows a kernel density plot around the box plot. A kernel density plot, in turn, plots a probability density function of a random (input) variable. A well known way to plot such probability functions is a histogram. A kernel density plot, however, has a smooth curve instead of discrete numbers. This smooth curve is placed around a box plot to draw a violin plot. We show markers in a violin plot to visualize the interquartiles and the median of the data.

Right in Figure 6.8, we depict a violin plot using the same data as the box plot. This data set has about 100 observations. Considering the violin plot, most of these observations (ca. 80) are either below 0.5 % error rate or between 1.0 and 1.3 % error rate. Addtionally, we see that there are less observations (ca. 20) with an error rate between 0.5 and 1.0 %. The box plot of Figure 6.8 does not provide this information, because it has a coarser granularity.

A Q-Q plot is often used to compare two ordered data distributions by plotting their quantiles against each other. That is, a point (x_i, y_i) on the plot refers to the i-th data point of the first distribution (x-coordinate) and to the i-th data point of the second distribution (y-coordinate). If both distributions are similar, then x is equal to y and the point lies on the diagonal line $y = x$. We use this plot to compare for the same configurations predicted versus measured properties. For a perfect prediction, all dots lie on the diagonal line. We visualize each configuration as a dot on the plot. In Figure 6.9, we illustrate a Q-Q plot with the ZipMe measurement and prediction data. We see that most of the predictions are correct by lying on the diagonal line. This means that the predicted footprint exactly corresponds to the measured

footprint. For some of the variants, we measured a too small footprint, which is indicated by a dot below the diagonal line.

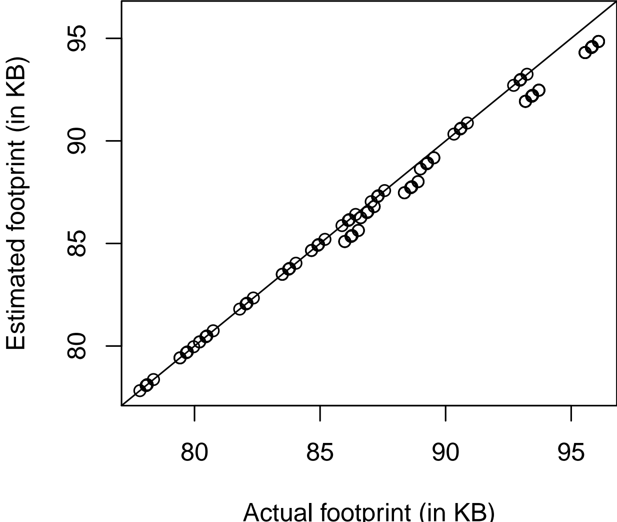


Figure 6.9.: Q-Q plot of prediction and estimated footprint of ZipMe.

In addition to the visual analysis, we compute the mean error rate (arithmetic mean) per customizable program (also noted here as average error rate). To this end, we compute for each sample variant P the error rate, sum them up, and divide the result by the number of measurements:

$$AvgerrorRate = \sum_{i=1}^{n} \frac{|measurement(P_i) - prediction(P_i)|}{measurement(P_i)} * 100 \qquad (6.1)$$

Furthermore, we compute the standard deviation in percent of all measurements per customizable program to quantify the scattering of predictions around the mean error rate.

6.2.2. Footprint

We selected footprint for several reasons:

- Although it may appear trivial, footprint is quite difficult to predict. As for performance, feature interactions can have an immense effect: Features that extend or alter the code of other features can significantly influence the footprint of many other features. Interactions due to shared libraries, nested `#ifdefs` (code is only included when two or more features are selected), or possible compiler optimizations make footprint difficult to predict.

- We can measure footprint quickly and without measurement bias, which is important for a large-scale evaluation with multiple customizable programs as ours. We can easily reproduce values, and we exclude noise and confounding influences, such as system load, which easily can bias benchmarks. In addition, since we need to automate a high number of measurements (not only for variants used to compute feature terms, which a normal user of our approach would do, but, in addition, also for reference variants to compare predicted and actual size), it comes in handy that measuring footprint is quick.

In the following, we describe the experimental design and the data analysis.

Experimental Material. As experimental units for our footprint prediction, we selected nine existing customizable programs with very different characteristics to cover a broad spectrum of scenarios. A prerequisite was that we can customize all programs at compile-time so that we can measure different program sizes. In Table 6.3, we provide an overview of the programs: We selected programs of different sizes (2 500 to 13 million lines of code, 5 to 100 features), implemented with different languages (C, C++, and Java) and different variability mechanisms (conditional compilation and feature-oriented programming), from different domains (e.g., operating systems, database engines, end-user applications), and from different developers (both academic and industrial). Although very different programs are used, the main technical commonality is that we can automatically generate and compile variants for a given feature selection. To set the error rate into perspective for footprint, we provide also the highest and lowest measured value in Table 6.3.

Features are either explicitly given by an already existing feature model (i.e., LinkedList, Prevaylor, ZipMe, PKJab, SensorNetwork, Violet) or derived from documentation. For SQLite and Berkeley, we analyze the documentation to identify features. The document specifies preprocessor flags to turn functions on and off. We extracted this information and created a corresponding feature model. The configuration is given as preprocessor flags to generate the according program.

From Linux, due to the huge configuration space, we considered only a subset of 25 features, selected as representative by a domain expert. The domain expert selected the following features, which cover both modular features, such as drivers, as well as crosscutting features : DEBUG_BUGVERBOSE, INLINE_SPIN_LOCK, OPTIMIZE_INLINING, CC_OPTIMIZE_FOR_SIZE, MODULE_UNLOAD, FRAME_POINTER, MODULE_SRCVERSION, DNOTIFY, INOTIFY_USER, FIRMWARE_IN_KERNEL, SND_VERBOSE_PROCFS, POWER_SUPPLY_DEBUG, PCNET32, NF_CONNTRACK_IPV6, NLS_ISO8859_15, NO_HZ, NET_POLL_CONTROLLER, PRINTK_TIME, SATA_NV, SC520_WDT, KPROBES_SANITY_TEST, I2C_DEBUG_ALGO, CHR_DEV_SCH. Among the 25 features were some features that we knew would change the footprint (as the evaluated non-functional property) of other features (e.g., *OPTIMIZE_INLINING* and *CC_OTPIMIZE_FOR_SIZE* both apply global optimizations).

Program	Domain	Lang.	Techn.	Feat.	Variants	LOC	Size in KB Min*	Size in KB Max*
LinkedList	Component	Java	Comp.	18	492	2 595	4.4	10.5
Prevayler	Database	Java	CC	5	24	4 030	87	169
ZipMe	Compression	Java	Comp.	8	104	4 874	79	99
PKJab	Messenger	Java	Comp.	11	72	5 016	39	161
SensorNetwork	Simulation	C++	Comp.	26	3 240	7 303	19	875
Violet	UML editor	Java	Comp.	100	10^{20}	19 379	6.3	185
Berkeley DB	Database	C	CC	8	256	209 682	1 800	2 740
SQLite	Database	C	CC	85	10^{23}	305 191	166	200
Linux kernel+	OS	C	CC	25	$3{\cdot}10^{24}$	13 005 842	11 245	13 829

* Minimal and maximal size of highly variable programs may not be exact, because we cannot measure all variants. We list the smallest and largest measured value.
\+ We use only a subset of 25 features of the Linux kernel selected by a domain expert. CC: conditional compilation, Comp.: composition approach.

Table 6.3.: Overview of the customizable programs used in the evaluation of footprint prediction.

Experimental Procedure. We compiled all C-based programs with GCC and with -O2 optimization, which performs all compiler optimizations that do not involve a size-speed trade-off. Since footprint measurements are not influenced by the used hardware and we kept the same compiler for all measurements, we could parallelize the footprint measurements on three systems.

Deviations occurred in the experiment for SQLite. It was not possible to measure all variants that are valid with respect to the feature model. In these cases, we run into compilation errors, because of undocumented dependencies between features (compilation flags). However, we could perform all measurements that were necessary to compute all feature terms.

Results. In Table 6.4, we summarize the results of our footprint measurements and predictions for all customizable programs. We observe that the relative number of measurements is low. Especially for customizable programs with a high variability, we need to measure below 1 % of all configurations to achieve an error rate below 1 % on average without Violet. Hence, referring to our research question Q1, our predictions are usually very accurate; a mean error rate of 0.6 % for all programs without Violet and 21.6 % including Violet. We visualize the distribution of the error rates in Figure 6.10 using a violin plot. It shows that the predictions are not scattered but are mostly around a 0–1 % error rate. Berkeley DB and the SensorNetwork simulation exhibit a higher error rate. These are caused by feature interactions at the level of source code. We explain feature interactions in the next chapter.

Program	Measurements		Error Rate in %		
	Absolute	Relative in %	Mean	Std. Dev.	Median
LinkedList	11	2	0.9	0.9	0.51
Prevayler	5	21	0.1	0.1	0.03
ZipMe	8	8	0.6	0.6	0.40
PKJab	8	11	0	0	0.00
SensorNetwork	26	1	0.5	1	0.23
Violet	80	0	186.7	34.4	209.27
Berkeley DB	9	4	1.9	2.2	0.80
SQLite	85	0	0	0	0.03
Linux kernel	25	0	0.4	0.3	0.20

Table 6.4.: Overview of mean prediction error rate and measurement effort for footprint. Relative measurement effort compared to all valid configurations. Std. Dev.: Standard deviation.

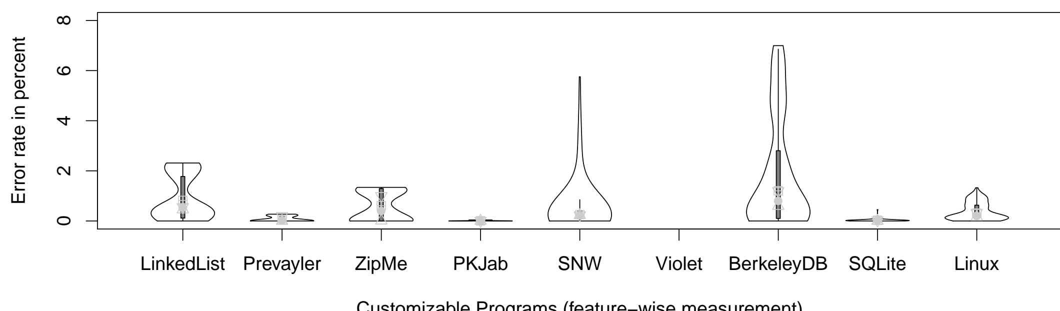


Figure 6.10.: Error rates in percent of all customizable programs for footprint. SNW: SensorNetwork.

An interesting case in the analysis represents Violet. The mean error rate is 186.7 %. We show the corresponding violin plot in Figure 6.11. In our experiments, there is no better prediction than with an error rate of 90 % and no worse prediction that with a error rate of 220 %. Although the prediction error rate seems to be irrational, the reason why we obtain these error rates is simple. We determine the influence of a feature only for a specific base configuration. This delta is the difference in footprint between a variant without the current feature and a variant with this feature. In the case of Violet, we have an n-to-m mapping between features and implementation units. That is, when measuring footprint of a single feature, we measure the footprint of multiple implementation units that are also used by other features. When summing the influences of multiple features to predict a variant's footprint, we sum up the footprint of the same implementation units multiple times. This leads to a large error rate. With this in mind, we can explain why the prediction

error rate has its maximum at around 220 % error rate. This error rate is where the largest number of features shares the same implementation units. We do not obtain an error rate of 0 %, because we selected 100 sample configurations, in which we randomly selected more features than the minimal number of features leading to always an overestimation of the footprint.

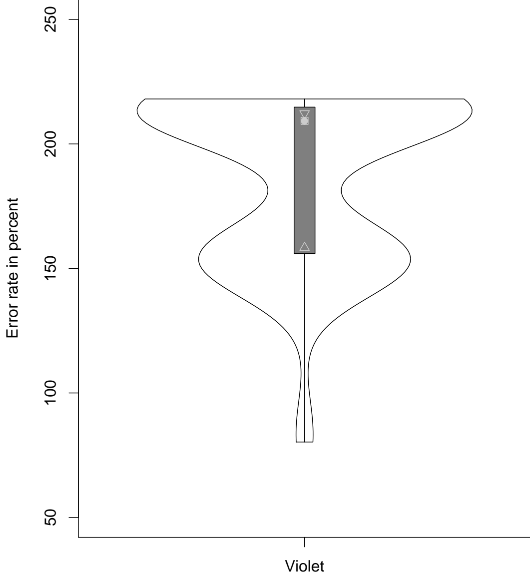


Figure 6.11.: Error-rate distribution of predicting Violet's footprint.

The key results are as follows:

- Predictions with an accuracy of above 99 %, on average, are possible with only a linear number of measurements.
- Prediction accuracy is independent of programming languages, implementation techniques, and domain.
- Complex mappings between feature selection and implementation artifacts cause feature interactions that considerably increase error rates.

6.2.3. Main-Memory Consumption

To increase external validity of feature-wise measurement and to evaluate whether the non-functional property has an influence on prediction accuracy, we performed a second experiment. We selected main-memory consumption for the following reasons:

- Main-memory consumption is a property that emerges at runtime. That is, it is not a static property, such as footprint, but a dynamic one, which can substantially vary depending on which features and how many features are

selected. We expect that predicting main-memory consumption is a challenging task, because we expect many feature interactions.

- Measurement bias for main-memory consumption occurs, but is usually very low. We wanted to explore how accurate predictions are when measurement bias occurs. That is, we evaluate if our approach is still feasible for non-functional properties in which measurements contain noise and may be subject to confounding influences.
- Finally, we chose main-memory consumption, because we are able to automate the measurement procedure for main-memory consumption performing a large number of measurements in a reasonable time.

Experimental Material. To evaluate prediction of main-memory consumption, we initially selected seven existing customizable programs (see Table 6.5). We use fresh programs, because we measure a different characteristic compared to footprint, which requires different measurement techniques and, partly, different customization techniques. For example, footprint is not interesting for programs that are customized via program parameters, because their sizes remain unchanged. Instead, we require a benchmark to measure the runtime behavior, which again is hard to define for a complex system (e.g., a single benchmark to measure the full Linux kernel).

Since benchmarks are often used and important in the database domain, we selected Berkeley DB and SQLite. These systems represent customizable programs, for which variants are generated using conditional compilation. Note that feature models of both programs differ to the models we used for footprint prediction, because we included features that are likely to change memory consumption (e.g., different page sizes) and excluded features that are not executed by the benchmark we used. As additional programs, we selected Curl, LLVM, x264, RAR, and Wget; programs that users can customize via program parameters. We selected these programs to demonstrate that our approach can be applied to black-box programs, for which either no source code or no domain knowledge is available. We deliberately include many different domains, such as video encryption, compilers, and data transfer. Furthermore, these programs are well documented, such that we could easily create feature models for them. Finally, all sample systems are industrial strength real-world applications.

Experimental Procedure. Compared to footprint, we face two additional challenges. First, when measuring a non-functional property at run-time, we have to execute a benchmark application, which we discuss shortly. Second, we face measurement bias; that is, measuring the same variant several times may result in different values. To overcome measurement bias, we measure each variant three to ten times depending on the program. We use this number of repetitions, because of two reasons. First, for all customizable programs (with the exception of RAR), the standard deviation of measuring a single variant is less than 1 % of the arithmetic

Program	Domain	Lang.	Techn.	Feat.	Variants	LOC
Curl	Data transfer	C	CP	13	768	52 341
LLVM	Compiler	C	CP	11	1 024	47 549
x264	Video Encoding	C	CP	16	1 152	45 743
Wget	Data transfer	C	CP	16	5 120	34 880
Berkeley DB	Database	C	CC	18	2 560	209 682
SQLite	Database	C	CC	39	3 932 160	305 191
RAR	Compression	C++	CP	38	500 000	N/A

Table 6.5.: Overview of customizable programs used to predict main-memory consumption. CC:= conditional compilation; CP: command-line parameter.

mean, which is sufficient for our studies. Second, increasing the number of measurements per variant would substantially increase the time needed for this evaluation. Thus, we decided to include more programs in the evaluation and reduce the number of repetitions per measurement instead of reducing the standard deviation to, say 0.1 %, and measuring a single variant hundred times. From these measurements, we compute the arithmetic mean and use it for our subsequent computations.

We use standard benchmarks (if available) for all customizable programs, because self-developed benchmarks would bias the outcome of the measurements and represent a possible threat to construct validity, which we want to minimize. Furthermore, standard benchmarks are created to simulate a common workload that is used in practice, which is our intended goal. To measure the maximum required memory when performing a benchmark, we use the Linux standard program *time*. That is, we execute *time* and pass the program to be measured as an argument. *Time* outputs the maximum used memory when the execution of the passed program has ended.

We used the following benchmarks:

- We use Oracle's standard benchmark for Berkeley DB. Similarly, we execute a benchmark script provided by SQLite to run a typical workload.
- LLVM is a modular compiler infrastructure. For our benchmarks, we use the *opt-tool* that provides different compile-time optimizations. We measure the main-memory LLVM needs to compile its standard test suite in several configurations (e.g., inline functions and combine redundant instructions).
- x264 is a command-line tool to encode video streams into H.264 and MPEG-4 AVC format. We measure the main memory needed to encode the trailer of the cartoon *Sintel*, often used as a standard benchmark (735 MB) for video-encoding projects.
- Curl and Wget are applications to transfer data over the Internet. As we found no standard benchmark, we download Apache's user manual which contains static HTML pages, CSS files and pictures. Due to the manual's folder struc-

	Measurements		Error Rate in %		
Program	Absolute	Relative in %	Mean	Std. Dev.	Median
Curl	11	1	0.4	0.6	0.16
LLVM	11	1	28.8	26.3	14.18
x264	12	1	27.9	47.8	0.03
Wget	14	0	18.6	24.9	9.25
Berkeley DB	15	1	1.4	1.3	1.03
SQLite	26	0	4.4	3.8	3.40

Table 6.6.: Overview of mean prediction error rate and measurement effort for main-memory consumption. Relative measurement effort to all valid configurations. Std. Dev.: Standard deviation.

ture, we can use several features of both programs (e.g., recursive download) to measure significant effects on memory consumption.

We measured main-memory consumption with the following systems, but measure all individual configurations of a customizable program on the same system: AMD Athlon64 2.2GHz, 2GB RAM, Debian GNU/Linux 7; AMD Athlon64 Dual Core @2.0GHz, 2GB RAM, Debian GNU/Linux 7; Intel Core2 Quad @2.4GHz, 8GB RAM, Debian GNU/Linux 7.

Deviations. Early during our measurements of RAR, we identified huge measurement biases. Measurements of the same configuration deviated by 50 to 100 percent. This behavior was present with the Windows and Linux version of RAR. A possible reason may be that the algorithm of RAR is not deterministic. As a result, we were unable to approximate the influence of a single feature and thus discarded this program. For all other customizable programs, the measurement bias is less than one percent.

Results. We summarize the results of the main-memory experiment in Table 6.6. In contrast to footprint, we observe higher error rates, which we partially expected due to the dynamic nature of main-memory consumption. We achieve for the programs Curl, Berkeley DB, and SQLite a low mean error rate of 3 % on average. Considering the measurement bias of 1 %, the true error rate is 2 % on average, which is surprisingly good considering the low number of measurements (below 1 % of all variants). The picture changes for the programs LLVM, x264, and Wget. Here, we observe a mean error rate of 25 %. To answer our research question Q2, the mean error rate of the main-memory experiment over all sample programs is 13.6 %.

We show the error-rate distributions of predicting main-memory consumption in Figure 6.12. Considering LLVM, x264, and Wget, we note that most of our predictions have an error rate of below 19 % percent, but there are many variants which exhibit a high error rate. We recognized a pattern: If certain feature combinations are selected, the prediction becomes inaccurate (because of interactions). This indicates the presence of feature interactions. That is, there are combinations of features that cause a higher usage of main memory than when used in isolation. Hence, features of these programs tend to interact on the level of main-memory consumption, whereas features of programs Curl, Berkeley DB, and SQLite seems to be cohesive in their memory usage. As a result of this analysis, we observe that feature-wise measurement without considering feature interactions can be accurate even for non-functional properties that emerge at run-time if programs have cohesive features. Cohesive means in this case that features implement a self-contained functionality that does not interact with other functions of a program. If this is not the case, predictions become inaccurate and ignoring feature interactions to save measurements does not pay off anymore.

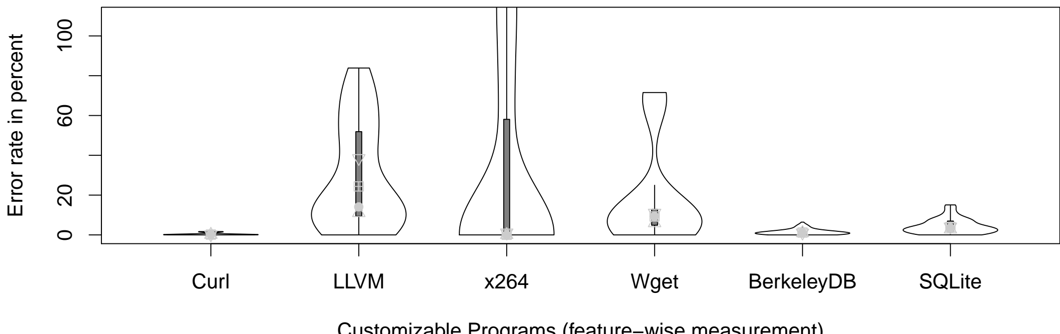


Figure 6.12.: Error rates in percent of all customizable programs for main-memory consumption.

The key results are as follows:

- The mean prediction error rate is 13.6 %, which is significantly worse than for footprint.
- Prediction accuracy heavily differs depending on the non-functional property (i.e., static property vs. runtime property).
- Unlike footprint, prediction error rates heavily differ with different programs.

6.2.4. Performance

We conducted a third experiment to validate our findings of the second experiment. That is, we want to identify whether prediction error rates are worse for run-time (or dynamic) non-functional properties compared to the static property footprint.

Furthermore, we further extend external validity by selecting a property that is relevant in research and practice. We selected performance for the following reasons:

- Performance is one of the most important non-functional properties of programs. For example, performance is critical for business processes, user experience, and real-time requirements. Hence, supporting the accurate prediction of performance underlines the practicability of our approach for real-world scenarios.
- Performance is a challenging property for prediction for two reasons. First, it depends on many factors (e.g., environmental influences, background activity, workloads), such that a prediction may be inaccurate due to unknown influences. Second, performance of a program emerges from the interplay of a program's functions. Thus, we expect many relevant feature interactions, which may make our feature-wise measurement inaccurate when we approximate only the feature terms.
- Similarly to the other properties, we are able to automate the measurement procedure for performance, such that we can execute a large number of measurements in a reasonable time.

Experimental Material. We selected six existing industrial-strength real-world programs (i.e., three customizable programs and three software product lines) with different characteristics to cover a broad spectrum of scenarios (see Table 6.2.4). From these six programs, we kept Berkeley DB, SQLite, LLVM, and x264 from the previous experiment to ease comparison of results. We additionally selected the Apache web server and Berkeley DB Java version to have programs for which performance is relevant and predictions have a practical impact.

The programs are of different sizes (45 000 to 300 000 lines of code, 192 to millions of configurations), implemented in different languages (C, C++, and Java), and configurable with varying mechanisms (such as conditional compilation, configuration files, and command-line options).

The programs we selected have usually under 3 000 configurations. The reason is that we can actually measure *all* configurations of these programs in a reasonable time. Hence, even though it required over 60 days of measurement with multiple computers, we could actually perform the brute-force approach and determine accuracy of our prediction over *all* configurations.

Experimental Procedure. We automated the process of generating programs according to specific configurations and running the benchmark. Since Berkeley DB (C and Java) and SQLite use compile-time configuration, we compiled a new variant for each configuration that includes only the relevant features. For Apache, LLVM,

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

Table 6.7.: Overview of sample programs used in the evaluation of performance.

and x264, we mapped the configuration to command-line parameters. We used five standard desktop computers for the measurements.[8]

We repeated each measurement between 5 to 20 times depending on the measurement bias. It is known that measurement bias can cause false interpretations and are difficult to control [Mytkowicz et al., 2009], especially for performance [Georges et al., 2007]. The width of the 95 % confidence interval is smaller than 10 % of the according means. We use the arithmetic mean of all measurements of a single configuration C as $\Pi(C)$.

We use standard benchmarks, either delivered by the vendor or used in the community of the respective application. We did not develop our own benchmark to avoid bias and uncommon performance behavior caused by flaws in benchmark designs.

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

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

LLVM is a modular compiler infrastructure, which we described previously. For our benchmarks, we use the *opt-tool* that provides different compile-time optimizations.

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

	Measurements		Error Rate in %		
Program	Absolute	Relative in %	Mean	Std. Dev.	Median
Berkeley DB CE	15	0.6	44.1	42.3	49.97
Berkeley DB JE	10	3	17.7	19.6	11.27
Apache	9	4.7	14.9	24.8	5.88
SQLite	26	0	7.8	9.2	6.90
LLVM	11	1.1	7.8	9	7.49
x264	12	1	29.6	22	29.23

Table 6.8.: Overview of mean prediction error rate and measurement effort for performance. Relative measurement effort compared to all valid configurations. Std. Dev.: Standard deviation.

We measure the time LLVM needs to compile its standard test suite (i.e., with different optimizations, such as inline functions and combine redundant instructions enabled). In this case, the workload is the program code from the LLVM test suite that has to be compiled with the enabled optimizations.

For x264, we measured the time needed to encode the video trailer *Sintel* (735 MB).

Results. In Table 6.8, we show the results of our performance predictions. We observe higher error rates compared to the other experiments. The mean error rate over all programs is 20.3 % (Q3). We observed larger measurement bias compared to the main-memory measurement. For example, we observed a difference of 7 % when measuring SQLite's performance of a single variant multiple times. Hence, the prediction for SQLite is quite accurate, because they are nearly in the range of a measurement bias. Nevertheless, our results show that when many interactions occur, feature-wise measurement may become too inaccurate.

The violin plot in Figure 6.13 shows that even for Apache, most of the predictions have an error rate of less than 15 %. Considering the measurement bias of 5 % on average for all customizable programs, most of the predictions have a true error rate of below 10 %, which is suprisingly good for a linear number of measurements. However, Berkeley DB's C and Java version and x264 show that a large number of predictions are evenly distributed with error rates from 0 to 90 %. Here, we observe the true limitations of feature-wise measurement when ignoring feature interactions.

The key results are as follows:

- The mean prediction error rate is 20 %, which is even worse than for main-memory consumption.

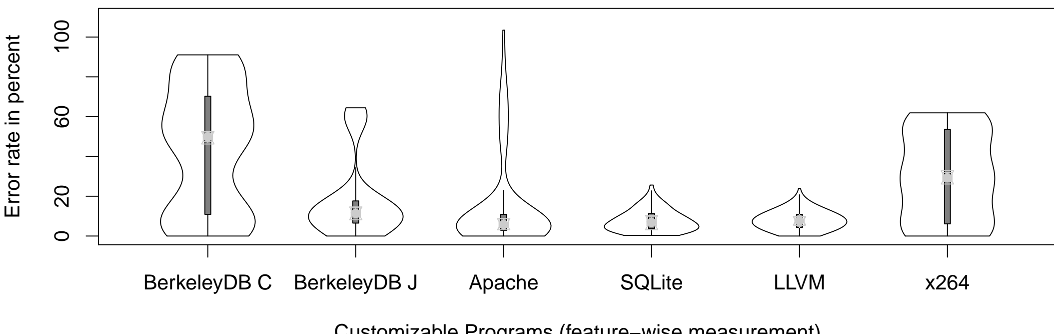


Figure 6.13.: Error rates in percent of all customizable programs for performance.

- Prediction accuracy differs between main-memory consumption and performance for a single program (e.g., Berkeley DB C).
- We confirmed that prediction error rates heavily differ among different programs.

Next, we describe threats to validity and discuss our evaluation afterwards.

6.2.5. Discussion: The Influence of Feature Interactions

In our evaluation, we answered the research question whether feature-wise measurement is feasible to predict different non-functional properties of highly customizable programs without considering feature interactions. Regarding measurement effort, we can clearly state that we always need only a linear number of measurements. Hence, even for programs, such as SQLite, which allows us to generate up to 2^{85} variants, the approach scales. Unfortunately, the picture is not that clear when it comes to prediction accuracy. The observed error rates substantially differ between non-functional properties and sample programs. Our hypothesis is that feature interactions cause the observed error rates. We discuss their influence in the following.

Footprint. We achieve the best accuracy for footprint. With one exception, we reach a mean error rate for all sample programs of 0.6 %. That is, a prediction accuracy of 99.4 % for all variants of eight different programs. This is an amazingly good result, considering the large diversity of used programs, implementation techniques, and programming languages.

Let us have a closer look at Berkeley DB, Violet, and the Linux kernel, because their results show interesting points for further investigations. Berkeley DB is a customizable database that makes exhaustive use of nested `#ifdefs`. This means, it is often the case that a certain feature combination requires additional code, which increases footprint for this feature combination. Hence, our predictions become

inaccurate. For Linux, we expected heavy inaccuracies regarding the 100 randomly generated variants, because all Linux features affect the size of other features (as selected by a domain expert). We were surprised that we still achieved a quite precise prediction, even without considering feature interactions, partially because the features had a weaker effect than expected.

For Violet, we observed the largest error rates (above 180 %). Hence, we analyzed this program and identified that there is a complex mapping between (some) features and implementation units. That is, an individual feature may map to multiple implementation units (i.e., feature a maps to units $\tau(G) \cdot \tau(H) \cdot \tau(J)$) and a single implementation unit may be used by multiple features (i.e., features a, b, and c map to $\tau(G)$). Hence, when measuring such a feature, the corresponding variant contains several implementation units that are also present when measuring another feature's variant. Therefore, predicting footprint of a variant that includes multiple features with an overlapping set of implementation units is inaccurate, because we consider the footprint of the implementation units multiple times.

We believe that programs with a complex mapping are the exception rather than the rule, because in many customization techniques, such as command-line parameter, a feature maps to a single variable (which in turn changes the behavior of the program) and not to multiple variables. Also for compile-time customization, mapping from features to preprocessor variables (i.e.,`#define` statements) is mostly one-to-one, as it is usually the case for programs implemented with feature-oriented and aspect-oriented programming. Considering a component-based program, we may have an n-to-m mapping, but this mapping is often not distributed over many components. That is, n or m is small, which was, however, not the case for Violet. Nevertheless, to improve predictions also in the presence of complex mappings, we have to find means to incorporate the mapping in the measurement and prediction process.

Main-Memory Consumption. In our second experiment, we observe higher error rates, on average, for all programs. For footprint, a feature interaction occurs that degrades our prediction when in a specific feature combination additional code is included or code is reused by multiple features. Since these cases can be easily detected by static code analyses, they do not cause a serious problem. This is different when considering main-memory consumption or performance. Here, we often have no concrete code fragment that causes a feature interaction. Instead, features interact implicitly by using the same resource: main memory. Here, we may use more main memory in a certain feature combination, which is, however, not explicitly encoded. For example, there is no nested `#ifdef` statement that consumes additional main memory. Hence, we argue that interactions occur in a greater number. Our assumptions is also backed up by the increased prediction error rates compared to footprint.

Performance. In our performance experiment, we observe similar results as for main-memory consumption. The error rate is higher for more programs compared to our footprint evaluation. Hence, feature interactions seem to exist in larger numbers or are more significant than for footprint. From this insight, we conclude that without considering feature interactions in measurement and prediction, we cannot achieve accurate predictions for many programs. Furthermore, a common user cannot rely on domain knowledge when identifying feature interactions, because programs, such as Apache or x264, are black boxes for them. Investigating the over 230 000 lines of code for possible feature interaction in a reasonable time is infeasible for a common user and a challenge even for a developer. Also, asking a developer may work in a single case, but is likely to fail when all customers start asking for possible feature interactions in their specific application scenario. In other cases, we do not have any implementation artifacts available, nor do we know anything about the internals to guess which features may interact. Hence, we have to develop means to identify feature interactions for black-box programs.

Comparison of Non-Functional Properties. We want to discuss the influence of different non-functional properties on prediction accuracy in more detail. First, let us have a look on Figure 6.14. We compare average prediction error rates for the three non-functional properties. Clearly, we see that footprint predictions are significantly more accurate than predictions of main-memory consumption and performance. One reason is that footprint is not subject to measurement bias, but the other properties. This can explain that the approximation of the influence of single features may already be inaccurate and therefore, the prediction. However, this has only a small effect.

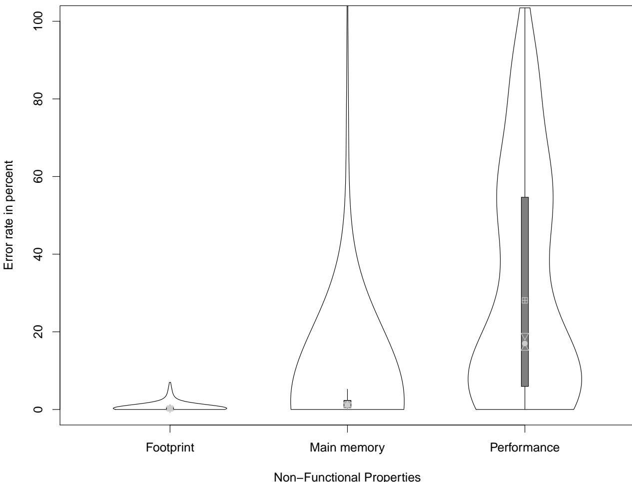


Figure 6.14.: Violin plots of average prediction error rates of the three non-functional properties.

We have the hypothesis that the main difference between these properties is due to the greater number of causes that potentially yield feature interactions. For footprint, a feature interaction exists when code is reused or additionally introduced in a certain feature combination. Hence, it requires a physical artifact to emerge. This is different for performance and main-memory consumption. Here, environmental influences, implicitly shared hardware resources (e.g., CPU), and the source code can cause feature interactions. These large number of possible causes may explain the higher average error rates and may also be a reason why different programs exhibit such large differences for the same property. For example, we predicted all variants of Berkeley DB C version for main-memory consumption with an average error rate of 1.4 %, whereas we predicted with the same program an average error rate of 44.1 % for performance. Since the source code and features remain the same, other factors, such as resource usage, must be a cause of these inaccuracies, which cannot occur for footprint.

6.2.6. Extended Problem Statement

We use a constructive research approach. Hence, we extend our initial problem statement (see Section 3.1.6) with insights we gained during our evaluations. We found that we satisfied our initial requirements only partly. Although we determined the influence of individual features in a black-box fashion, our predictions are not accurate in all cases. We identified that feature interactions have a crucial influence on non-functional properties. Hence, a holistic measurement and prediction approach has to detect feature interactions and quantify their impact (R4). However, detecting feature interactions usually increases measurement effort, because we need to measure more variants. Thus, we require a detection approach that is both: efficient in terms of number of measurements and provides accurate predictions. We extend our initial problem statement with requirement R4:

- **(R1)** Accurate predictions of non-functional properties.
- **(R2)** Black-box approach.
- **(R3)** Determining the influence of individual features on non-functional properties.
- **(R4)** *Efficient detection of feature interactions.*

Based on these requirements, we extend feature-wise measurement to incorporate detection and quantification of feature interactions, which we present in the next chapter. Next, we discuss threats to validity of our evaluation.

6.2.7. Threats to Validity

Most of the presented threats to validity apply to all our evaluations. If necessary, we notify when there are additional threats in other evaluations.

Construct Validity. Benchmarks influence the outcome of measurements. By choosing inappropriate benchmarks, we may improve or degrade prediction accuracy. We use standard benchmarks delivered by vendors or used in the respective community if possible. These benchmarks are designed by domain experts to represent a common program workload. Hence, we perform our predictions according to the expected workload. In case of Wget, we could not find any standardized benchmark, which leaves room for a validity threat. We chose to download the Apache manual as a benchmark, because there is a large spectrum of common use cases (e.g., large files, many small files, nested folders, and pictures).

Conclusion Validity. The reliability of measures strongly affects the conclusion validity of experiments. In our first experiment, we use footprint, defined as binary size of the generated program. For this measure, we can accurately determine the true size of programs using OS functions either by aggregating the size of all class files of a Java program or by determining the size of the executable in a case of a C/C++ program. Regarding main-memory consumption, we use the Linux Gnu tool *time* and depend therefore on its reliability. It measures the maximum resident set size of the process during its lifetime in KB. Since we measure only peak memory and not the average consumption over time, our measurements are not affected by page swapping of the OS or other influential factors that may be able to affect validity of measurement. Furthermore, we repeated measurements and identified that the differences of several runs with the same configurations are below one percent.

Internal Validity. For customizable programs with many features and for the main-memory and performance evaluation, we only sampled 100 variants to compare prediction and measured property, because we cannot possibly generate and measure all products (exponential with the number of features) in reasonable time – this is exactly the motivation for our approach. We are aware of our evaluation leaving room for outliers, but we believe that 100 samples provide a reasonable number.

When measuring main-memory consumption and performance, we have to deal with measurement bias. We repeated each measurement three to ten times and use the arithmetic mean of these measurements to compute a feature' influence and for our evaluation. For performance, we similarly repeat measurements, such that the width of the 95 % confidence interval is smaller than 10 % of the according means. With the exception of RAR, we observed only small deviations when measuring the same variant multiple times for main-memory consumption (below 1 %) and

a maximum bias of 7 % for performance, which indicates that the measurement procedure is reliable. Also, we are aware that measurement bias can cause false interpretations [Mytkowicz et al., 2009]. Since we aim at predicting performance for a special workload, we do not have to vary benchmarks.

External Validity. Although we use a large variety of different customizable programs, we are aware of that the results of our evaluations are not automatically transferable to all other customizable programs and all kinds of customization techniques. We selected real-world customizable programs from different domains, having different sizes, and using varying implementation techniques. Furthermore, different compilers and workloads are used to evaluate prediction accuracy for a broad range of application scenarios. Our used programs have feature models with a typical structure and number of constraints (according to the criteria defined by [Thüm et al., 2009]). We did not evaluate customizable programs with an unusual, possibly degenerated feature model, which might influence the computation of the variant set (cf. Section 6.1.2). Thus, we cannot generalize our results to such programs.

Although we used three important non-functional properties with different characteristics, we cannot yet judge our approach for properties that exhibit different behaviors for the same workload within the same environment. That is, if the measurement of the same variant yields different results, also the predicted variant can have heavily changing values of a non-functional property. We do not address this issue in this thesis and leave it for future work.

Finally, we cannot generalize our evaluation to non-functional properties other than footprint, main-memory consumption, and performance. However, we argue that footprint, main-memory consumption, and performance are subject to many internal and external influences as many other non-functional properties do. Hence, we expect that also other properties, such as energy consumption, are influenced by the same factors and produce an equivalent behavior. Furthermore, these external and internal influences have a crucial impact on the applicability of our approach, which we could handle for these three properties. In this series of experiments, we want to convey that the approach of approximating non-functional properties per features (i.e., without taking feature interactions into account) is realistic at all.

6.3. Related Work

Only a few approaches apply measurements of non-functional properties to customizable programs (especially software product lines). Dave Zubrow and Gary Chastek [2003] proposed measures that evaluate the development effort for a software product line. Lopez-Herrejon and Apel [2007] express the complexity of a product line in terms of variation points with a dedicated metric. An approach close to our work is the measurement of the binary size of aspect-oriented programs [Hunleth and

Cytron, 2002]. The authors compiled aspects in distinct files and measured the binary size. The footprint of different variants can then be computed. Another related approach for optimizing non-functional properties was developed in the *COMQUAD* project [Aigner et al., 2003]. The project focuses on techniques for tracing and adapting non-functional properties in component-based systems. Particular, alternative implementations can be selected dynamically and are weaved as non-functional aspects in the component [Göbel et al., 2004]. The approach requires an own component model, which is an extension of *Enterprise JavaBeans* and *CORBA Components* and relies on aspect-oriented programming as implementation technique. In contrast to these approaches, we consider also other non-functional properties and address the exponential number of variants of customizable programs. Moreover, we use a black-box approach. That is, we are independent of how a program is implemented and the customization is realized. Hence, we are not restricted to aspects or components.

Sincero et al. [2007, 2010] propose to estimate a variant's non-functional properties based on a knowledge base consisting of measurements of already produced variants. Their *Feedback* approach aims to find a covariance between feature selection and measurement. This way, it can give information about how a feature influences a non-functional property during configuration. In contrast to our approach, they do not determine to what extent a feature affects non-functional properties, but present a qualitative statement, such as feature a improves performance. When it comes to program derivation, they do not present an expected value for a variant's non-functional properties, as we do, but show with a slider whether a feature selection improves a property.

Abdelaziz et al. [2011] argue that most measurement approaches for the prediction of non-functional properties lack generality, as they are applicable only to specific application scenarios or infrastructures [Chen et al., 2005, Yacoub, 2002]. In contrast, our work can be used for a broad range of applications of different domains, implementation techniques, etc.

Profiling tools, such as *GNU gprof*[9] and TPTP [10], usually intercept the execution of a target application and monitor its execution. They can identify performance bottlenecks or other performance-relevant execution states. However, profiling tools have the drawback that only a single configuration can be profiled at a time. Results of such a profiling run can usually not be transferred to other configurations to gain information about how different configurations affect the performance. Hence, they are commonly combined with prediction models.

Chen et al. [2005] use a combined benchmarking and profiling approach to predict the performance of component-based applications. Based on a benchmark and a Java profiling tool, a performance prediction model is constructed for application

[9]http://www.gnu.org/software/binutils
[10]http://www.eclipse.org/tptp

server components. In contrast, we correlate the measurements to the configuration, and measure only those configurations from which we expect to detect performance feature interactions.

Especially in the database domain, performance benchmarks and predictions are frequently used. For example, Agrawal et al. [2004] propose an automated database tuning advisor that is capable of computing the optimal database schema for achieving the best performance (e.g. which indexes or materialized views should be created). Similarly, the DB2 Design Advisor estimates for a given workload the best database design [Zilio et al., 2004], which can be compared to a configuration in our case. Both approaches perform measurements or evaluate query plans via their internal query optimizers to predict the performance of a certain design. Different features, such as indexes, materialized views, and partitioning, influence each other, which is similar to the non-functional feature interactions we are looking for in the next chapter. Such approaches commonly use cost models and are limited to a single database engine and heavily rely on domain knowledge. Furthermore, we do not need any domain knowledge about the database internals for our predictions.

6.4. Summary

We proposed feature-wise measurement of customizable programs to quantify the influence of individual features on non-functional properties. We have shown that by measuring two variants that differ only in a single feature, we can interpret the result of both measurements and the influence of the differing feature on a non-functional property. We demonstrated the computation of these influences as *feature terms* in the prediction model for the common relationships that are present in feature models. We presented our algorithm to compute a set of configurations that have to be measured. We explained that computing the influence of features not individually, but all at once with a set of equations, minimizes errors caused by measurement bias and feature interactions.

We evaluated our approach with three experiments using the non-functional properties footprint, main-memory consumption, and performance. We selected customizable programs from different domains (e.g., databases, operating systems) and origins (i.e., academic and real-word), implemented with different programming languages, and customized with varying techniques (e.g., compilation and command-line parameters). We demonstrated that feature-wise measurement achieves a mean prediction error rate of 0.6 % for all programs without Violet for footprint. However, we observed high error rates for main-memory consumption (13.6 %, on average, ranging from 0.4 % error rate for Curl to 28.8 % for LLVM) and performance (20.3 %, on average, ranging from 7.8 % error rate for SQLite to 44.1 % for Berkeley DB C version). We identified that feature interactions are a main cause for these error

rates and extended our problem statement with the requirement to detect feature interactions.

We made the following observations in our evaluation:

- Prediction accuracy depends on the non-functional property. We observed low error rates for footprint (below 5 %) and high error rates for main-memory consumption (below 14 %) and performance (below 21 %). We identified feature interactions as a cause for these inaccuracies, which occur in different quantity for the different properties.
- Different programming languages and customization techniques have no effect on the accuracy of predictions.
- Feature-wise measurement performs surprisingly good for many programs considering the measurement of only a linear number of variants.

Clearly, the evaluation has shown that we have to determine also the feature-interaction terms in our prediction model. Hence, our next step is to develop means to quantify the influence of feature interactions on non-functional properties to improve prediction accuracy.

7. Feature Interactions

This chapter shares material with the following papers:

- "Scalable Prediction of Non-Functional Properties in Software Product Lines." in *SPLC'11* [Siegmund et al., 2011] and
- "Scalable Prediction of Non-Functional Properties in Software Product Lines: Footprint and Memory Consumption." in *IST'12* [Siegmund et al., 2012b]

We introduce feature interactions with respect to non-functional properties. That is, we describe with several examples what feature interactions at the level of non-functional properties are, give possible causes for feature interactions, and discuss different types of interactions. Based on this knowledge, we propose an extension to feature-wise measurement to incorporate feature interactions. We propose two feature-interaction-detection approaches that tackle some of the requirements given in the previous section and evaluate their accuracy and measurement effort. Based on this evaluation, we make several observations that lead to a further analysis of the evaluation results. In this analysis, we gain additional insights in the nature of feature interactions, such as patterns for the distribution of interactions among features. Based on these insights, we derive heuristics for the detection of feature interactions, which we use in an automated detection approach presented in the next chapter.

7.1. Introducing Feature Interactions

A common definition of feature interaction is the following [Calder et al., 2003b, Nhlabatsi et al., 2008]:

> Two features interact if their simultaneous presence in a configuration leads to an unexpected behavior, whereas their individual presences do not.

This definition raises two questions. First, we have to clarify what an unexpected behavior is. Second, we have to discuss when features interact.

To understand what an unexpected behavior is, we have to clarify first that there are different types (or levels) of feature interactions. That is, depending on our scope and context, a feature interaction may be a completely different thing. For example, in the context of non-functional properties, a feature interaction occurs when non-functional properties of a variant (perhaps) unexpectedly change for a certain

feature combination (e.g., performance drops significantly only when selecting two specific features in combination). This can be the case if two features are computation intensive, such that frequent context switches are necessary, which have a significant effect on the performance of both features.

With a semantic context, the meaning or functionality of a program may change for a certain feature combination. Remember our example from fire and flood control in Section 4.1, in which the behavior of the system was unexpected when selecting both sensors, the flood-control and the fire-alarm sensor.[1] Hence, feature interactions can mean different things. In this thesis, we concentrate on feature interactions at the level of non-functional properties only.

The definition above states that two features interact only if there is an unexpected behavior. Although interaction terms are always present in Batory's composition model, features are not required to interact. That is, an interaction term does not necessarily have a corresponding implementation or, when considering non-functional properties, does not always have a non-zero value. Equation 4.1 defines that a customizable program with n features has 2^{n-1} feature-interaction terms. To relate the above definition to Batory's feature-composition model and, subsequently, to our prediction model, we have to differentiate between *relevant* and *non-relevant* feature interactions. Relevant feature interactions cause an unexpected behavior as stated in the definition above. This means that a feature-interaction term of a relevant feature interaction is *not zero* and has, thus, an *observable* effect in a variant. This ultimately means that all non-relevant feature-interaction terms have a value of *zero* and have no measurable effect on non-functional properties. Hence, when speaking of feature interaction, we refer to relevant feature-interaction terms that have a measurable effect on non-functional properties.

To clarify what a feature interaction at the level of non-functional properties is, we give two concrete examples.

7.1.1. Example: Footprint Feature Interaction.

To illustrate a non-functional feature interaction, we show in Figure 7.1 the C++ implementation of a customizable linked list with two features: *PrintList* and *PrintElement*. Features are implemented with conditional compilation. To determine the feature terms, we measure each individual feature. That is, we measure the footprint of the code in Lines 5 and 6 as well as Line 11 for feature *PrintList*. We would not measure Lines 8 and 9, because these lines are compiled only for a variant that contains both features *PrintList* and *PrintElement*. Hence, if we predict footprint of a variant that includes both features, the prediction would be inaccurate. To predict

[1] During fire, the fire-alarm sensor turns water on, but then flood is detected by the flood-control sensor, so water is turned off and the building burns down.

```
1  class List {
2    int numberOfElements;
3    Element* head;
4  #ifdef PrintList
5    void printList() {
6      cout << numberOfElements;
7  #ifdef PrintElement
8      if(this->head != NULL)
9        printElement(this->head);
10 #endif
11   }
12 #endif
13 #ifdef PrintElement
14   void printElement(Element* node){
15     node->print();
16     if(node->hasNext())
17       this->printElement(node->getNext());}
18 #endif
19 };
```


Figure 7.1.: C++ code of a customizable list with two features: *PrintList* and *PrintElement*. We show the feature model in the upper right corner.

the footprint correctly, we have to measure the influence of the feature interaction caused by the additional lines of code (Lines 8-9).

As another example, consider a set of features that use the same resource. A shared resource may be an external library or otherwise shared code. We can easily extend the list example to show the problems of shared resources. We may use an external library to log the elements of a list instead of printing them. To this end, we change the call to *cout* and method *print* (in Lines 6 and 15 of Figure 7.1) to use the external logging library. The library has a considerably larger binary size than the features itself. When approximating a feature's non-functional properties, the predominant part of the footprint would stem from the logging library. Because we measure the size of the library for both features, we would predict the size of a variant with both features unexpectedly incorrect. The reason is that both features share the same library, which is included only once in the variant, but was measured twice (once for each feature). So the interaction term in our prediction model for these features would correct for the double count of the logging library.

7.1.2. Example: Performance Feature Interaction.

A relevant feature interaction for performance occurs when we observe a completely different performance for a specific feature combination than expected. This can be a degraded or improved performance. Consider the following example: A streaming database has two optional features: *Encryption* and *Compression*. The first step is to measure performance of both features in isolation. That is, we gener-

ate for each feature two variants: one with and the other without that feature. Assuming we executed a benchmark and measured the following response times $\Pi_{Perf}(Encryption) = 100\,s$ and $\Pi_{Perf}(Compression) = 80\,s$, we predict a variant with both features:

$$\begin{aligned} \Pi_{Perf}(Encryption \times Compression) &= \Pi_{Perf}(Encryption) + \Pi_{Perf}(Compression) + \\ &\quad \Pi_{Perf}(Encryption\#Compression) \\ &= 100\,s + 80\,s + 0\,s \\ &= 180\,s \end{aligned}$$

However, when we actually measure performance of this variant, we obtain a value of $140\,s$. The discrepancy between the predicted $180\,s$ and the measured $140\,s$ is caused by a feature interaction. Feature *Compression* compresses the data in the workload. Afterwards, the compressed data is encrypted. Since the compressed data is significantly smaller, encryption does not take $100\,s$, but only $60\,s$. Hence, when we want to predict the variant with both features correctly, we have to set the feature-interaction term $\Pi_{Perf}(Encryption\#Compression) = -40\,s$ in our prediction model.

In the footprint example, the interaction exists, because a developer defined additional code in a nested `#ifdef`, which is compiled only when both features, *PrintList* and *PrintElement*, are selected in combination. That is, the feature interaction is caused by the source code of the program. In the performance example, the interaction is not visible directly in the source code, but by using and changing a shared resource. Both features, *Encryption* and *Compression*, work on the same resource data stream. If one feature changes the data, it ultimately affects all other features using the same data afterwards. We discuss causes of feature interactions in Section 7.1.4 in detail.

7.1.3. Order of Interactions.

Feature interactions can occur with an arbitrary number of features. To denote the number of interacting features of a feature interaction, we use the term ´´order". That is, the order of an interaction specifies the number of interacting features. When two features interact, we observe a first-order feature interaction (also called pair-wise interaction). If three features interact, we observe a second-order interaction and so forth. In literature, a feature interaction with an order of two or higher is called a higher-order feature interaction [Kim et al., 2008].

The order of an interaction is also encoded in our prediction model. A first-order feature interaction is denoted with a single interaction symbol #. For instance, the interaction between features a and b is denoted with $a\#b$. We denote a second-order interaction between features a, b, and c with $a\#b\#c$.

7.1.4. Causes of Feature Interactions

Relevant feature interactions at the level of non-functional properties can have multiple causes. In the following, we present these causes to help developers in detecting and omitting feature interactions and to help understand our evaluation results.

Functional Interactions. If two features interact at a functional level, for instance, a feature parallelizes another feature's processing, it affects often also non-functional properties. Hence, to nearly every functional feature interaction, we observe also a non-functional feature interaction for some non-functional properties. To interact at a functional level, a feature usually requires additional code that is responsible for operating together with another feature (e.g., calling methods of other features or changing data). These code units are called lifters or derivatives [Liu et al., 2006] for compositional implementation techniques and nested `#ifdefs` for conditional compilation. Hence, when searching for non-functional feature interactions, we may start searching for functional feature interactions first by using static code analysis. We can measure non-functional properties of the corresponding feature combination, in which the additional code is used, to determine whether a non-functional feature interaction exists.

Shared Resources. Features can share arbitrary resources. These resources can be implementation units such as components and aspects, external libraries, and hardware resources. For example, features *PrintList* and *PrintElement* share the same code fragment in line 4 of Figure 7.1. If a customizable program contains a shared resource, we can analyze the source code, the architecture, or an implementation model to identify shared resources. However, this identification may require a high effort, because different tools and techniques are needed to retrieve the necessary information. For example, to find the shared code in Line 4, we need a parser, which scans all code files in the program for a shared code pattern.

Using composition based approaches, such as FeatureHouse [Apel et al., 2012], we may analyze a feature model. For example, we search in a feature model for elements with requires relationships. Particularly, we search for the pattern: feature a OR feature b requires feature c. In this case, c is a shared resource. When considering shared resources that do not belong to the program itself (e.g., external libraries) the identification becomes complex. Even a call of a simple print function in Java can result in a shared resource (i.e., importing a Java SDK function). Finally, sharing non-code related resources, such as energy, working memory, and threads, raises the most challenging problems, because there is no easy way to identify these feature interactions. Such interactions have no physical representatives, e.g., in the form of implementation units.

Environmental Influences. The environment, in which a variant runs, can also impose non-functional feature interactions. Consider a customizable program with two features that require large portions of main memory. If our system runs out of memory, these two features interact, because they both have not enough memory to run properly (e.g., we need to frequently swap large chunks of data to persistent storage), which degrades performance substantially. If we select only one of these features or if we increase the amount of available memory, the program runs as expected. Other environmental factors are operating systems, hardware resources in general, user interactions, as well as interactive and dynamically changing events.

7.2. Detection Approaches

There are multiple ways to detect feature interactions. Here, we present two approaches: (a) using domain knowledge, called *interaction-wise (IW)* measurement and (b) using *pair-wise* measurement. We selected interaction-wise measurement, because many feature-interaction-detection approaches use domain knowledge and implementation artifacts to find interactions, which we present in related work (Section 8.5 in the next chapter). Hence, we adopt the general idea of using implementation artifacts and transfer it to the detection of non-functional feature interactions.

We choose pair-wise measurement, because related approaches in software testing use the same technique to detect bugs and errors for certain feature combinations [Oster et al., 2010]. Hence, we want to evaluate whether this state-of-the-art technique is transferable to the detection of non-functional properties. In the next chapter, we propose an additional solution that is based on the insights we gain during the evaluation of interaction-wise and pair-wise measurement.

Interaction-Wise Measurement. Previously, we described different causes of non-functional feature interactions. Often, we have to know the customizable program and its source code to identify such causes. However, if we have this knowledge, we can use it to find relevant feature-interaction terms. As we explained for our prediction model (see Section 4.2), we have to find only the relevant feature-interaction terms (i.e., those terms that are non-zero) to improve prediction accuracy. With domain knowledge, we *know* these interactions, for example, by statically analyzing the source code to find functional feature interactions. The remaining task is to measure these interactions, such that we can quantify their impact on non-functional properties.

For each identified interaction, we produce a variant that contains the features that interact. By calculating the difference between the measured and the predicted value, we quantify the influence of the interaction on a non-functional property. That

is, we subtract the influence of all already quantified terms from the measurement. The remaining value represents the influence of the interaction:

$$\Pi(a\#b) \quad = \quad \Pi(a \times b) - \Pi(a) - \Pi(b)$$

where $\Pi(a \times b)$ quantifies the measurement and $\Pi(a)$ and $\Pi(b)$ represent the influences of both selected features, which we determined using feature-wise measurement.

This way, we have to measure $O(n+k)$ variants, in which n is the number of features and k is the number of known interactions. If $k = 0$, interaction-wise measurement is identical to feature-wise measurement. Measuring all interactions improves the accuracy in the prediction of a variant's non-functional properties. Especially when a customizable program contains a large number of features, domain knowledge can help to identify which of them interact. Our assumption, which we want to evaluate, is that this approach results in a solid prediction base.

Pair-Wise Measurement. Pair-wise measurement (or in general n-wise measurement) means that we measure all (first-order) interactions between pairs of features. That is, a pair-wise measurement means that we measure and quantify all feature-interaction terms, in which exactly two features interact. A triple-wise measurement would mean that we additionally measure all feature-interaction terms, in which three features interact (an order of two).

Since we do not use any domain knowledge, pair-wise measurement is a brute-force-like method to quantify additional terms to improve prediction accuracy. With an increasing order of interactions, we have to measure an increasing number of variants. Hence, only the measurement of pair-wise interactions is usually feasible. We measure $n(n-1)/2 + n$ variants, in which n is the number of features. The approach results in a substantially increased variant set to measure compared to feature-wise measurement ($O(n)$ vs. $O(n^2)$). This concept is similar to pair-wise testing [Williams, 2000], in which test configurations are determined, such that all input variables of a programs are covered.

Also both approaches cannot accomplish all our goals individually, we expect that the evaluation of these approaches provides insights and observations to develop a feature-interaction detection that achieves all goals. In the the following, we evaluate these two approaches and discuss their results.

7.3. Evaluation

The goal of our evaluation is to rate prediction accuracy and measurement effort of interaction-wise and pair-wise measurement. To this end, we use two experiments with the non-functional properties footprint and main-memory consumption. For

main-memory consumption, we cannot use static code analysis to find all interactions manually, because interactions are often not visible in the source code. Also, we do not have expert knowledge, which is, however, necessary to manually identify feature interactions. Hence, we can evaluate prediction accuracy regarding main-memory consumption only with pair-wise measurement.

We did not evaluate pair-wise measurements for performance, because we would need two months for the additional measurements.[2] Since we observed in the previous experiment that performance behaves equal to main-memory consumption, we omit performance here, but use performance for the experiment in the next chapter.

The evaluation is structured as follows. First, we present our research questions and the adapted experimental design. Second, we show for each non-functional properties the results of the experiment separately (Sections 7.3.2-7.3.3). Note that we neither make a discussion at this point nor do we compare both approaches, but present only the data. Afterwards, we compare both feature-interaction-detection approaches by analyzing and discussing the experimental results in Section 7.3.4.

7.3.1. Experiments

Since we use the same experimental setup as in Section 6.2 (see Figure 7.2 for a short summary), we describe only the differences here. To achieve our goal of rating feasibility of both measurement approaches, we state the following research questions:

- **Q1:** What is the average error rate of interaction-wise measurement? (only footprint experiments)
- **Q2:** What is the average error rate of pair-wise measurement?
- **Q3:** How do interaction-wise and pair-wise measurements perform against feature-wise measurement and each other in terms of prediction accuracy and measurement effort?

Again, we use the error rate of predictions as the metric for our evaluation: $\frac{|actual-predicted|}{actual} * 100$. Since the measurement effort plays a crucial role, we also compare both measurement approaches against the *brute-force approach (BF)*. The brute-force approach assumes that we measured all variants, such that the error rate is 0 % and the effort is equal to the number of valid configurations (i.e., $O(2^n)$ in the worst case for n features).

[2] We cannot conduct interaction-wise measurement for performance for the same reasons as for main-memory consumption, i.e., static code analysis does not find all interactions and expert knowledge is not available.

Experimental Setup for Feature-Wise Measurement (see Section 6.2):

- Error rate as metric for evaluation: $\frac{|actual-predicted|}{actual} * 100$
- Two-step design: (i) Build prediction model with measurements and (ii) evaluate prediction model against whole population or randomly selected variants.
- Footprint material: *LinkedList, Prevayler, ZipMe, PKJab, SensorNetwork, Violet, Berkeley DB, SQLite*, and *Linux kernel.*
- Main-memory-consumption material: *Curl, LLVM, x264Wget, Berkeley DB CE*, and *SQLite.*

Figure 7.2.: Recap: Experimental setup of feature-wise measurement.

Experimental Design. For the experimental design, the first step is to build the prediction model. The main difference to feature-wise measurement is that we have two additional measurement approaches, for which we build two additional predictions models (see Figure 7.3 IW and PW). To this end, we have to determine additional configurations, for which we produce the corresponding variants and measure their non-functional properties. We approximate feature-interaction terms by using the difference of prediction and measurement. Phase 2 is similar to our first experiment, but we have to predict the error rates for our two additional prediction models. Again, we measure all or 100 random variants and compare measurement against prediction.

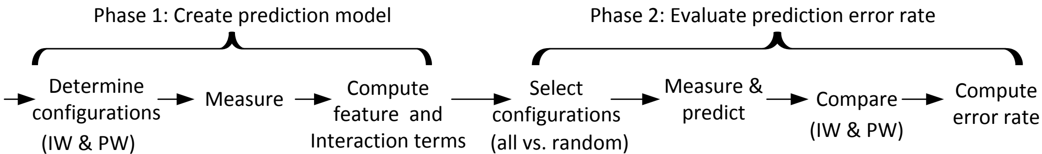


Figure 7.3.: Experimental design with interaction-wise (IW) and pair-wise (PW) measurement.

For the interaction-wise prediction model, we measure the influence of known interactions. We analyze the source code of the used customizable programs to gain the knowledge which interactions exist. To this end, we build a model that describes the mapping between features and implementation units to find feature interactions. As explained in Section 4.2, we can have several instances of the abstract feature-composition model. One instance is the prediction model and another instance is the mapping model.

To build the mapping model, we have to determine how features map to implementation units by analyzing the source code of our sample programs. With a self-written tool, we detect nested `#ifdef` statements in programs based on conditional compilation (cf. Table 6.2.4). For compositional approaches, we search for the existence of interaction modules (e.g., derivatives [Liu et al., 2006]). If we find a nested `#ifdef` or an interaction module, we define the corresponding mapping. For example, we detected

for Berkeley DB that features *Statistics* and *Replication* have nested `#ifdefs`. Hence, we additionally create the term $\tau(\textit{Statistics\#Replication})$ in the mapping model. Selecting both features results in the compilation of three implementation units: $\textit{Statistics} \times \textit{Replication} = \tau(\textit{Statistics}) \cdot \tau(\textit{Replication}) \cdot \tau(\textit{Statistics\#Replication})$. In the case that two features a and b map to the same implementation unit, we note $\tau(a \cdot b)$ in the mapping model.

We measure additional configurations, for which we either have to compile an additional implementation unit (i.e., when $\tau(a\#b)$ exists) or when multiple features map to the same unit (i.e., $\tau(a \cdot b)$). In the first case, we need only one additional measurement to quantify the influence of $\tau(a\#b)$ on footprint. In the second case, we need multiple additional measurements when more than two features map to the same implementation unit. For instance, if features a, b, and c map to a single unit $\tau(a \cdot b \cdot c)$, we have to measure four additional configurations: $C_1 = \{a, b\}$, $C_3 = \{a, c\}$, $C_3 = \{b, c\}$, and $C_4 = \{a, b, c\}$. This is because our prediction model $a \times b \times c$ expands to seven terms, that is, $\Pi(a)$, $\Pi(b)$, $\Pi(c)$, $\Pi(a\#b)$, $\Pi(a\#c)$, $\Pi(b\#c)$, and $\Pi(a\#b\#c)$, that all have an influence on prediction accuracy.

Regarding pair-wise measurement, we define that each pair of features has a relevant feature interaction and so we have to determine its influence on a non-functional property.

In the following, we first describe both experiments, footprint and main-memory consumption, separately and present the corresponding results. Afterwards, we analyze and discuss the results by comparing interaction-wise and pair-wise measurement regarding prediction accuracy, measurement effort, and applicability (in Section 7.3.4).

7.3.2. Footprint

In this section, we present the experimental procedure and the results for the footprint experiment.

Experiment Procedure. To measure footprint with the interaction-wise approach, we have to manually define relevant feature interactions. This changes the experimental procedure of the previous experiment in Section 6.2.2, such that we define additional configurations for measurement.

We use three different sources to identify feature interactions depending on the program. For Violet, LinkedList, ZipMe, PkJab, and SensorNetwork, we use the mapping between domain features and implementation units. We analyze the source code of Berkeley DB, Prevayler, and SQLite for nested `#ifdefs`. We ask a domain expert for the Linux kernel to identify feature interactions. For each of the defined interactions, we measure an additional variant that includes the features that par-

ticipate in the respective interaction. Furthermore, we try to minimize the number of features of this variant to only include the interaction that we want to determine.

Deviations. For SQLite, we observed compilation errors of valid variants. As a consequence, it was not possible to measure all variants that are valid according to the feature model. The effect of the compiler errors is that we could not measure all pair-wise interaction and not all manually specified interaction. That is, the feature-wise prediction model is complete, but the other prediction models lack some interaction terms.

These incomplete prediction models may degrade accuracy, because of the missing terms. However, considering the huge configuration space of SQLite, the few failed configurations are negligible. The reason for these compilations errors is that SQLite was not tested for all possible combinations, as also noted by the developers.[3]

Results. In Table 7.1, we summarize the results of the footprint measurements and predictions for all customizable programs. To put the results into perspective, we additionally show the measurement effort of feature-wise measurement and the brute-force approach (i.e., measuring all variants). We use box plots in the table to highlight the distribution of the error rates and to ease comparing the different prediction models. We use box plots instead of violin plots, because they provide a more comprehensible view on the same data. Hence, we ease comparing the raw data in the table with the distribution.

Predictions based on domain knowledge are in general better than for feature-wise measurement. For Q1, we identify an error rate of 0.18 %, on average, (i.e., an average accuracy of 99.82 %). That is, when determining relevant feature-interaction terms based on code analysis and mapping information, we achieve very precise prediction accuracy even for the problematic case of Violet. In Figure 7.4, we show the error-rate distributions for the interaction-wise measurement.

Regarding Q2, determining pair-wise interaction terms usually reduces prediction error rate for all programs without Violet from 0.6 % (FW) to 0.23 % (see Figure 7.5). With Violet, the average error rate is 80.5 % over all customizable programs. This high average error rate is due to Violet's error rate of 722 %.

The key results are as follows:

- Interaction-wise measurement considerably improves prediction accuracy and handles also complex cases.
- Pair-wise measurement can improve prediction accuracy, but may even worsen the results when complex mappings between features and implementation units exist.

[3]Source: `http://www.sqlite.org/compile.html` [Accessed on August 30th, 2012].

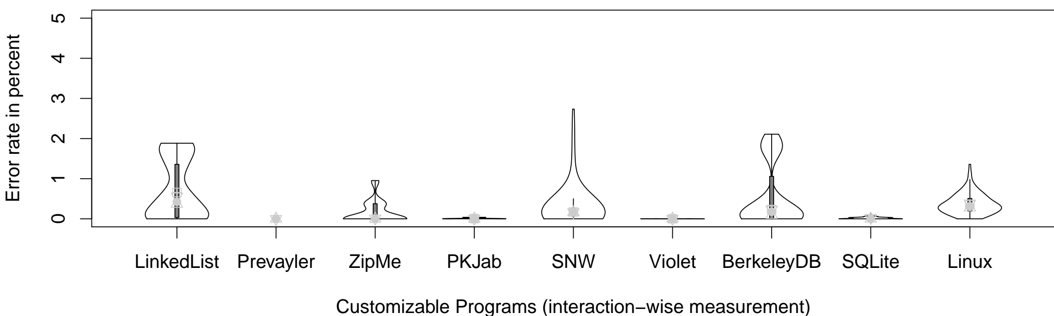


Figure 7.4.: Error-rate distribution of predicting footprint with interaction-wise measurement.

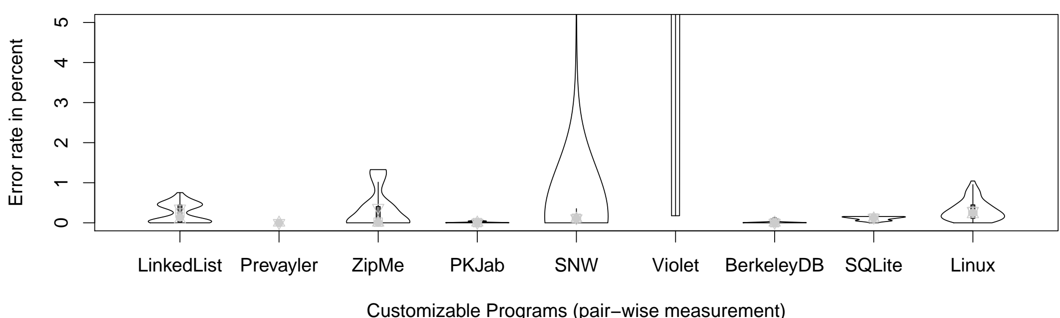


Figure 7.5.: Error-rate distribution of predicting footprint with pair-wise measurement.

- Measurement effort of pair-wise measurement is larger than for interaction-wise measurement.

Next, we evaluate if our observations hold for a further non-functional property: main-memory consumption.

7.3.3. Main-Memory Consumption

For the selected programs in the experiment of main-memory consumption, we are either no domain experts or we have no access to them. Since feature interactions are not necessarily caused by functional interactions, we cannot use source-code analysis as we did for the footprint experiment. Hence, most of the programs are black-box programs, for which we know only how they can be customized. Thus, we cannot use interaction-wise measurement here, but only pair-wise measurement.

The experimental setup as well as the experimental procedure are equal to the one for feature-wise measurement of Section 6.2. That is, we use the same programs and the same benchmarks. Hence, we present the results next.

Program	Appr.	Effort: # Measurements		Error Rate (in %): Distribution	Mean±Std	Median
LinkedList	FW	11	2 %		0.9± 0.9	0.51
	IW	13	3 %		0.7± 0.7	0.43
	PW	88	18 %		0.2± 0.2	0.13
	BF	492	100 %	−6 −4 −2 0 2 4 6	---	
Prevayler	FW	5	21 %		0.1± 0.1	0.03
	IW	7	29 %		0± 0	0.0
	PW	17	70 %		0± 0	0.0
	BF	24	100 %	−6 −4 −2 0 2 4 6	---	
ZipMe	FW	8	8 %		0.6± 0.6	0.4
	IW	10	10 %		0.2± 0.3	0.0
	PW	21	20 %		0.3± 0.5	0.01
	BF	104	100 %	−6 −4 −2 0 2 4 6	---	
PKJab	FW	8	11 %		0± 0	0.0
	IW	8	11 %		0± 0	0.0
	PW	36	50 %		0± 0	0.0
	BF	72	100 %	−6 −4 −2 0 2 4 6	---	
SNW	FW	26	1 %		0.5± 1	0.23
	IW	34	1 %		0.3± 0.5	0.16
	PW	252	8 %		0.2± 0.6	0.11
	BF	3 240	100 %	−6 −4 −2 0 2 4 6	---	
Violet	FW	80	0 %		186.7± 34.4	209.27
	IW	2115	0 %		0± 0	0.0
	PW	5229	0 %		722.5± 362	997.43
	BF	10^{20}	10 %)	−6 −4 −2 0 2 4 6	---	
Berkeley	FW	9	4 %		1.9± 2.2	0.8
	IW	15	6 %		0.5± 0.8	0.2
	PW	33	13 %		0± 0	0.0
	BF	256	100 %	−6 −4 −2 0 2 4 6	---	
SQLite	FW	85	0 %		0± 0	0.03
	IW	146	0 %		0± 0	0.0
	PW	3306	0 %		0.1± 0	0.12
	BF	10^{23}	100 %	−6 −4 −2 0 2 4 6	---	
Linux	FW	25	0 %		0.4± 0.3	0.2
	IW	207	0 %		0.4± 0.3	0.31
	PW	326	0 %		0.3± 0.2	0.25
	BF	$33{\cdot}10^{23}$	100 %	−6 −4 −2 0 2 4 6	---	

Table 7.1.: Error rates in percent of footprint predictions using the approaches (Appr.): feature-wise (FW), interaction-wise (IW), pair-wise (PW), brute force (BF). Mean: mean error rate, Std: standard deviation.

Results. In Table 7.2, we summarize the error rates of predicting main-memory consumption for the sample programs. Additionally, we show the results of feature-wise measurement and the brute-force approach for comparison. Using the pair-wise approach (Q2), we see that the error rate usually decreases (with the exception of Wget and LLVM, which we discuss in the following): The average error rate is 11 % for all programs and 0.9 % without Wget and LLVM. This means that our assumption, that measuring how each pair of features affects a non-functional property improves accuracy of predictions by 2.6 % when compared to feature-wise measurement. However, this improvement requires additional measurements. These additional measurements tend to increase by up to 6 % for programs with limited variability (e.g., Curl and LLVM) and by less than one percent for SQLite. Furthermore, we need to measure less than 10 % of all variants to determine all pair-wise interactions. For example, in the case of SQLite, we need to measure only 317 variants which is 0.008 % of all possible variants.

		Effort		Error Rate (in %)		
Program	Appr.	# Measurements		Distribution	Mean±Std	Median
Curl	FW	11	1 %		0.4± 0.6	0.16
	PW	56	7 %		0.1± 0.2	0.06
	BF	768	100 %	-10 0 10 20 30	---	
LLVM	FW	11	1 %		28.8± 26.3	14.18
	PW	56	6 %		43.1± 59.5	12.85
	BF	1 024	100 %	-10 0 10 20 30	---	
x264	FW	12	1 %		27.9± 47.8	0.03
	PW	66	6 %		1.9± 5	0.0
	BF	1 152	100 %	-10 0 10 20 30	---	
Wget	FW	14	0 %		18.6± 24.9	9.25
	PW	91	2 %		20.6± 50.6	12.85
	BF	5 120	100 %	-10 0 10 20 30	---	
BerkeleyDB	FW	15	1 %		1.4± 1.3	1.03
	PW	98	4 %		1.4± 1.3	0.97
	BF	2 560	100 %	-10 0 10 20 30	---	
SQLite	FW	26	0 %		4.4± 3.8	3.4
	PW	317	0 %		0.3± 1.8	0.0
	BF	3 932 160	100 %	-10 0 10 20 30	---	

Table 7.2.: Error rates in percent of predicting main-memory consumption of all customizable programs using the approaches (Appr.): feature-wise (FW), pair-wise (PW), brute force (BF). Mean: arithmetic mean error rate of predictions, Std: standard deviation of predictions.

In Figure 7.6, we show the error-rate distributions for pair-wise measurement. We see that LLVM and Wget still exhibit high error rates, but error rates of x264 could

be reduced from 27.9 %, on average, to 1.9%. A closer look at the distribution of Wget's error rates shows that the mean (thick line) of all predictions is close to zero (see box plots in Table 7.2). That is, a large number of predictions are still accurate (even more accurate than feature-wise measurement). However, there are a few outlier predictions that show a very high error rate (i.e., over 75 %), which leads to a high arithmetic mean error rate. We can easily see this case for x264 with feature-wise measurement when considering the median (cf. 7.2). The median is 0.03, which means that half of our predictions have an error rate of nearly zero percent. That is, most of our predictions are perfect.

However, we further use the arithmetic mean for evaluation, although in statistics, the median usage is standard. Although our results are clearly better with the median, we would also hide the interesting cases (i.e., the outliers). Hence, we stick with the arithmetic mean for discussion. As a result, at least partially, determining pair-wise feature interactions improves prediction accuracy. However, there are still inaccurate predictions, which are caused by higher-order feature interactions.

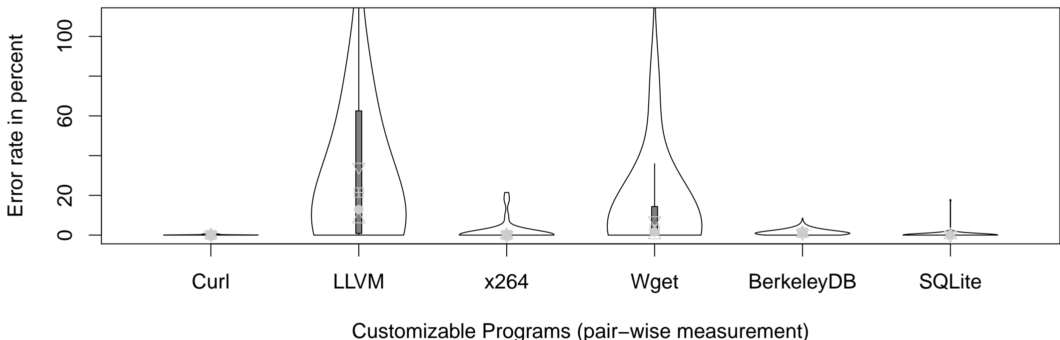


Figure 7.6.: Error rates in percent of all customizable programs for main-memory consumption using pair-wise measurement.

The key results are as follows:

- Identifying interactions based on domain knowledge is challenging for main-memory consumption, because expert knowledge is needed to find interactions that are not directly visible in the source code.
- Pair-wise measurement partly improved and partly worsened prediction accuracy, which we discuss next.
- Measurement effort of pair-wise measurement is still low in terms of absolute number of measurements, especially for highly customizable programs, such as SQLite.

Next, we compare interaction-wise and pair-wise measurement with the results of the two experiments to answer research question 3.

7.3.4. Discussion and Analysis of Measurement Approaches

In this section, we answer our third research question (i.e., how the three measurement approaches perform against each other). To this end, we compare prediction accuracy, measurement effort, and applicability. For each of these topics, we present our findings with a concrete example first and summarize and generalize our findings at the end to give recommendations which one to use in practice.

Comparing Prediction Accuracy

To compare prediction accuracy of the three measurement approaches, we use two examples at which the difference of these approaches become clear: Violet and LLVM. We use these examples, because they represent interesting and challenging cases. In our summary, we generalize these findings to all observed customizable programs.

Example: Violet. For Violet, we observed the largest error rates for feature-wise (left violin in Figure 7.7) and pair-wise measurement (right violin in Figure 7.7). We investigate the reason for these error rates later in Section 7.4 when analyzing feature interactions. Here, it is important that by having the knowledge only about the mapping from features to implementation units, we reach a error rate of below 1 %, on average. We achieved this by measuring less than half of the variants we needed to quantify all pair-wise interactions (2115 vs. 5229 measurements). This example demonstrates that interaction-wise measurement can outperform pair-wise measurement, especially when there are higher-order feature interactions. Since we quantified the influence of all pair-wise feature interactions, only higher-order feature interactions can cause these inaccuracies in predictions.

Example: LLVM. In Violet, we could improve prediction accuracy significantly with domain knowledge. Unfortunately, we do not have the necessary knowledge for LLVM to show how this would affect accuracy. Nevertheless, we tried to identify feature interactions.

We believe higher-order feature interactions are very plausible for LLVM; we hypothesize that they can be explained as follows. Each LLVM feature toggles a different optimization phase during compilation; each optimization might act differently on a code fragment, depending on how and whether previous optimizations have transformed it. For instance, function inlining enables other optimizations to operate on longer code fragments; depending on the size of code fragments and on the possibilities for optimizations, further optimization might be executed.

To evaluate our assumptions, we used the documentation of LLVM to manually define feature interactions. Overall, we measured 129 variants of LLVM, which is 12.6 % percent of all variants. Although we are no domain experts, our predictions

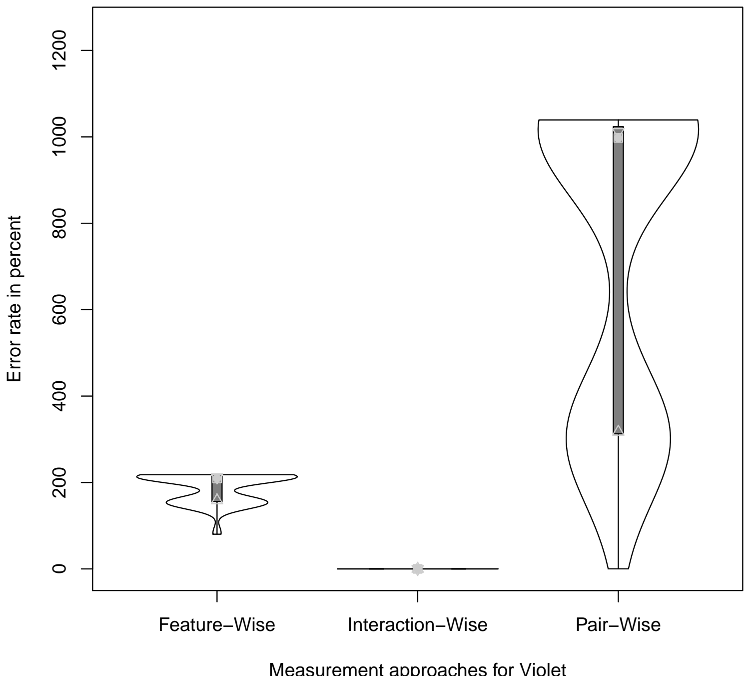


Figure 7.7.: Error-rate distribution of Violet for the three measurement approaches.

significantly improved to a mean error rate of 11 %, which is an improvement of 22 % compared to pair-wise measurement and 14 % compared to feature-wise measurement (see Figure 7.8). This means that even with little domain knowledge, it is possible to achieve considerable improvements of prediction accuracy.

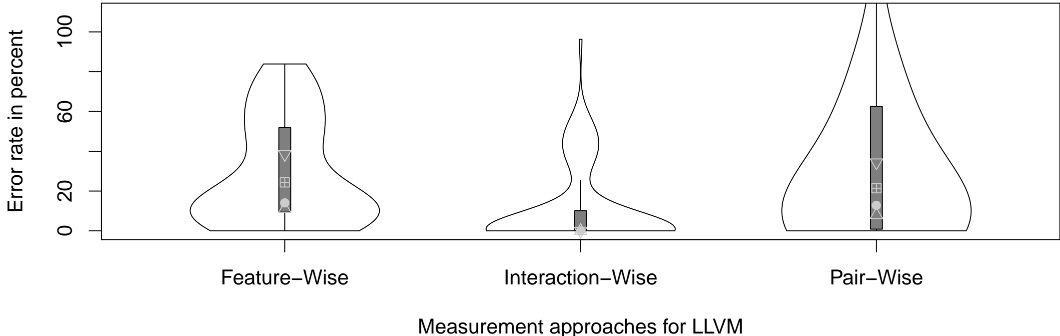


Figure 7.8.: Error-rate distributions of different measurement approaches for LLVM.

Comparing Measurement Effort.

Regarding measurement effort, we use Berkeley DB as a concrete example, because we can easily analyze at which point pair-wise measurement wastes resources. Similar to prediction accuracy, we summarize and generalize our findings at the end of this section.

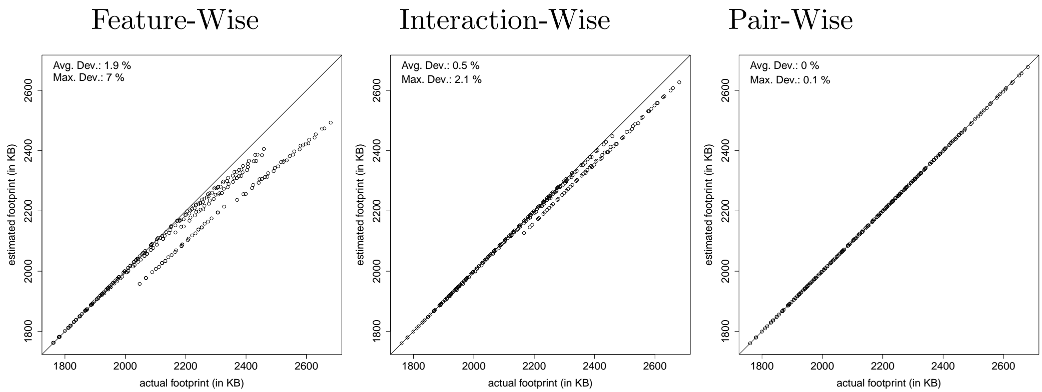


Figure 7.9.: Measured and predicted footprint in KB of Berkeley DB (compiled as static link library) using different approaches.

Example: Berkeley DB. In Figure 7.9, we show the results of our different measurement approaches for Berkeley DB. We use Q-Q plots to visualize the trend at which predictions become more inaccurate. Each dot represents a single configuration, for which we compare predicted footprint (y-axis) against actual footprint (x-axis). If the prediction is correct, the dot lies on the diagonal line. Since each feature increases the footprint of a variant, configurations with more features are located more right.

Although we measured only 9 variants of Berkeley DB with feature-wise measurement, we achieve a mean error rate of about 1.9 % only for all 256 variants. We often predict a footprint that is too low, because we did not determine feature interactions that include additional code in a variant. Hence, for variants containing an increasing number of features that depend on each other, the error rate increases (see left Q-Q plot in Figure 7.9). We see that the distance from a dot to the diagonal increases when the size of a variant increases, too. This means that there is additional code included with more features that we did not consider.

With interaction-wise measurement, the average error rate is reduced to 0.5 %. Thus, by measuring only 15 variants of Berkeley DB, we can predict the footprint of all 256 variants with nearly perfect accuracy (99.5 % on average). But the Q-Q plot in the center of Figure 7.9 clearly shows that we missed only a single interaction. That is, with only 16 measurements, we would have found all interactions of Berkeley DB.

Although, pair-wise measurement eliminated errors entirely (maximum error rate of 0.1%), it required 37 measurements. This means that we performed more than twice the measurements that would actually be needed.

Hence, this example makes clear that pair-wise measurement can waste resources to gain only a slight improvement of measurement accuracy. For Violet, we described previously that we required only half of the measurements with the interaction-wise approach compared to the pair-wise approach. This observation holds for all other programs: We measured 2 547 variants with interaction-wise measurement and 9 308 with pair-wise measurement. Hence, we require nearly four times more measurements with the pair-wise approach. In relation to the overall number of variants, the difference is 13.4 %. That is, 13.4 % of the whole population of all used customizable programs must be additionally measured when using pair-wise measurements compared to interaction-wise measurement.

Applicability

The benefit of using interaction-wise measurement compared to pair-wise measurement was clear up to know. However, when comparing applicability of both approaches, the picture changes. Of course, we can use interaction-wise measurements only when we have domain knowledge or the source code. This is already a key limitation compared to pair-wise measurement, which can be used also for black-box programs. However, we additionally found a further limitation of interaction-wise measurement: expectation of the kind of feature interactions. We make this finding clear with the example Linux kernel.

Example: Linux kernel. For Linux, we expected large error rates regarding the 100 randomly generated variants, because many Linux features affect the size of other features. We were surprised that we still achieved a quite precise prediction even with feature-wise measurement, partially, because the features had a weaker effect than expected. We slightly improved the accuracy of the interaction-wise approach, because we defined feature interactions between more than two features (i.e., we defined an interaction between every feature and the features *OPTIMIZE_INLINING* and *CC_OTPIMIZE_FOR_SIZE*).

In Figure 7.10, we show Q-Q plots of all measurement approaches for the Linux kernel. With feature-wise measurement, we tend to over *and* under estimate a variant's footprint, because there is code reused *and* additional code introduced. With interaction-wise measurement, the domain expert defined only those interactions that introduce additional code and therefore increase the size of a variant. The domain expert did not define an interaction that decreases the size of a variant, because he was either not aware of this case or could not determine for which combination of these features this case occurs. This is an important insight, because it indicates

that certain types of interactions are easier to find than others. For instance, a simple call to a print method in Java in two different features already results in a code reuse, because code from the Java standard package is included in the program. This is not obvious and may be difficult and costly to detect. The result is that all our interaction-wise predictions are over estimates (see middle of Figure 7.10).

Our finding is backed up by the distribution of error rates with pair-wise measurement. Here, pair-wise measurement reduced the distance of dots to the diagonal in *both* directions, because it makes no assumptions about sharing or introducing code. This indicates that there are interactions that even a domain expert cannot foresee. If such interactions would have a more severe influence on a non-functional property, interaction-wise measurement would have performed worse than pair-wise measurement. Hence, applicability of interaction-wise measurement is further restricted to the available knowledge.

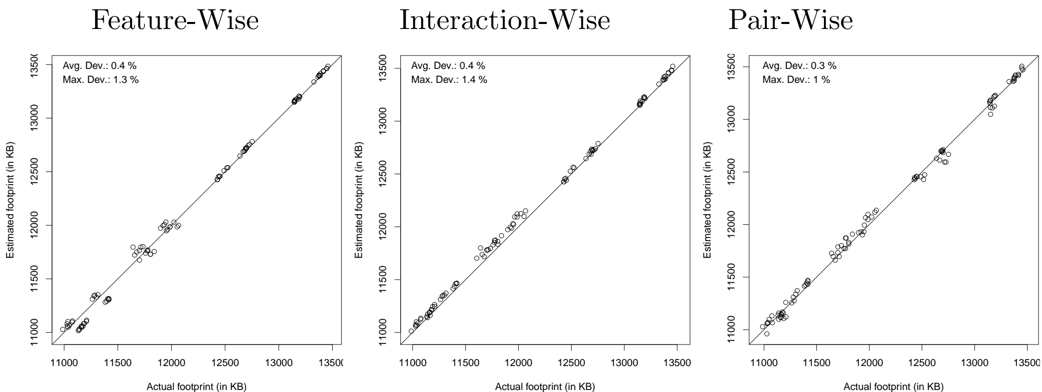


Figure 7.10.: Measured and predicted footprint in KB of Linux kernel using different approaches.

Summary.

We summarize the evaluation of interaction-wise and pair-wise measurement in Table 7.3. We answer the research questions in terms of average error rates (for Q1-Q2) and measurement effort relative to a brute-force approach (Q3). We can see that, for many sample programs, all three measurement approaches provide a high accuracy of predictions. However, interaction-wise measurement outperforms pair-wise measurement in both, prediction accuracy and measurement effort and interaction-wise measurement gives clearly more accurate predictions than feature-wise measurement.

For pair-wise measurement, we observe that higher-order feature interactions play a crucial role for the accuracy, because we determined all first-order interactions, but still got high error rates. This is why our predictions for Violet, LLVM, and Wget

	Q1&Q2: Error rate			Q3: Effort		
Experiment	FW	IW	PW	FW	IW	PW
Footprint	21.3 %	0.1%	80 %	5.0 %	6.6 %	20 %
without Violet	5.5%	0.2 %	0.2 %	5.8 %	7.5 %	22 %
Main Memory	13.6 %	N/A	11 %	0.6 %	N/A	4.1 %
without LLVM & Wget	8.5 %	N/A	0.9 %	0.7 %	N/A	4.2 %

Table 7.3.: Evaluation of feature-wise (FW), interaction-wise (IW), and pair-wise (PW) measurement and overview of research questions and experiment results. Effort means measurement effort. Q1-Q3 refer to the research questions given in the experiment description.

are inaccurate; for that reason we provide the error rates with and without these programs. Table 7.3 shows that if a program has no higher-order feature interactions, pair-wise measurement has an equal accuracy as interaction-wise measurement (0.2 % vs. 0.2 % for footprint and a mean error rate of only 0.9 % for main-memory consumption), but is applicable for black-box programs.[4] This is an interesting insight, which leads to further research questions about the distribution and order of feature interactions. For this purpose, we did a more detailed analysis in Section 7.4.

Insight 1: *The number of higher-order feature interactions varies between customizable programs.*

We also want to compare measurement effort for all approaches. The effort of interaction-wise measurement is only slightly higher than for feature-wise measurement; an increase of 1.6 % relative to the number of all variants used for the footprint experiment. Since these few additional measurements improved prediction accuracy considerably, it indicates that the overall number of *relevant* feature interactions in a customizable programs is low, which is another important insight.

Insight 2: *The number of relevant feature interactions is low (below quadratic).*

Regarding pair-wise measurement, the effort is around 20 %, on average, for footprint. However, the reason for this high percentage is that the absolute number of valid configurations is low. That is, when measuring all pairs of features, we nearly measured all valid variants of programs with only few features. Since programs with a limited customizability are not our intended scenario (since we can measure all variants for these programs), important cases are SQLite, Linux, and Violet for which we need to measure less than 1 % of all variants. In the main-memory-consumption experiment, the relative number of measurements is less than 5 %, which is low, but substantially larger than for feature-wise measurement.

[4]For interaction-wise measurement, we have to know all major interactions.

Table 7.3 shows that more measurements provide more accurate predictions, with some exceptions. These exceptions, however, could be easily fixed with domain knowledge. A further result is that one of nine programs for footprint and two of six programs for main-memory consumption exhibit a high error rate. Although the number of these programs is too small to know how often such exceptions occur, it indicates that in 75 % of all programs, a simple feature-wise technique including a pair-wise measurement is sufficient and, for the rest, a more sophisticated feature-interaction-detection approach or domain knowledge is needed to improve prediction accuracy.

The key findings are as follows:

- Interaction-wise measurement outperforms feature-wise and pair-wise measurement in prediction accuracy and, partly, in measurement effort.
- Applicability of interaction-wise measurement is significantly limited compared to pair-wise measurement.
- The number of higher-order feature interactions varies between customizable programs.
- The number of feature interactions is low.

Based on this evaluation, we give recommendations of which of the approaches to use in practice.

Recommendation and Generalization.

The results have shown that feature-wise measurement is a good initial approximation of a variant's non-functional properties. However, if domain knowledge is available, we suggest to always use it, because it can significantly improve accuracy with only few additional measurements. Additionally, complex mappings or unknown feature interactions can cause large error rates making predictions less accurate.

If domain knowledge is not available, it is difficult to decide whether it is worth to measure more variants to increase prediction accuracy. For some non-functional properties, such as footprint, it might be feasible to extract information about interactions from the source code. Sometimes other sources may be available.

At this point, the measurements are often already sufficiently accurate to use them during program derivation – our initial goal –, for example, to rule out variants that obviously do not fulfill the required constraints or to determine a set of possible candidates for the optimal variant.

Finally, if domain knowledge is not available, pair-wise measurements are a good strategy to increase accuracy of predictions at cost of an increased effort for measurements (from $O(n)$ to $O(n^2)$). We recommend it only if there is either no domain knowledge available or to combine it with interaction-wise measurement, when the number of features is acceptably small. But we also recommend to verify a prediction

model that is based purely on pair-wise measurements, because our experiments have shown that there are cases in which pair-wise measurements produce heavy error rates because of higher-order feature interactions.

We illustrate the trade-off between measurement effort and prediction error rate in Figure 7.11. In general, the error rate decreases with additional measurements, but a stakeholder must be aware of the fact that too many measurements render the approach infeasible. For example, if we want to measure all 8 000 features of the Linux kernel [Lotufo et al., 2010] (we considered only 25 in our evaluation) with pair-wise measurement, we would need about 64 million measurements (which, extrapolating from our experiments, would take roughly 2 years to measure on a cluster of 1 000 computers). In contrast, feature-wise measurement requires the measurement of only about 8 000 variants (which could be realistically done in one day using a cluster of 100 computers). For our approach, balancing between desired accuracy and investment in measurements is essential.

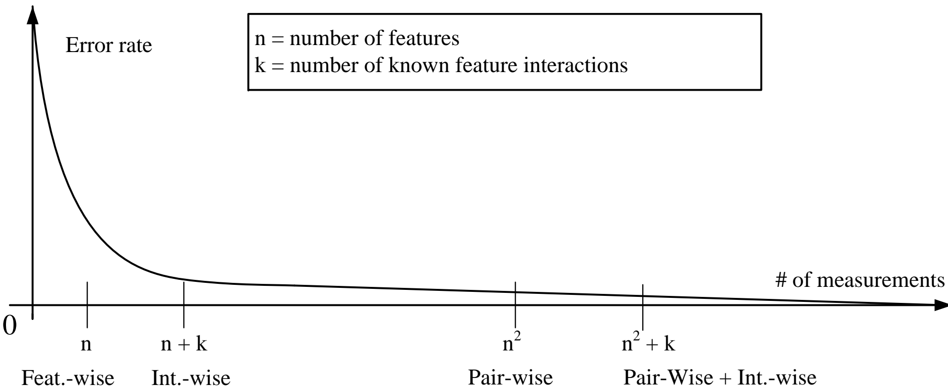


Figure 7.11.: Conceptual relation between number of measurements and error rate of predictions.

Unfortunately, the findings also show that all three approaches, feature-wise, interaction-wise, and pair-wise measurement, do not satisfy our stated goals of this thesis. Interaction-wise measurement would be a good candidate, but it does not satisfy our second requirement (R2): black-box approach. Hence, we did not find a method that allows us to produce accurate predictions with only few measurements and is applicable for black-box programs. At this point, it is clear that efficiently finding relevant feature interactions without using domain knowledge is the key to reach our goals. To follow the path of our constructive research method, we conducted a deep analysis of feature interactions with the material of this experiment. We describe this analysis in the next section, but, first, we discuss threats to validity of this experiment.

Threats to Validity of Feature-Wise Measurement (see Section 6.2.7):
Construct validity:
- Stated research questions before conducting the experiments.
- Using standard benchmarks for main-memory measurement rather than self-developed.

Conclusion validity:
- Footprint can be accurately measured without any measurement bias.
- Measuring only peak memory to determine main-memory consumption.

Internal validity:
- Using 100 sample variants for highly customizable programs.
- Measurement bias for main-memory consumption were below 1 %.

External validity:
- Measurement of two different non-functional properties.
- Measurement of many, very different real-world programs with varying implementation techniques, languages, etc.

Figure 7.12.: Recap: Threats to validity of feature-wise measurement.

7.3.5. Threats to Validity

Since we used the same experimental setup as for feature-wise measurement (see Figure 7.12 for a short summary), we discuss only an additional threat to validity.

Internal Threat to Validity. In our footprint evaluation, we observed high error rates for Violet. These error rates were caused by complex mappings between features and implementation units. As we have shown, when this mapping is known, we can easily compute which configurations have to be additionally measured to achieve accurate predictions. Hence, the real problem is when we have a complex mapping between features and implementation units and this mapping is not known. Thus, the question is: Is this the rule or exception in real-world applications? We believe that unknown complex mappings are the exception. Considering software product lines and customizable programs implemented with preprocessors, we almost always have a direct mapping between features and preprocessor flags, because we have to know these flags in order to generate the desired program. Since most of the customizable real-world programs today are based on preprocessors, this is already a strong argument.

Furthermore, new programming paradigms, such as aspect-oriented and feature-oriented programming aim at a direct one-to-one mapping between features and implementation units. The general idea is to model features at the domain level and implement a feature accordingly Apel and Kästner [2009]. Although Violet is implemented with feature-oriented programming, the mapping problem is an improper modeling or implementation of the functionality. Finally, complex mappings also

occur for component-based software development. Here, however, we always need the mapping between features and components to be able to assemble this components Pour [1998]. Hence, we believe that there is in most cases either the mapping known to be able to generate a program or the mapping is one-to-one.

7.4. Analysis of Feature Interactions

The previous evaluation has shown satisfying, but also unsatisfying results regarding accuracy and feasibility of our approaches. On the one hand, interaction-wise measurement enables accurate predictions, but on the other hand, it is not applicable for black-box programs. Also for pair-wise measurement, the picture is unclear. For most of the programs, our predictions are accurate, but for three others (Violet, LLVM, and Wget), we observed error rates that are higher than feature-wise measurement.

We identified that feature interactions play a crucial role for the accuracy of predictions. We already gained two insights from the previous evaluation: Higher-order feature interactions exist not always, but in some programs (Violet, LLVM, and Wget). Furthermore, there are only a limited number of higher-order feature interactions. This lead us to the conclusion that we have to further analyze the nature of feature interaction to be able to develop an automated feature-interaction detection for black-box programs. To this end, we decided to use the existing data we gained from the previous experiment to answer our new research questions.

- **Q4:** What is the distribution of feature interactions among all features?
- **Q5:** Do all features interact or only few?
- **Q6:** At which order are most of the detected feature interactions?
- **Q7:** Are there any reoccurring patterns recognizable that help us detecting relevant feature interactions without the need of domain knowledge?

We defined research questions Q4 and Q5 based on our Insight 2 (there are only few feature interactions). We wanted to know whether these few interactions are distributed among all features or only among few. Furthermore, we wanted to identify the influence of higher-order feature interactions on non-functional properties (Q6 and Q7), because pair-wise measurement showed that there are some cases at which determining all first-order feature interactions is not sufficient to produce accurate predictions (Insight 1).

Again, we use the same experimental results as in the previous experiment in Section 7.3. Our analysis gives us additional insights from which we develop heuristics that help us detecting feature interactions without domain knowledge, which is one of our main contributions in this thesis.

Research Question 4: Distribution of Feature Interactions.

With research question 4, we are interested in the distribution of feature interactions. To answer this question, we created several graphs in which we connected features with a line for which we found a relevant feature interaction. In Figure 7.13, we show such a graph for SQLite created with the data of the footprint experiment.[5] We visualize all pair-wise interactions using a line for which we determined a non-zero value. The important part of this figure is the pattern of the edges. The picture clearly shows that two features (*SQLITE_OMIT_COMPLETE* and *SQLITE_OMIT_PRAMGA*) interact with many other features and that most features interact only with these two and maybe two other features. Hence, it seems that there are features that are more critical to the prediction of non-functional properties than others. A feature that interacts with many other features raises the possibility of higher-order feature interactions.

The distribution of feature interactions is, therefore, not uniform, but very concentrated on some spots. We call these features *hot-spot features*, because they interact with many other features and form a hot spot of interactions. This graphical analysis answers our research question:

Insight 3: *The distribution of feature interactions among features of a customizable program is not uniform. Instead there are few hot-spot features that interact with many other features.*

Research Question 5: Occurrence of Feature Interactions.

To answer whether all features interact or only few (research question 5), we draw similar graphs as previously, but with a different layout. What we cannot easily identify in Figure 7.13 becomes clear with the layout of Figure 7.14. Again, dots represent features and lines represent pair-wise interactions. Many features do not interact at all. In this layout, features that interact are placed towards the center of the figure, and features that do interact are placed at the border. Again, only the pattern is important and not the feature names. We see that a large number of features do not interact with any other features. This means that we do not need to determine feature-interaction terms for a large number of features, which significantly reduces the number of measurements to provide accurate predictions.

This picture backs up our finding of the increased measurement effort of pair-wise measurement compared to interaction-wise measurement. In Section 7.3.4, we found that for Berkeley DB, 16 measurements would be necessary to find all interactions, but pair-wise measurements required 37. This is because the pair-wise approach determined the influence of feature-interaction terms of features that never interact. Hence, an import insight, which also follows from Insight 2 and 3 (the number

[5]The graphs of the remaining programs look similar.

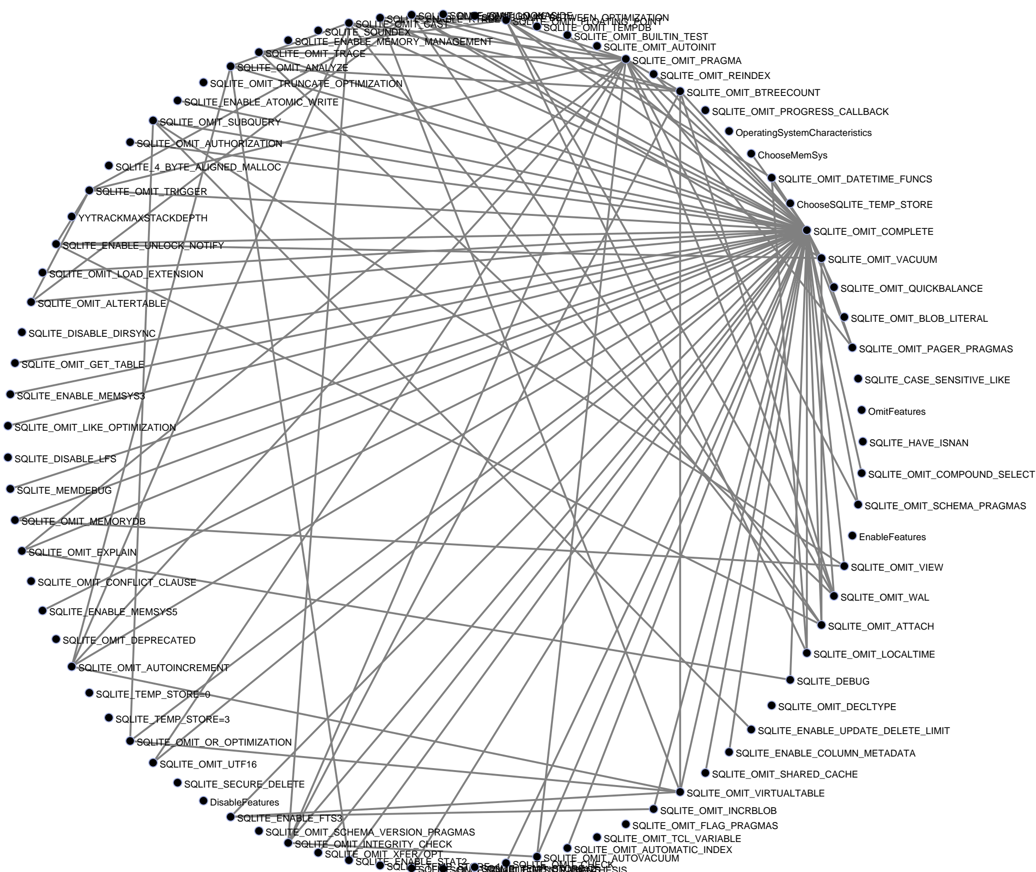


Figure 7.13.: Distribution of pair-wise interactions of SQLite's features (footprint experiment).

of feature interactions is low and their distribution is not uniform), is that many features do not interact.

Insight 4: *Many features do not interact at all.*

Research Question 6: Order of Feature Interactions.

The order of feature interactions has an influence on the number of variants we have to measure to determine this interaction. Also, the order of an interaction determines whether measurement approaches, such as pair-wise measurement, exhibit high error rates. Hence, it is important to know how many feature interactions occur at which order. That is, whether most of the interactions are of a higher order or only between two features. This finding would help to detect feature interactions more efficiently,

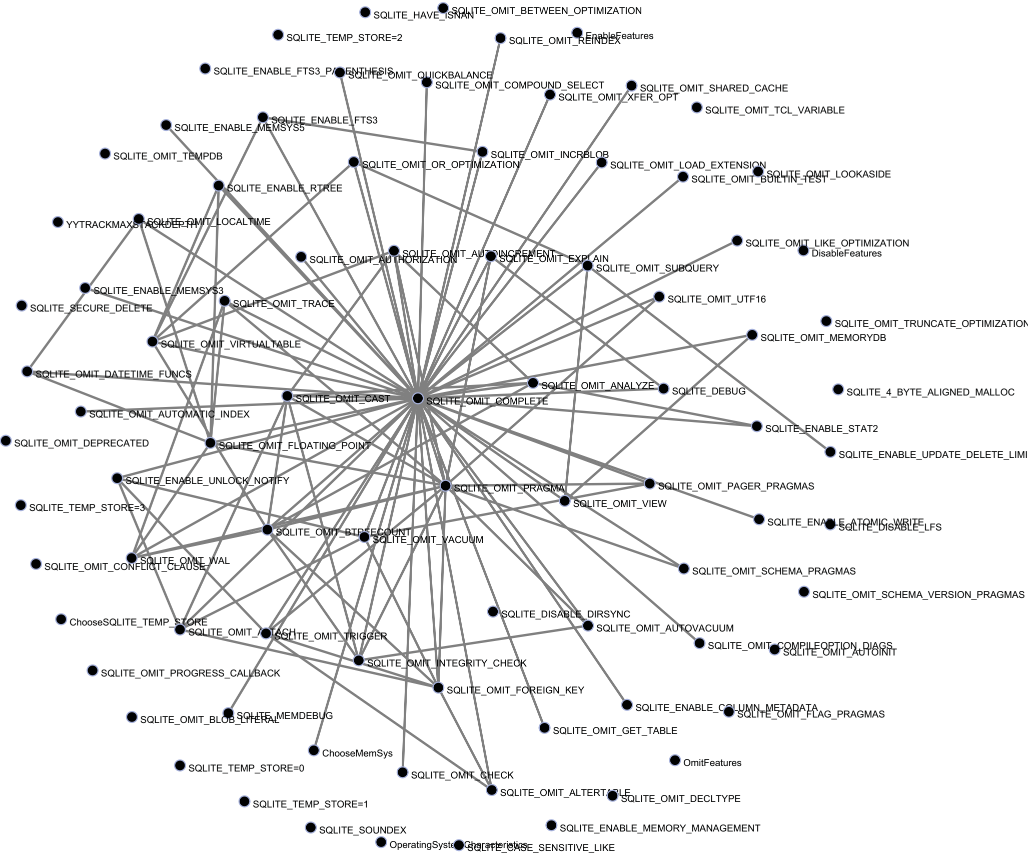


Figure 7.14.: A large number of features in SQLite do not interact (footprint experiment).

because we can search for feature combinations at which relevant interactions are more likely.

Since we do not know all interactions for all used programs, we can give only a qualitative analysis. In eight out of nine customizable programs in the footprint experiments and in four out of six programs in the main-memory-consumption experiment, pair-wise measurement produced the highest accuracy. This means that, at least for our experiments, in three out of four programs, first-order interactions exist with no or only few higher-order interactions. For LLVM and Violet, we can compare interaction-wise with pair-wise measurement regarding the order of feature interactions. For both programs, first-order interactions had the largest number compared to the number of each interaction at each individual order. Furthermore, we observed that to each higher-order interaction, at least two first-order interactions exist with the participating features. This leads to the following insight:

Insight 5: *Most feature interactions are first-order (pair-wise) feature interactions.*

Research Question 7: Patterns of Feature Interactions.

With our last research question, we want to investigate whether there are certain patterns in the distribution of feature interactions. That is, whether there are first-order interactions from which we can infer the existence of higher-order interactions. If there are patterns, we can develop according heuristics to find feature interactions with fewer measurements. We use Violet as an example, because the error rate of below 1 % with interaction-wise measurement indicates that we actually determined all relevant feature interactions, including higher-order interactions.

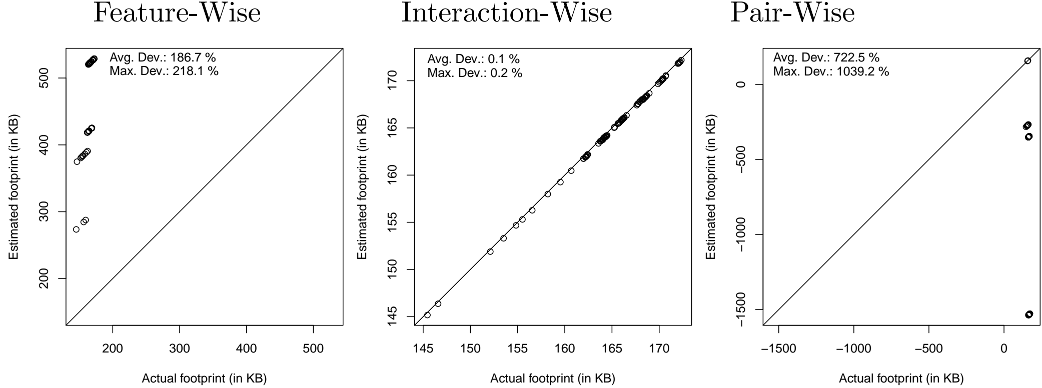


Figure 7.15.: Measured and predicted footprint in KB of Violet using different measurement approaches.

The reason for Violet's high error rates is a complex mapping between (some) features and implementation units. That is, an individual feature may map to multiple implementation units and a single implementation unit may be required by multiple features. Hence, when measuring such a feature, the corresponding variant contains several implementation units that are also present when measuring another feature's variant. Therefore, predicting the footprint of a variant that includes multiple features with an overlapping set of implementation units is inaccurate, because we consider the footprint of the implementation unit multiple times. This is why we see an upper shift in the feature-wise plot in Figure 7.15. It means that we usually predict a too high footprint.

Predictions of pair-wise measurement are even worse, because more than two features map to the same code. When predicting a variant containing three features, we aggregate three times the first-order feature-interaction terms, though only two times would be correct. If more features interact, the inaccuracy quadratically increases, which is an interesting insight about feature interactions. That is, when there are

multiple pair-wise interactions with the same set of features, we may account for too many interactions. The increasing inaccuracies with an increasing usage of first-order interaction terms in a prediction are shown as a down shift of the dots in the pair-wise plot in Figure 7.15. This is why we predict even a negative footprint of over 1,500 KB (-1,036 % worst-case error rate). By determining higher-order interactions, we can correct the predictions. This indicates the existence of a reoccurring pattern.

To illustrate the cause of the high error rates, we show an excerpt of Violet's product-line model in Figure 7.16. We observe a complex mapping between features and implementation units. Using feature-wise measurement, we determine the influence of all feature terms on footprint (indicated with the annotated values). However, what we actually measure is not the feature, but the corresponding implementation units. Since several features use the same implementation units, we account for these shared implementation units multiple times. That is, if we predict a variant with all features (*Class Diagram (CD)*, *Object Diagram (OD)*, *UseCase Diagram (UD)*, and *State Diagram (SD)*), we sum up four times the size of *Diagram Module*, three times the size of *Window*, and two times the size of *I/O Operations*.

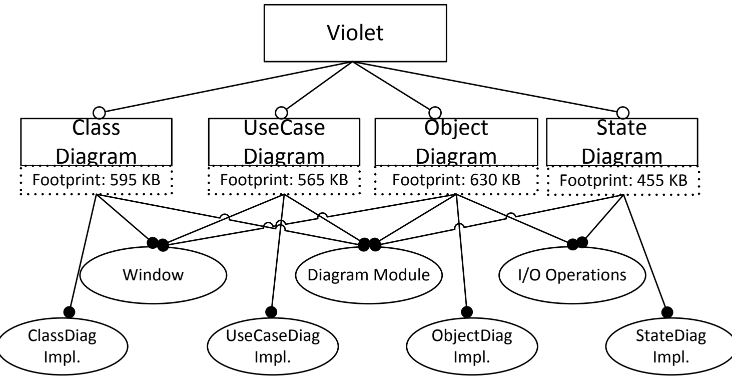


Figure 7.16.: Partial product-line model of Violet including the mapping between features and implementation units.

To make this problem clear, we show a table that lists all determined feature and feature-interaction terms using pair-wise measurement:

Feature Terms	Π_F in KB	Interaction Terms	Π_F in KB
CD	595	*CD#OD*	-415
OD	630	*CD#UD*	-415
UD	565	*CD#SD*	-205
SD	455	*OD#UD*	-415
–	–	*OD#SD*	-325
–	–	*UD#SD*	-205

In this table, we see that all feature terms have a positive footprint, because selecting a single feature causes the compilation of multiple implementation units. We also

see that all pair-wise interactions have a negative footprint. The reason is that, when we determine the size of a pair-wise interaction, we measure a variant with two features and compare it against the prediction, which accounts for the size of several implementation units twice. For example, when we want to determine feature interaction $CD\#OD$, we measure a variant with features CD and OD and compare it to the corresponding prediction:

$$\begin{aligned} \Pi_F(CD\#OD &= \Pi_F(CD \times OD) - \Pi_F(CD) - \Pi_F(OD) \\ &= 810\,KB - 595\,KB - 630\,KB \\ &= -415\,KB \end{aligned}$$

The value of - 415 KB is exactly the size of the shared implementation units by the two features, i.e., $\Pi_F(\tau(CD \cdot OD))$. All other pair-wise interactions are determined similarly. The problem is that, when we want to predict a program with more than two features, we include too many pair-wise interactions, leading to a too high subtraction. For example, if we predict a program P that contains all features, we obtain the following false prediction:

$$\begin{aligned} \Pi_F(P) &= \Pi_F(CD \times OD \times UD \times SD) \\ &= \Pi_F(CD) + \Pi_F(OD) + \Pi_F(UD) + \Pi_F(SD) + \\ &\quad \Pi_F(CD\#UD) + \Pi_F(CD\#OD) + \Pi_F(CD\#SD) + \\ &\quad \Pi_F(OD\#UD) + \Pi_F(OD\#SD) + \Pi_F(UD\#SD) \\ &= 595\,KB + 630\,KB + 565\,KB + 455\,KB - \\ &\quad 415\,KB - 415\,KB - 205\,KB - 415\,KB - 325\,KB - 205\,KB \\ &= 265\,KB \end{aligned}$$

We obtain a result that is smaller than the minimal measured variant. This example shows that by omitting higher-order interaction terms (e.g., $CD\#OD\#UD$), we get false predictions. Hence, we have to consider also higher-order feature interactions to achieve accurate predictions. This is especially the case when multiple pair-wise interactions exist with an overlapping set of features.

From this insight, we observe the following pattern: If multiple features use the same resource (e.g., implementation unit, CPU, memory, and data), they all interact pair-wise, because each of these features influence the other features. Moreover, when having these first-order interactions, also the combination of more than two features influences non-functional properties. Hence, based on the first-order interactions, we have to determine higher-order interactions. We can, therefore, **infer** which higher-order features must be additionally determined.

We visualize the identified pattern in Figure 7.17. Here, four features use the same resource (e.g., a component). Hence, all features interact with each other as denoted with the dotted lines. Moreover, when selecting three or more features, again

the predicted outcome changes, because either we have included too many first-order interactions (as in the example of Violet) or the combination of these features unexpectedly changed the non-functional property. Hence, we must determine, in addition to the six first-order interactions, also four second-order and one third-order interaction.

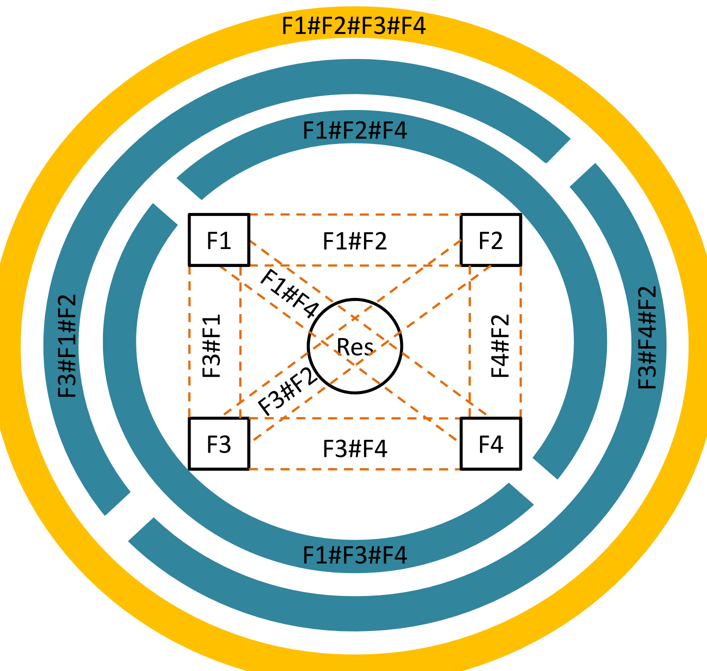


Figure 7.17.: Pattern of the occurrence of non-functional feature interactions. Orange: first-order interactions; blue: second-order interactions; yellow: third-order interaction.

We define the following recursive pattern: If two of these three interactions $\{a\#b, b\#c, a\#c\}$ are non-zero, then we expect that also the interaction $a\#b\#c$ is non-zero, i.e., has an influence on a non-functional property. Recursively, if two of these three interactions $\{a\#b\#c, a\#c\#d, b\#c\#d\}$ are non-zero, then we expect that also the interaction $a\#b\#c\#d$ is non-zero and so on. This pattern can even connect multiple interaction "bulks". That is, when we have two groups of features that both use their own resource (i.e., twice the circle of Figure 7.17), but in which at least one feature depends on both resources (e.g., a hot-spot features), we can determine a higher-order interaction that includes features of both groups. Also note that this pattern does not require any source code or domain knowledge. It needs only to know which feature interactions are non-zero. Hence, it is applicable for black-box programs.

Insight 6: *Feature interactions can occur in patterns when multiple features access or use the same resource. This pattern suggests how feature interactions distribute to higher orders.*

7.5. Extended Problem Statement

From the previous evaluation and analysis, we obtain six important insights that are relevant not only for a new feature-interaction-detection approach, but also for the whole feature-interaction research community. That is, we produced empirical data giving new indications regarding the distribution of non-functional feature interactions for the first time. Based on these results, we again extend our requirements:

- **(R1)** Accurate predictions of non-functional properties.
- **(R2)** Black-box approach.
- **(R3)** Determining the influence of individual features on non-functional properties.
- **(R4)** Efficient detection of feature interactions.
- **(R5)** *Identifying interacting and non-interacting features.*
- **(R6)** *Identifying hot-spot features.*
- **(R7)** *Infer higher-order from detected first-order feature interactions.*

The new requirements (R5-R7) are based on our findings. We expect that an accurate, efficient, and general applicable prediction approach has to meet all these requirements. In the next chapter, we propose an approach that incorporates these requirements and evaluate it for the non-functional property performance – a property for which we observed the highest error rates with feature-wise measurement.

7.6. Summary

We introduced feature interaction in this chapter. We explained how to measure feature interactions to compute feature-interaction terms for our prediction model. We outlined different causes of feature interactions (e.g., shared resources) and discussed different orders of interactions. We proposed two approaches of identifying interactions: interaction-wise and pair-wise measurement.

For interaction-wise measurement, we use domain knowledge or program analysis to manually identify feature combinations that may cause feature interactions. For pair-wise measurement, we assume that each pair of feature interacts and measure the corresponding feature combination to quantify the influence of these interactions.

- **Insight 1:** *The number of higher-order feature interactions vary between customizable programs.*
- **Insight 2:** *The number of relevant feature interactions is low (below quadratic).*
- **Insight 3:** *The distribution of feature interactions among features of a customizable program is not uniform. Instead there are few hot-spot features that interact with many other features.*
- **Insight 4:** *Many features do not interact at all.*
- **Insight 5:** *Most feature interactions are first-order (pair-wise) feature interactions.*
- **Insight 6:** *Feature interactions can occur in patterns when multiple features access or use the same resource. This pattern describes how feature interactions distribute to higher orders.*

Figure 7.18.: Insights regarding the nature of feature interactions.

We evaluated both approaches with the non-functional properties main-memory consumption and footprint. We used the same programs as for feature-wise measurement to foster comparison the different approaches in terms of accuracy and measurement effort. The results are: Interaction-wise measurement has an average accuracy of 99.9 % for footprint. Pair-wise measurement has an average accuracy of 99.8 % for footprint and 99.1 % for main-memory consumption, when no higher-order interactions are present. In the presence of higher-order interactions, the accuracy drops to 20 % (footprint) and 89 % (main-memory consumption), on average. This low accuracy is, however, predominated by few programs with overproportionally high error rates. We discussed the results and extended our problem statement, such that we need an automated feature-interaction detection to achieve always good predictions with few measurements and to be applicable for black-box programs. We present the key insights of our evaluation in Figure 7.18.

We use our observations to develop an automated feature-interaction detection, which we present next.

8. Automated Feature-Interaction Detection

This chapter shares material with the following paper:

- "Predicting Performance via Automated Feature-Interaction Detection." in *ICSE'12* [Siegmund et al., 2012a],

Based on the additional requirements that we specified in the previous chapter, we propose an automated feature-interaction detection approach. Our approach satisfies all thesis goals:

- Quantification of the influence of features on non-functional properties,
- Detection of feature interactions,
- Black-box approach without the need of domain knowledge or source code,
- Scalable measurement effort using heuristics instead of an exhaustive search, and
- Precise predictions, which we validate in an evaluation using performance.

8.1. Introducing Deltas

In the following, we introduce the concept of *deltas*. Deltas integrate in our prediction model in such a way that they comprise multiple terms into a single value. That is, a delta represents the influence of a feature and/or multiple feature interactions on a non-functional property. Unlike feature-wise measurement and the preceding interaction-detection approaches, deltas have the following benefits:

1. Deltas are independent of the relationships in a feature model (i.e., there are no special cases).

2. Deltas document with which configuration a feature was measured and therefore its influence determined.

3. Deltas comprise multiple terms into a single value to reduce computation effort.

The evaluation of pair-wise measurement in the previous chapter showed that we cannot measure in a brute-force fashion all n-wise interactions, because this does not scale. The question is: Which terms do we have to determine? From Equation 4.9, we know that a variant of n features yields an exponential number of terms (see

Recap: Equations of the Prediction Model (see Section 4.2):

$$\begin{aligned}\Pi(P) &= \Pi(a \times b \times c)\\ &= \Pi(a \cdot b \cdot c \cdot a\#b \cdot a\#c \cdot b\#c \cdot a\#b\#c)\\ &= \Pi(a) + \Pi(b) + \Pi(c) + \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c) \quad (4.9)\end{aligned}$$

Figure 8.1.: Recap: Selecting three features requires the approximation of seven terms.

Figure 8.1). We cannot compute or measure a value for each term, as this is infeasible for anything beyond programs with few features.

Furthermore, our prediction model in Equation 4.9 assumes that we can measure each feature in isolation. This is not always possible (as we have already shown in Chapter 6 for feature-wise measurement). We avoid both problems (i.e., exponential number of terms and individual measurement of terms) by composing multiple terms that cannot be separately measured as a single term, called a *delta*. Given a *base configuration* C, we compute the impact of a feature a on C's non-functional property as the property delta induced by feature a:

$$\Delta(a, C) = \Pi(a \times C) - \Pi(C) \quad (8.1)$$

Using Equations 8.1 and 4.9, we can describe the terms that a delta comprises as follows:

$$\begin{aligned}\Delta(a, C) &= \Pi(a \times C) - \Pi(C) \quad // \ (8.1)\\ &= \Pi(a\#C) + \Pi(a) + \Pi(C) - \Pi(C) \quad // \ (4.1)\\ &= \Pi(a\#C) + \Pi(a) \quad (8.2)\end{aligned}$$

That is, $\Delta(a, C)$ is the contribution of a on a non-functional property by itself *plus* the contributions of a's interaction with all terms in C on a non-functional property. If C is the empty set, then $\Delta(a, C) = \Pi(a)$, which means that a's delta maps directly to the term of a. Furthermore, if C is a variant of i features, $\Delta(a, C)$ is a sum of $O(2^i)$ terms. This is the key to handle the exponential number of interaction terms and a key difference to our previous approaches.

Another benefit of the previous definition is that it is independent of the relationships in a feature model. We can use a satisfiability solver to compute the corresponding configurations and determine all according deltas. Hence, we increase generality. As we demonstrate in our evaluation (Section 8.4), knowing $\Delta(a, C)$ for some C is often sufficient to accurately predict non-functional properties of programs that include a. We do not need to assign values to each of $\Delta(a, C)$'s terms;

we measure only two variants of 8.1 instead of 2^i terms. Herein lies the key to the efficiency and practicality of our approach.

8.2. Determining Deltas

In the following, we describe our approach to compute each feature's delta. First, we present the general approach and continue with a step-by-step description using a concrete example.

To determine the delta of feature a, we need two measurements: $\Pi(a \times C)$ and $\Pi(C)$. By adding a to C, we add the behavior and influence of this feature to the variant. Importantly, this delta is only valid (or accurate) for the given base configuration. If C changes to C', then the influence of a on C' may change, too, resulting in a different delta value. Reasons for this change are feature interactions, which we discuss later in this chapter.

Determining the Base Configuration. We purposefully made a delta dependent on the base configuration. This way, we can determine how many terms are comprised to a single delta. That is, by increasing the number of features in the base configuration, we also increase the number of interaction terms that a delta comprises (see $\Pi(a\#C)$ in Equation 8.2). To determine the influence of a feature, we aim at excluding as many interaction terms as possible from C. The intention is that we want to measure only the influence of feature a and not how feature a works together with other features. Hence, by minimizing C, we minimize the number of interaction terms $a\#C$ that affect the computation of $\Delta(a, C)$. The general concept is as follows: For each feature a, we find a *minimal configuration* $min(a)$ that has a minimal number of features and does not contain a. The goal is to introduce feature a in the second configuration and to minimize the number of selected features. We define a minimal configuration as follows:

***Definition* 4.** *Minimal Configuration*: Let $min(a)$ and $a \times min(a)$ be two valid configurations, such that (i) $min(a)$ does not contain a and (ii) $min(a)$ is a minimal set of features that could be composed with a. We call $min(a)$ a *minimal configuration.*

With constraints between features, in principle, there can be multiple minimal configurations (for example, in the presence of mutually exclusive features). In this case, we use the minimal configuration that includes features of the initial feature set (cf. Section 6.1.1). Furthermore, we admit the empty or null program as a minimal configuration when determining non-functional properties of a root feature. We determine each feature's delta as:

$$\Delta(a, min(a)) \quad = \quad \Pi(a \times min(a)) \ - \ \Pi(min(a))$$

Example: A Customizable DBMS. Consider the feature model in Figure 8.2, which has five features. The minimal valid configurations for all features are:

Feature	$min()$
b	$\emptyset$
i	b
t	b
e	b
d	$b \times e$

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

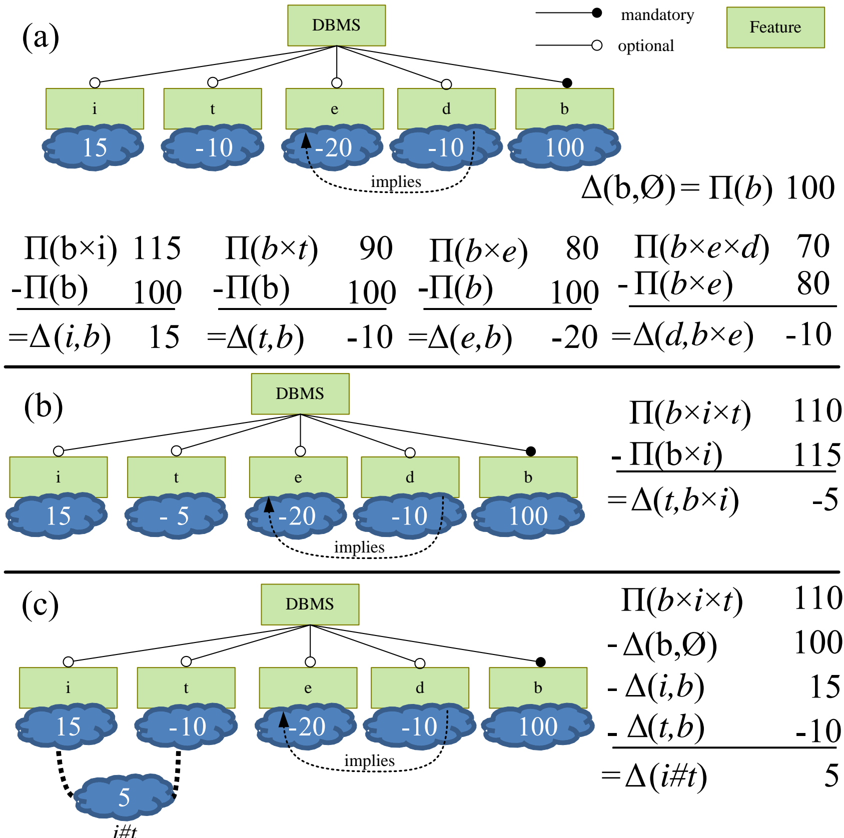
$$\begin{aligned}
\Delta_{Perf}(b, min(b)) &= \Pi_{Perf}(b) - 0 = 100 \\
\Delta_{Perf}(i, min(i)) &= \Pi_{Perf}(b \times i) - \Pi_{Perf}(b) = 15 \\
\Delta_{Perf}(t, min(t)) &= \Pi_{Perf}(b \times t) - \Pi_{Perf}(b) = -10 \\
\Delta_{Perf}(e, min(e)) &= \Pi_{Perf}(b \times e) - \Pi_{Perf}(b) = -20 \\
\Delta_{Perf}(d, min(d)) &= \Pi_{Perf}(b \times e \times d) - \Pi_{Perf}(b \times e) = -10
\end{aligned}$$

To predict performance of a configuration, we simply add the deltas of all relevant features. For example, for configuration $b \times t \times i$, we predict $\Delta_{Perf}(b, min(b)) + \Delta_{Perf}(t, min(t)) + \Delta_{Perf}(i, min(i)) = 100 - 10 + 15 = 105$. This is in line with our prediction model, because if we expand the deltas, we yield the following equation:

$$\begin{aligned}
\Pi_{Perf}(b \times t \times i) &= \Delta_{Perf}(b, min(b)) + \Delta_{Perf}(t, min(t)) + \Delta_{Perf}(i, min(i)) \\
&= \Pi_{Perf}(b) + \Pi_{Perf}(b\#min(b)) + \Pi_{Perf}(t) + \Pi_{Perf}(t\#min(t)) \\
&\quad \Pi_{Perf}(i) + \Pi_{Perf}(i\#min(i)) \\
&= \Pi_{Perf}(b) + \Pi_{Perf}(t) + \Pi_{Perf}(i) + \Pi_{Perf}(t\#b) + \Pi_{Perf}(i\#b)
\end{aligned}$$

Without considering feature interactions explicitly, the above equation is correct and even accounts for two feature-interaction terms implicitly.

Feature Interactions and Deltas. Unfortunately, this prediction scheme can become inaccurate. When measuring a feature's delta, we might obtain very different results when using different base configurations. Consider Figure 8.2b, which computes the delta for feature t with a different base configuration. Our first value, computed above, was $\Delta_{Perf}(t, min(t)) = -10$, whereas the newly computed value is $\Delta_{Perf}(t, \{b \times i\}) = -5$. Consequently, predictions for the same configuration $b \times t \times i$ will differ when using $\Delta_{Perf}(t, min(t))$ (105) or $\Delta_{Perf}(t, \{b \times i\})$ (110). The difference is due to feature interactions. We clarify the difference between both deltas by



i = Index, t = Transactions, e = Encryption, d = Decryption, b = Base;
Measured values are transactions per second.

Figure 8.2.: Measuring deltas for features and interactions. We omitted the subscript "$_{Perf}$", because of space constraints.

mapping all *known terms* (i.e., quantified by the deltas) to our prediction model. All remaining terms are set to zero and, thus, omitted in the equations below.

First, lets have a look at the prediction model for the configuration $b \times t \times i$:

$$\begin{aligned}\Pi_{Perf}(b \times t \times i) \quad = \quad & \Pi_{Perf}(b) + \Pi_{Perf}(t) + \Pi_{Perf}(i) + \Pi_{Perf}(b\#t) + \Pi_{Perf}(b\#i) + \\ & \Pi_{Perf}(t\#i) + \Pi_{Perf}(b\#t\#i)\end{aligned}$$

We consider all terms that may affect the non-functional properties of the variant with features a, b, and c. However, when using the deltas of Figure 8.2a, we have determined only a subset of all terms. There are terms for which we have no knowledge about how they influence non-functional properties. In particular, we have no

knowledge about the terms $\Pi_{Perf}(t\#i)$ and $\Pi_{Perf}(b\#t\#i)$ and, therefore, set them to zero. Hence, we build the following prediction model:

$$\begin{aligned}\Pi_{Perf}(b \times t \times i) &= \Delta_{Perf}(b, min(b)) + \Delta_{Perf}(t, min(t)) + \Delta_{Perf}(i, min(i)) \\ &= \Pi_{Perf}(b) + \Pi_{Perf}(t) + \Pi_{Perf}(i) + \Pi_{Perf}(t\#b) + \Pi_{Perf}(i\#b)\end{aligned}$$

Considering the delta of feature t in Figure 8.2b with a different base configuration, we obtain a different prediction model:

$$\begin{aligned}\Pi_{Perf}(b \times t \times i) &= \Delta_{Perf}(b, min(b)) + \Delta_{Perf}(t, \{b \times i\}) + \Delta_{Perf}(i, min(i)) \\ &= \Pi_{Perf}(b) + \Pi_{Perf}(t) + \Pi_{Perf}(i) + \Pi_{Perf}(i\#b) + \\ &\quad \Pi_{Perf}(t\#b) + \Pi_{Perf}(t\#i) + \Pi_{Perf}(t\#b\#i)\end{aligned}$$

We highlight the differences between both deltas for feature t with the color red in the above equation. We can see that the delta in the second variant expands to two additional feature-interaction terms: $\Pi_{Perf}(t\#i)$ and $\Pi_{Perf}(t\#b\#i)$. If only one of these terms is not zero, we measure a different delta, which we actually do in our example. Hence, by changing the base configuration of a delta, we can influence the number of determined interaction terms. Detecting and quantifying the influence of interactions allows us to overcome the differences among different deltas leading to consistent predictions. The question is: Which features interact that cause this discrepancy?

If we know that two features interact, we can improve our prediction by determining the influence of their interaction on a non-functional property. Our approach is to compare our prediction against the measurement of a variant. In our prediction, we do not know the influence of a feature interaction, whereas the measured value contains the interaction's influence. If we want to determine the influence of interaction $a\#b$ on a non-functional property, we build the following equation:

$$\begin{aligned}\Pi(a\#b) &= \Pi_{measured}(a \times b) - \Pi_{predicted}(a \times b) \\ &= \Pi(a) + \Pi(b) + \Pi(a\#b) - \Pi(a) - \Pi(b) - 0 \\ &= \Pi(a\#b)\end{aligned}$$

In our prediction, we do not know the value of the term $\Pi(a\#b)$ and must set it to zero. Since we already determined the influences of features a and b in isolation (i.e., $\Pi(a)$ and $\Pi(b)$), the only term that differs is $\Pi(a\#b)$.

Similar to features, we can express interactions also in terms of deltas, because also interactions can be determined with different base configurations. This is important, because, in practice, we cannot generate for each interaction term an exactly corresponding variant. Consider the interaction $i\#t$ of Figure 8.2c. To determine its influence, we cannot generate a variant that includes only features i and t. Instead, we always have to include also feature b. However, this means that we do not de-

termine $\Pi_{Perf}(i\#t)$ solely, but also $\Pi_{Perf}(b\#i\#t)$. We clarify this in the following equation:[1]

$$\begin{aligned}\Pi(b \times i \times t) - \Delta(b, \emptyset) - \Delta(i, b) - \Delta(t, b) &= \Pi(b) + \Pi(i) + \Pi(t) + \Pi(b\#i) + \\ &\quad \Pi(b\#t) + \Pi(t\#i) + \Pi(b\#t\#i) - \\ &\quad \Pi(b) - \Pi(i) - \Pi(t) - \Pi(b\#i) - \Pi(b\#t) \\ &= \Pi(t\#i) + \Pi(b\#t\#i) \\ &= \Delta(t\#i, min(t\#i))\end{aligned}$$

Rather than determine the interaction term $\Pi(t\#i)$, we determine the delta between predicted and measured variant, in which the interaction term is present. Hence, we write $\Delta(t\#i, min(t\#i))$, in which $min(t\#i)$ represents the base configuration that has the minimal number of features and includes the interaction term $t\#i$. Again, by changing the base configuration, we influence the number of interaction terms that a delta comprises.

In Figure 8.2c, we illustrate such a measurement and prediction for interaction $i\#t$. Knowing the interaction's delta improves our predictions: in our example, it patches the value of $\Delta(t, min(t))$. For higher-order interactions, we proceed in a similar way. The challenge is how to find interactions that actually contribute to a non-functional property out of an exponential number of potential interactions. To solve this problem, we use the insights we gained in the previous chapter when analyzing the evaluation of interaction-wise and pair-wise measurement. Based on these insights, we define how to detect relevant feature interactions for black-box programs.

8.3. Feature-Interaction Detection

Our goal is to identify feature interactions automatically using a small number of measurements. To this end, we specified seven requirements that an accurate prediction approach has to satisfy (see Figure 8.3). Our approach consists of two steps: (1) identifying features that participate in relevant interactions (called *interacting features*) and (2) finding minimal combinations of interacting features that actually cause a feature interaction. We use the setting from Figure 8.2 as our running example.

8.3.1. Detecting Interacting Features

Our first step is to identify features that interact. One of the insights of our previous evaluation is that *only few features interact.* By finding these features, we reduce our search space, which separates this approach from pair-wise measurement of

[1] We omit the subscript $_{Perf}$ for space reasons.

- **(R1)** Accurate predictions of non-functional properties.
- **(R2)** Black-box approach.
- **(R3)** Determining the influence of individual features on non-functional properties.
- **(R4)** Efficient detection of feature interactions.
- **(R5)** Identifying interacting and non-interacting features.
- **(R6)** Identifying hot-spot features.
- **(R7)** Infer higher-order from detected first-order feature interactions.

Figure 8.3.: Recap: Specified requirements for a prediction approach that satisfy the thesis goals.

the previous chapter. For example, suppose a program has 16 features, in which 4 features interact, the rest do not. We have to look only at $2^4 = 16$ instead of $2^{16} = 65536$ configurations to detect interactions (or $16^2 = 256$ configurations for pair-wise measurement). In the evaluation of pair-wise measurement in the previous chapter, we identified for SQLite that from 88 features, only 54 features interact (cf. Section 7.3). Hence, we satisfy our requirement R5.

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

$$a \textit{ interacts} \Leftrightarrow \; \exists\, C, D \mid C \;\neq\; D \;\wedge\; \Delta(a, C) \;\neq\; \Delta(a, D)$$

To avoid measuring $\Delta(a, C)$ for a potentially exponential number of configurations of C, we use a heuristic. We determine the deltas of a that are most likely to differ, because it is affected by the largest number of feature interactions: We compare $\Delta(a, min(a))$, the delta for the minimal configuration, with $\Delta(a, max(a))$, a delta for a configuration with the most features selected.[2] That is, we specify a maximal configuration that maximizes the number of features:

***Definition* 5.** *Maximal Configuration*: Let $max(a)$ and $a \times max(a)$ be two valid configurations, such that (i) $max(a)$ does not contain a and (ii) $max(a)$ is a maximal set of features that can be composed with a.[3] We call $max(a)$ a *maximal configuration.*

[2]We allow the empty set as a valid configuration. This is necessary to create a maximal configuration for mandatory features. Furthermore, some constraints result in the presence of multiple maximal configurations. In this case, we have to check against all of them.

[3]This may be not unique. That is, there might be several maximal set of features. In this situation, we have to measure multiple maximum configurations.

$\Delta(a, max(a))$ is, therefore, defined as:

$$\Delta(a, max(a)) \;=\; \Pi(a \times max(a)) \;-\; \Pi(max(a))$$

The rationale of determining $max(a)$ is that it maximizes the number of features that could interact with a. Consequently, if $\Delta(a, min(a))$ and $\Delta(a, max(a))$ are similar, then a does not interact with the features that are present in $max(a)$, but not in $min(a)$. Otherwise, a interacts with those features (we do not know yet with which features and to what extent). Thus, with at most four measurements per feature (two for $\Delta(a, min(a))$ using $\Pi(a \times min(a))$ and $\Pi(min(a))$, and two for $\Delta(a, max(a))$ using $\Pi(a \times max(a))$ and $\Pi(max(a))$), we discover interacting features.[4]

In our running example, we determine the following maximal configurations and assume the following corresponding measurements:[5]

Feature	$max()$	$\Pi_{Perf}(max())$
i	$b \times t \times e \times d$	60
t	$b \times i \times e \times d$	85
e	$b \times i \times t$	110
d	$b \times i \times t \times e$	90

Note $max(e)$ does not include d, as d requires e for a valid configuration (cf. Figure 8.2). With these additional measurements, we compute the additional deltas as follows with six measurements:

$$\begin{aligned}
\Delta_{Perf}(i, max(i)) &= \Pi_{Perf}(i \times max(i)) - \Pi_{Perf}(max(i)) &= 20\\
\Delta_{Perf}(t, max(t)) &= \Pi_{Perf}(t \times max(t)) - \Pi_{Perf}(max(t)) &= -5\\
\Delta_{Perf}(e, max(e)) &= \Pi_{Perf}(e \times max(e)) - \Pi_{Perf}(max(e)) &= -20\\
\Delta_{Perf}(d, max(d)) &= \Pi_{Perf}(d \times max(d)) - \Pi_{Perf}(max(d)) &= -10
\end{aligned}$$

We conclude that features i and t are interacting:

$$\begin{aligned}
\Delta_{Perf}(i, min(i)) &\neq \Delta_{Perf}(i, max(i)) & \textit{since} \quad 15 &\neq 20\\
\Delta_{Perf}(t, min(t)) &\neq \Delta_{Perf}(t, max(t)) & \textit{since} \quad -10 &\neq -5\\
\Delta_{Perf}(e, min(e)) &= \Delta_{Perf}(e, max(e)) & \textit{since} \quad -20 &= -20\\
\Delta_{Perf}(d, min(d)) &= \Delta_{Perf}(d, max(d)) & \textit{since} \quad -10 &= -10
\end{aligned}$$

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

[5] Surprisingly, $max(b)$ is an empty configuration, because feature b is mandatory; the only valid configuration without feature b is the empty set.

We know that feature i interacts with a feature in the set $max(i)\backslash min(i)$.[6] From these candidate features, we can exclude features b, e, and d, because their deltas do not change. Feature t remains the only candidate for interaction. The same conclusion would have been reached had we analyzed feature t (concluding feature i is the only possible interaction candidate). In this way, we found the feature combination that causes an interaction. Note that if we find more than two interacting features, we have no information which feature combination causes an interaction. Finding these feature combinations is the goal of the next step. Again, we use our insights of the previous chapter to accomplish the goal.

8.3.2. Identifying Feature Combinations Causing Interactions

After detecting all interacting features, we have to find the specific, valid combinations that actually have an influence on a non-functional property. Suppose we know that features a, b, and c are interacting. We have to identify which of the following interactions have an influence on a non-functional property: $a\#b$, $a\#c$, $b\#c$, or $a\#b\#c$. Again, we do not want to measure all combinations (whose number is exponential in the number of interacting features).

At this point, our other findings of the previous experiment become important. We identified that (a) most of the interactions are first-order interactions, (b) there are patterns from which we can infer higher-order feature interactions based on detected first-order interactions, and (c) there can be hot-spot features (cf. Section 7.4). To this end, we developed three heuristics. Each makes an assumption under which it can detect interactions (thus, improving prediction accuracy) with few additional measurements. That this, the more of our heuristics we use, the more measurements we need, but the more interactions we detect. This way, we attempt to detect all relevant interactions without measuring *all* configurations. Again, our heuristics are based on the experience we gained during the analysis of feature interactions in the last chapter. We further backed up our assumptions with findings of related research regarding the occurrence of functional feature interactions [Liebig et al., 2010] and coupling of feature code [Apel and Beyer, 2011].

Note that our heuristics are based on each other. That is, the second heuristic requires the detected feature interactions as a result of applying the first heuristic. Hence, we consecutively apply our heuristics to a customizable program. We explore in our evaluation whether our heuristics actually reduce measurement effort and improve accuracy of our predictions.

Auxiliary – Implication Graph. In all three heuristics, we reason about feature chains in an implication graph. An *implication graph* is a graph in which nodes represent features and directed edges denote implications between features. Using

[6] $max(i)\backslash min(i) = A - B$ in set algebra.

implications, we conclude that $\Delta(a, min(a))$ always includes the influence of all interactions with features implied by a (i.e., all features in a's implication chain). For example, if feature a always requires the presence of feature b, then we have implicitly quantified the influence of interaction $a\#b$ when computing $\Delta(a, min(a))$. This mechanism reduces computation effort in all heuristics, especially for hierarchically-deep feature models and for feature models with many constraints. We describe the concept of the implication graph in Section 8.3.3 in detail.

Heuristic 1 – Pair-Wise Interactions (PW). Based on Insight 5 in Section 7.4, we assume that pair-wise (or first-order) interactions are the most common form of non-functional feature interactions.

In addition to our analysis, we justify this assumption as follows: Related research often uses a similar approach. The software-test community often uses pair-wise testing to verify the correctness of programs [Cohen et al., 1996, Tai and Lei, 2002]. Pair-wise testing was also applied successfully to test feature interactions in the communication domain [Williams, 2000] and to find bugs in product-line configurations [Oster et al., 2010]. Furthermore, analysis of variability in 40 large-scale programs showed that functional interactions (i.e., nested `#ifdefs`) occur mostly between two features [Liebig et al., 2010]; although functional interactions do not necessarily cause non-functional feature interactions, we assume that this distribution also holds for most non-functional properties, because the additional code may have some affect.

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

Heuristic 2 – Composition of Higher-Order Interactions (HO). Based on Insight 6, we assume that second-order feature interactions (i.e., interactions among three features) can be predicted by analyzing already detected pair-wise interactions.

In our analysis, we found the following pattern: If two of these three interactions $\{a\#b, b\#c, a\#c\}$ are non-zero, then also the interaction $a\#b\#c$ is non-zero, i.e., has an influence on a non-functional property. The rationale is, if three features interact pair wise in any combination, they likely also participate in a triple-wise (second-order) interaction. For example, if both $a\#b$ and $b\#c$ allocate 1 GB RAM, then it is likely that there is an interaction $a\#b\#c$ that results in a lower performance (because 2 GB RAM was allocated). Although we could analyze this phenomenon only for the footprint experiment, a different footprint may also indicate a possible impact on other non-functional properties, such as performance and energy consumption, because either functionality is added (increased footprint) or is removed (decreased footprint). This added or removed functionality can cause deviations in observed non-functional properties.

We do not consider other higher-order interactions for this heuristic to save a huge number of measurements, although this is possible, because we found that this pattern propagates to higher interactions. Thus, we might miss some interactions in attempt to balance measurement effort and accuracy.

Heuristic 3 – Hot-Spot Features (HS). Finally, based on Insight 4, we assume the existence of *hot-spot* features. We identified that there are usually few features that interact with many features and there are many features that interact only with few features. High coupling between features or many dependencies can impact the non-functional properties of the whole system, because both features strongly interact with each other at the implementation level.

In the previous chapter, we found that footprint feature interactions do not normally distribute over all features in a customizable program. Instead, there are hot-spot features that interact with (nearly) all other features. For example, the Linux kernel feature *CC_Optimize_For_Size* interacts with all other features. That is, it changes the size of all other Linux modules. Similarly, SQLite's feature *SQL_OMIT_COMPLETE* interacts with many other features (cf. Figure 7.13).

Recent studies about module coupling and feature cohesion back up our assumption [Apel and Beyer, 2011, Taube-Schock et al., 2011]. Features and modules are often coupled in a scale-free graph. In a scale-free graph, links l (or dependencies) of modules follow a power-law distribution. This means that the probability $P(l)$ of linking to a given node is proportional to the number of existing links that the node has and the scaling exponent γ [Barabási et al., 1999]: $P(l) \sim l^{-y}$. High coupling between features or many dependencies can impact the performance of the whole system, because both features strongly interact with each other at the implementation level. Since we confirmed this assumption not only on functional feature interactions, but also on non-functional feature interactions of the type footprint, we assume that this holds for many non-functional properties, such as performance.

Using this heuristic, we perform additional measurements to locate interactions of hot-spot features with other interacting features. Specifically, we attempt to locate second and third-order interactions for hot-spot features, because they seem to represent a non-functional property-critical functionality in a program. We do not identify interactions with an order higher than three, because this increases measurement effort substantially.

8.3.3. Realization

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

As an underlying data structure, we use an implication graph, as described earlier. We can easily generate this graph from a feature model using a satisfiability solver [She et al., 2011]. To locate pair-wise interactions (PW heuristic), we consider only pair-wise interactions between interacting features of different implication chains. We do not need to determine interactions of features belonging to the same implication chain, because the interaction is already included in the delta of the feature that is closer to the end of the chain. Consider the implication graph in Figure 8.4, when measuring the delta of feature F_4 with $min(F_4)$, then $min(F_4)$ already contains features F_1, F_2, and F_3, because we cannot derive a *valid* configuration without these features to measure $\Pi(F_4) \times min(F_4)$. Hence, $\Delta(F_4, min(F_4))$ has the following terms: $\Pi(F_4) + \Pi(F_1\#F_4) + \Pi(F_2\#F_4) + \Pi(F_3\#F_4)$. We comprised the last three terms as $F_4\#min(F_4)$ in our delta definition (cf. Equation 8.2). Hence, this concept is a further optimization to reduce measurements compared to pair-wise measurement.

The order of the measurements is crucial. For example, in Figure 8.4, if we measure the pair $F_4 \times F_7$ first, we would measure the influence not only of interaction $F_4\#F_7$, but also $F_1\#F_6$. That is, $\Delta(F_4\#F_5, min(F_4\#F_5))$ sums the influence of both terms. If we would later determine $\Delta(F_1\#F_6, min(F_1\#F_6))$, then we would consider the interaction twice in our prediction model leading to inaccurate predictions. Hence, our algorithm starts from the top of one implication chain and determines the influence of interacting features with the interacting features of another chain, also starting from the top. Afterwards, we continue with the next chain. In our example, the order we use to detect pair-wise interactions is $F_1\#F_6$, $F_1\#F_7$, $F_4\#F_6$, $F_4\#F_7$, $F_6\#F_{11}$, $F_7\#F_{11}$,$F_1\#F_{11}$, and $F_4\#F_{11}$.

To identify whether two features a and b interact, we compare the measured non-functional property $\Pi(a \times b)$ with the *prediction* of the same configuration that *includes* all known feature interactions up to this time. If the result of $\Delta(a\#b, C)$ exceeds a threshold (e.g., we use the standard deviation of measurement bias as a threshold), we record it.

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

Finally, we search for further second-order and third-order interactions involving hot-spot features (HS heuristic). We count the number of interactions per feature identified so far. Next, we compute the arithmetic mean of interactions per feature. We classify all features that interact above the arithmetic mean as hot-spot features (other thresholds are possible, too). With hot-spot features, we search (with the usual mechanism: additional measurements, comparing deltas) for interactions involving (1) a hot-spot feature, (2) a feature that already interacts with this hot-spot feature,

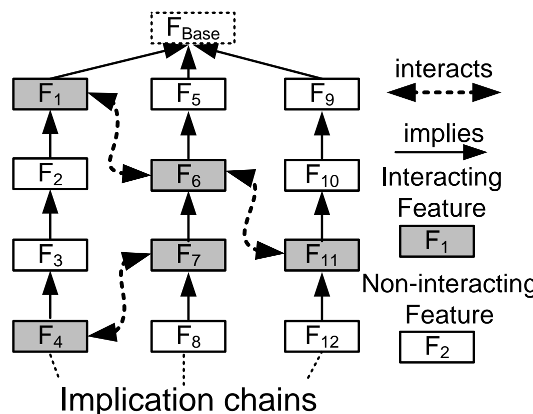


Figure 8.4.: Implication chains with interacting features.

and (3) an interacting feature that does not interact in a pair-wise manner with the hot-spot feature.

Optimization for Deep Implication Chains. We said above that the ordering matters at which we detect feature interactions – that is, why we start from the top of an implication chain. However, for deep implication chains, it may be beneficial to detect whether there is an interaction at all among features of two chains. For example, there is no interaction between features of the left and right chains in Figure 8.4. But we perform two measurements to check whether F_1 interacts with F_{11} and F_4 interacts with F_{11}. Consider more interacting features in both chains, we would need a quadratical number of measurements to identify only that there is no interaction at all. To overcome this problem, we can check whether two chains interact depending on the length of a chain with respect to the number of interacting features. If the prediction of a variant containing features that are located at the end of two implication chains is correct, we assume that there is no interaction among all features of both chains.

To determine the existence of at least one interaction in two chains is easy. Rather than starting from the top of a chain, we measure the combination of the interacting features that are at the bottom of two chains. That is, we measure the combination $F_4 \times F_7$, $F_4 \times F_{11}$, and $F_7 \times F_{11}$ to check for interactions among these chains.

Even if we detect an interaction between two chains, it can be beneficial to reduce the combinations of interacting features that cause this interaction. We do so by selecting interacting features at the middle of both chains and check whether there is an interaction. If not, we have to consider only feature interactions that are below of the chosen features. These additional checks have a complexity of $O(logn)$, in which n is the number of interacting features of both chains. Furthermore, these additional measurements are not in vain. If we detect interactions between two

chains, we reach eventually the point at which we used the feature combination for checking whether two chains interact. We predict non-functional properties for the corresponding variant with all identified interaction terms so far and compare it with the already executed measurement.

8.4. Evaluation

The goal of this evaluation is to verify whether our automated feature-interaction detection is feasible with respect to measurement effort and prediction accuracy. We use performance for this evaluation for the several reasons. First, for performance, we observed the highest error rates with feature-wise measurement. Hence, we consider performance as a challenging property for which many (higher-order) feature interactions exist. Second, performance is important in practice. Showing that our approach is feasible for performance is an important contribution not only for the research community, but also for practice. Third, our heuristics base on our findings of the properties footprint and main-memory consumption. If we would use these properties again for verification, we might observe only good results, because we tailored our approach to these two properties. Hence, we verify these heuristic with a new property. Finally, the measurement effort for this experiment is very high. Evaluating performance required over 60 days of measurement for all our customizable programs, including the measurement of the evaluation set.

We use the same experimental setup as for feature-wise measurement (see Figure 8.5). Since we observed measurement bias for performance, we measure each program several times. From these measurements, we compute the average performance (i.e., arithmetic mean) and the standard deviation. We use the average performance to compute the delta of a feature. We use the standard deviation to set the threshold at which we identify a feature interaction, because we consider every unexpected performance behavior above the measurement error as an interaction.

To rate accuracy and measurement effort, we state the following research questions:

- **Q1:** Do the heuristics reduce the average error rate of predictions, especially compared to feature-wise measurement?
- **Q2:** What is the additional measurement effort for the different heuristics?

Again, we use the error rate as a metric to answer the first research question. We compute an error rate of our prediction as the relative difference between predicted and actual performance: $\frac{|actual-predicted|}{actual} * 100$ and accuracy as 1-error rate in percent.

Results. In Table 8.4, we show the results of our six case studies: For each approach, we depict the required number of measurements, the time needed for these

Experimental setup for feature-wise measurement (see Section 6.2):

- Error rate: $\frac{|actual - predicted|}{actual} * 100$
- Two-step design: (i) Building prediction model with measurements and (ii) evaluating prediction model against whole population or randomly selected variants.
- Performance material: *Berkeley DB CE, Berkeley DB JE, Apache, SQLite, LLVM*, and *x264*.

Figure 8.5.: Recap: Experimental setup for performance.

measurements, and the number of identified interactions. Furthermore, we show the distribution of the error rate of our predictions with box plots. Finally, we show for each approach the mean error rate of all predictions, including the standard deviation. Again, when adding a new heuristic, we keep the previous heuristic working, because a new heuristic requires the detected feature interactions of the previous heuristic.

Program	Appr.	Effort: Measurements	Effort: Time (in h)	Interactions	Error Rate (in %): Distribution	Error Rate (in %): Mean±Std	Error Rate (in %): Median
Berkeley CE	FW	15 (0.6 %)	3	0		44.1±42.3	49.97
	PW	139 (5.4 %)	23	14		3.9±5.3	2.23
	HO	160 (6.3 %)	27	22		2.8±3.7	0.9
	HS	164 (6.4 %)	27	22		2.8±3.7	0.9
	BF	2 560 (100 %)	426	-		—	
Berkeley JE	FW	10 (3 %)	8.4	0		17.7±19.6	11.27
	PW	48 (12 %)	40	24		8.5±9.6	5.47
	HO	116 (29 %)	97	51		3.8±5.7	1.41
	HS	162 (40.5 %)	137	69		1.7±3.5	0.57
	BF	400 (100 %)	335	-		—	
Apache	FW	9 (4.7 %)	10	0		14.9±24.8	5.88
	PW	29 (15.1 %)	32	18		7.7±11.2	5.0
	HO	80 (41.7 %)	89	44		11.6±22.7	3.39
	HS	143 (74.5 %)	159	73		5.3±10.8	1.21
	BF	192 (100 %)	213	-		—	
SQLite	FW	26 (0 %)	2.1	0		7.8±9.2	6.9
	PW	566 (0 %)	47	2		9.3±12.5	6.04
	HO	567 (0 %)	47	3		7.1±9.1	5.79
	HS	569 (0 %)	47.4	3		7±9	5.75
	BF	3 932 160 (100 %)	ca. 327 680			—	
LLVM	FW	11 (1.1 %)	2	0		7.8±9	7.49
	PW	62 (6.1 %)	12	27		7.4±10.2	5.53
	HO	62 (6.1 %)	12	27		7.4±10.2	5.53
	HS	88 (8.6 %)	17	38		5.7±7	4.43
	BF	1 024 (100 %)	202	-		—	
x264	FW	12 (1 %)	2	0		29.6±22	29.23
	PW	81 (7 %)	16	13		17.9±27.2	2.32
	HO	89 (7.7 %)	17	17		5.1±15.1	1.32
	HS	89 (7.7 %)	17	17		5.1±15.1	1.32
	BF	1 152 (100 %)	224	-		—	

Table 8.1.: Evaluation results for customizable programs; approaches (Appr.): feature-wise (FW), pair-wise heuristic (PW), higher-order heuristic (HO), hot-spot heuristic (HS), brute force (BF). Mean: mean error rate of predictions, Std: standard deviation of predictions.

Feature-wise measurement (FW) does not use a heuristic and does not account for feature interactions (i.e., we note the results of our first experiment; cf. Section 6.2.4). We achieve good predictions for programs in which interactions have no substantial influence on performance. For example, our predictions have an average error rate of less than 8 % for all LLVM configurations. In contrast, we usually have a high error rate (e.g., over 44 % for BerkeleyDB C version) when no interactions are considered. The average error rate of feature-wise performance prediction is 20.3 %.

Using the pair-wise heuristic (PW) usually improves predictions significantly to a mean error rate of 9 % (i.e., 91 % accuracy, on average), because the majority of interactions are pair-wise, as also identified by our Insight 5 (cf. Section 7.4). The benefit of implication chains compared to the common pair-wise measurement is that it reduces the number of measurements. For example, we require 81 measurements to detect first-order interactions for x264 (see Table 8.4), which is 82 less than 163, which would be needed to measure all pairs of features.

With the higher-order (HO) heuristic, we achieve an average error rate of 6.3 % for all case studies, which means that we predict, on average, 93.7 % of all programs correctly. Interestingly, for LLVM, we could not find a feature combination that satisfies our preconditions to search for higher-order interactions. It is important to note that this heuristic usually doubles the number of measurements. For Apache, the error rate increases, because measurement bias over the determined threshold leads to a false detection of interactions. We detected these false positives when we search for third-order feature interactions, as we do with the hot-spot heuristic.

Finally, the hot-spot heuristic (HS) (including the other two heuristics) decreases error rate again to 4.6 % (an average accuracy of 95.4 %). Considering that the measurement bias for a single measurement of the case studies Apache, LLVM, and x264 is 5 %, for SQLite it is 7 %, and for Berkeley DB C and Java version it is 2 %, our predictions are as accurate as the measurement error of a single measurement (Q1).

To summarize our observations, we show the error-rate distributions using violin plots in Figure 8.6. We see that with each additional heuristic, we reduce the error rates of our predictions considerably. Interestingly, we observe for x264 some outliers. Although the overwhelming majority of variants is predicted within an error rate of about 5 %, there are still higher-order feature interactions that we do not find with our approach. A possible reason is that we did not follow the identified pattern of propagation of higher-order feature interactions completely, but stopped at the order of two (to limit measurement effort). Again, these are only heuristics and not an exhaustive measurement. We would need to measure a large number of additional variants to reduce the error rate by some few percents. The question is, it is worth measuring these variants and is it even possible for highly variable programs?

We believe that the results strongly suggest that the heuristics substantially improve prediction accuracy with only few additional measurements. Furthermore, our

average error rate of all variants stays within the measurement error, which means that our predictions are nearly perfect. In addition, when investigating the median of predictions (which is usually done in statistics), we observe that the average medians over all customizable programs drop from 18.47 % tp 2.36 % for the hot-spot heuristic. That means that half of our predictions of all programs achieve an accuracy of nearly 98 % by measuring only a small fraction of the programs and without considering domain knowledge.

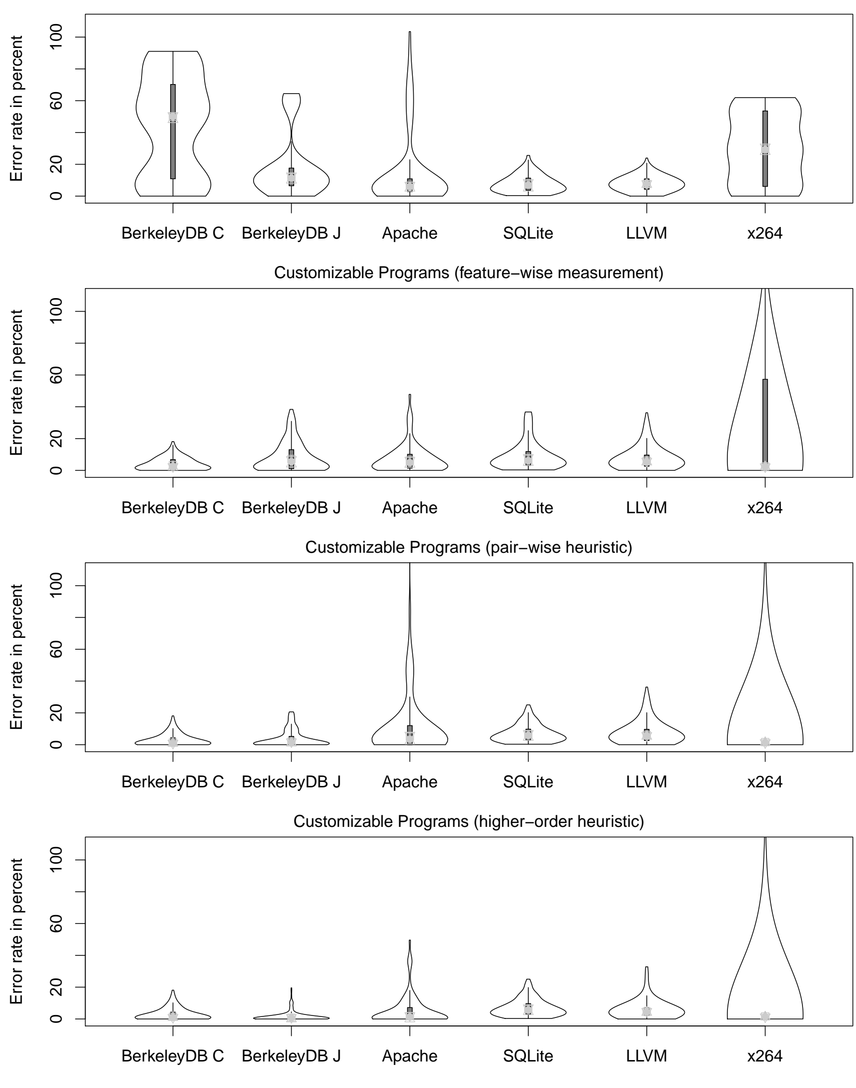


Figure 8.6.: Error-rate distributions for feature-wise measurement and our three heuristics: pair-wise, higher-order, and hot-spot.

The key findings are as follows:

- Predictions are nearly perfect: a error rate of 4.6 % lies within the measurement inaccuracies of the used programs.
- Median error rates are: FW 18.46 %, PW heuristc: 4.32 %, HO heuristic 3.06 %, and HS heuristc 2.36 %.
- Each heuristic improved prediction accuracy further. Average error rates: PW heuristic 9 %, HO heuristic 6.3 %, and HS heuristic 4.6 %.
- Measurement effort is low for highly customizable programs: using all heuristics, 0.01 % of all variants must be measured for SQLite.

8.4.1. Discussion

In the following, we discuss the influence of our heuristics regarding measurement effort and accuracy.

Influence of Heuristics. Since all our heuristics are consecutively applied, we can visualize the trade-off between additional measurements and error rate of predictions as in Figure 8.7 (Q2). For each of the customizable programs, we show a diagram, in which we compare measurement effort in percentage with respect to number of valid configuration with error rate in percent. As expected, with an increasing number of measurements, the error rate decreases. The results show that the relative number of measurements differs when we want to reach the same accuracy for different programs, but the absolute number of measurements stays low.

Interestingly, we always reach a point for which we can provide accurate predictions with a relatively low number of measurements. For example, for Berkeley DB C version, we need to measure only 5 % of all variants to reach an accuracy of 96 % on average. Further increasing the number of measurements by applying more heuristics does not make sense, because we reached already a sufficient accuracy for most application scenarios (i.e., we are in the region of the measurement error regarding our prediction error rate). Similarly for Apache, by measuring only 29 variants, we can predict all 192 variants with an accuracy of over 92 % on average. A considerable increase of measurement improves the accuracy only slightly.

We showed that the pair-wise heuristic generally improves accuracy, but also identified that it reaches not always a sufficiently high accuracy. The higher-order heuristic achieves with only 8 additional measurements (0.7 % of all variants), compared to the pair-wise heuristic, an improvement from 18 % error rate to 6 % error rate (94 % accuracy). Further, note that we have to measure approximately 0.1 % of all variants of SQLite, even when using all our heuristics, which demonstrates the scalability of our approach. Hence, we see that our heuristics make the connection between measurement effort and prediction accuracy clear, which allows us to find for each customizable program the sweet-spot at which our approach is feasible.

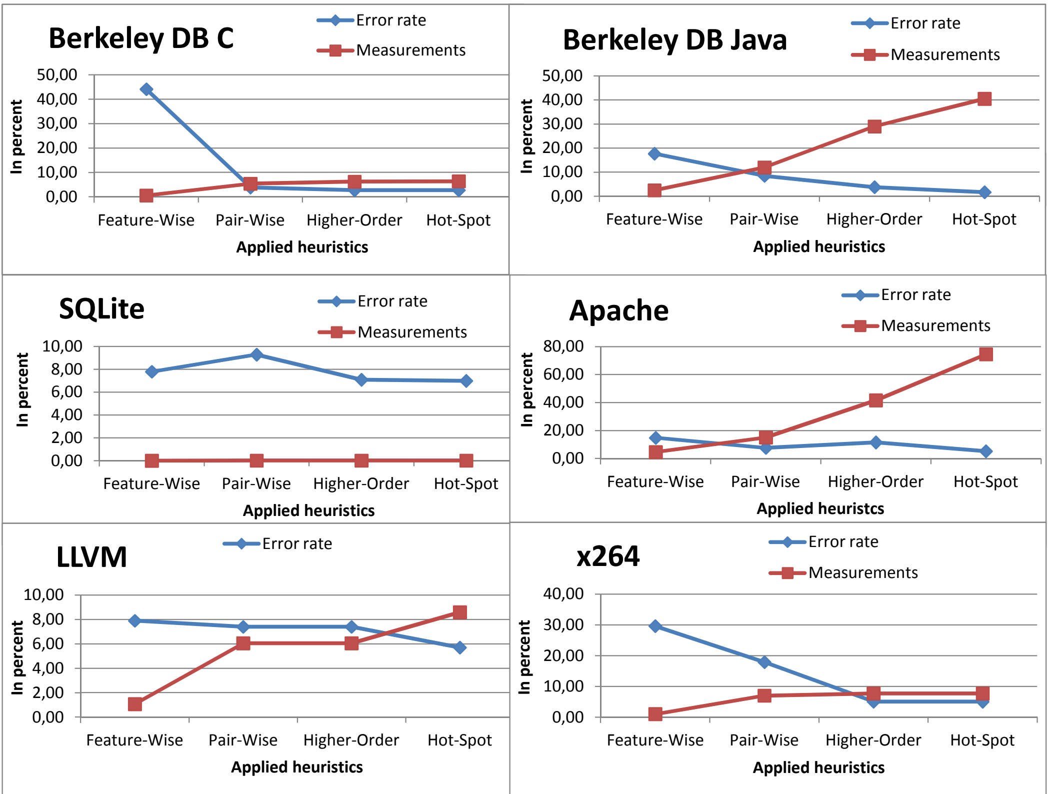


Figure 8.7.: Comparing percentage of measurements with average error rates of predictions for each heuristic and customizable program.

Accuracy. By additionally determining feature-interaction terms, we demonstrated that the approach is feasible and achieve an average accuracy of 95 %. That is, our predictions are nearly perfect, because they are in the range of the observed measurement bias for the case studies. It is important to note that we experienced large differences in accuracy, when we changed the threshold at which a feature interaction is detected. Having a too small threshold causes many false detections of interactions. The error rate increases, because we sum the influence of measurement bias instead of the influence of interactions.

We observed that we need a relatively large number of measurements when many alternative features exist compared to independent features, because alternative features limit the number of valid configurations significantly. For example, we can generate only 400 configurations in Berkeley DB Java, although it has 32 features. This number is below quadratic. Hence, already the detection of interacting features requires a relatively high number of measurements. However, having programs with a small number of valid configurations makes a brute-force approach feasible, which is not our intended scenario.

Threats to validity of feature-wise measurement (cf. Section 6.2.7):
Construct validity:
- Stated research questions before conducting the experiments.
- Using standard benchmarks for performance measurement rather than self-developed.

Internal validity:
- Using 100 sample variants for SQLite.

External validity:
- Measurement of many, very different real-world programs with varying implementation techniques, languages, etc.

Figure 8.8.: Recap: Threats to validity of feature-wise measurement.

Furthermore, we do not consider performance behavior of a program independently of the workload, which we leave for future work (see Section 9.3). We make accurate statements for any configuration given a *specific* workload. That is, we address end users who have a certain application scenario in mind, but do not know which configuration performs best. Measurements can be performed on a live system in a real environment, which produces more accurate predictions than standard benchmark results in a synthetic environment, which was one of our stated goals in Chapter 5. With a new workload, we have to repeat the measurements. We believe that many interactions still exist, although the values of the interactions will change. This, however, means that we may save measurements for new workloads, since we already know which features interact.

8.4.2. Threats to Validity

The threats to validity are the same as for feature-wise measurement (see Figure 8.8). We consider an additional external threat to validity.

External Threat to Validity. We evaluated our approach only for performance. We selected performance, because we identified the highest error rates using feature-wise measurement. Hence, we believe that performance is a challenging problem for an accurate prediction. Furthermore, we consider performance as a non-functional property that exhibits many non-functional feature interactions. Hence, an automated feature-interaction-detection approach must find feature interactions of different orders and in different distributions among all features. We showed for performance that the findings based on footprint and main-memory-consumption experiments hold for performance. Since it would be not fair to evaluate the heuristics for the

same properties that we used to define, we had to chose another property to maximize internal and external validity.

8.5. Related Work

There is a large body of research on automated detection of feature interactions (e.g., see Nhlabatsi et al. [2008] and Calder et al. [2003a] for surveys). Many approaches aim at detecting feature interactions at the specification level. For example, Calder and Miller use a pair-wise measurement approach based on linear temporal logic to detect feature interactions [Calder and Miller, 2006]. They specify the behavior of a product line in Promela (a modeling language). Using a model checker, they generate for each pair-wise combination a model checking run to verify whether the defined properties are still valid. Other approaches use state charts to model and detect feature interactions [Prehofer, 2004]. For example,Pomakis and Atlee [1996] translate feature specifications to a reachability graph. The authors use state transitions to detect whether a certain state is not exclusively reachable in isolation (i.e. a feature interaction occurs).

There are approaches that provide means to detect semantic feature interactions, i.e., feature interactions that change the functional behavior of a program. Some use model checking techniques to find semantic feature interactions [Classen et al., 2010, Lauenroth et al., 2009]. Apel's work uses model-checking techniques to verify whether semantic constraints still hold in a particular feature combination [Apel et al., 2010b, 2011]. Other approaches aim at investigating the code base to detect structural feature interactions. For example, Batory et al. [2011] and Liu et al. [2005] propose to model feature interactions explicitly using algebraic theory. In contrast to these approaches, we focus on non-functional feature interactions in a black-box fashion.

8.6. Summary

We presented an approach to automatically detect non-functional feature interactions of customizable programs. Our approach does not require any domain knowledge or source code. It considers a program as a black box, in which only the input variables are known (i.e., the configuration). Hence, we are not limited to a specific customization technique, implementation language, or domain.

The detection has two steps: (i) detecting which features interact at all and (ii) finding the combination of interacting features that actually cause a feature interaction. In the first step, we determine the influence of each feature two times. First, we measure a minimal configuration for a feature to minimize the influence of interactions on this measurement result. Second, we measure a maximal configuration

to maximize the influence of interactions on the result. We know that a feature interacts if the approximations of both measurements yield different results.

In the second step, we use three heuristics to find combinations of interacting features that influence non-functional properties. The first heuristic determines between each pairs of features whether an interaction exists (pair-wise heuristic). The higher-order heuristic bases on the identified pair-wise heuristic by measuring only those combinations of feature interactions, in which at least one feature participates in several pair-wise interactions. The hot-spot heuristic determines whether third-order interactions exist between interacting features and a hot-spot feature. A hot-spot feature has an overproportional number of interactions with other features. By counting interactions that we determined with both previous heuristics, we identify hot-spot features.

We evaluated the three heuristics with performance using six real-world customizable programs. We identified that the error rate decreases from 21.3 %, on average, over all programs using feature-wise measurement to 5 % using all three heuristics. We made the following observations:

- The order at which we determine feature-interaction terms matters. We need to determine feature interactions beginning from the root of an implication chain. Otherwise, we would account for the same feature interaction multiple times.
- Feature terms can be composed using deltas without sacrificing prediction accuracy. We showed that using deltas in combination with implication chains reduce the number of measurements significantly for programs with deep feature models.
- We confirmed the trade-off between measurement accuracy and number of measurements. With our heuristics, we required additional measurements, but could improve prediction accuracy. We limited the order of interactions that we want to determine to reduce the number of measurements.
- Non-functional feature interactions occur in some patterns. We used these patterns to define heuristics that help finding these interactions. With our evaluation, we could confirm the existence of these patterns as we find feature interactions to reduce the prediction error rate significantly.

We believe that these observations may also be important for detecting other kinds of feature interactions, such as behavior feature interactions. Hence, our research has not only an influence on non-functional properties, but also for the feature-interaction-detection community. This finalizes the measurement part of this thesis.

We presented means to determine the influence of individual features and different approach to detect and quantify the impact of feature interactions on non-functional properties. Together with Part 1 of this thesis, we enable measurement and predic-

tion of non-functional properties of customizable programs, such that we can compute (with constraint-satisfaction-problem solver) variants that are near-optimal for given non-functional requirements. With each evaluation and using the constructive research method, we extended our problem statement, such that we defined additional requirements that an accurate and feasible measurement and prediction approach must satisfy. We summarize our approach:

- Accuracy: We predict performance nearly perfect with an accuracy of over 95 %, on average, which is in the rage of measurement bias.
- Measurement effort: Our heuristics required measuring less than 1 % of variants of SQLite and requires few measurements for the other programs in absolute numbers.
- Black box: Our approach does not require any domain knowledge or source code.

Hence, we conclude that our automated feature-interaction-detection approach presented in this chapter satisfies all thesis goals and all requirements.

9. Concluding Remarks and Future Work

We summarize the results of the thesis in general and each chapter in detail. Subsequently, we state our contributions to the research community and to real-world applications. We conclude with an outline of current and future research directions.

9.1. Conclusion

We developed an approach that allows software developers to predict measurable non-functional properties, such as performance, footprint, and main-memory consumption of feature-customizable programs. We considered a broad range of customization techniques, such as automated generation using software product line engineering, command-line parameters, and configuration files.

To yield a tailor-made program of a customizable program (called a variant), users select customization options (called features) that satisfy their requirements. Although qualitative estimations about some features of a program may be possible (e.g., a hash index is often faster than a list), it is often not possible without a measurement to quantify the influence of a certain feature on non-functional properties. This quantification is, however, often required when a variant needs to fulfill guarantees (e.g., performance guarantees) or strict constraints (e.g., resource limitations of embedded systems) regarding non-functional properties. The benefit of customizable programs (i.e., providing many customization options for tailoring) becomes a drawback when users want to know which is the *best* variant for their requirements, because we observe an exponential explosion of the variant space: Each optional and independent feature doubles the number of variants, and already 265 optional features yield more variants than estimated number of atoms in the universe. Hence, a brute-force approach to measure non-functional properties is infeasible (cf. Chapter 3).

To overcome the scalability problem of measuring an exponential number of variants, we proposed to predict rather than measure a variant's non-functional properties (cf. Chapter 4). To this end, we presented a prediction model derived from a feature-composition models that allows us to determine which features and feature combinations have to be measured to predict a variant's non-functional properties. In Chapter 5, we reviewed and compared alternative measurement strategies to measure the influence of features and feature combinations on non-functional properties.

We found that our approach of *feature-wise measurement* satisfies our goals of: (i) reduced measurement effort, (ii) high prediction accuracy, (iii) large applicability (by supporting black-box programs), (iv) large generality (by supporting all measurable non-functional properties), and (v) support for measurements in realistic environments.

In Chapter 6, we described *feature-wise measurement* in detail. We generate two variants per feature (one with the feature and one without) and interpret the delta (or difference) of the measured properties of these variants as the influence of the corresponding feature. To predict a variant's non-functional properties, we aggregate (i.e., add, max, min, mean, etc.) the influences of all selected features. Furthermore, our approach considers all programs as black boxes. Hence, measurement and predictions are independent from certain programming languages, implementation techniques, customization mechanisms, application domains, and so on.

This simple approach is the key to manage a huge configuration space. Although this approach exhibits already a good prediction accuracy (e.g., an average accuracy of 94.5 % when predicting footprint for eight customizable programs), we encountered inaccuracies when feature interactions occur. A feature interaction exists if a variant with two features exhibits unexpected behavior of non-functional properties, whereas we do not detect such a behavior when only a single feature is present. Hence, to increase accuracy of predictions, we proposed two approaches to quantify the influence of feature interactions on non-functional properties: interaction-wise and pair-wise measurement (cf. Chapter 7). For interaction-wise measurement, we use domain knowledge and code analyses to manually identify which features interact. For pair-wise measurement, we assume that each pair of features interact and measure the corresponding interactions. We showed that interaction-wise measurement significantly increase prediction accuracy (i.e., 99.8 % accuracy, on average, for eigth programs), and pair-wise interactions often improve prediction accuracy, but sometimes also decrease accuracy.

We found that higher-order interactions may decrease accuracy for pair-wise measurement. Hence, we conducted a second analysis based on the same evaluation data. From this analysis, we gained several important insights: We observed that feature interactions are not evenly distributed over all features, but follow patterns. Furthermore, we identified that many features do not interact and that first-order interactions a more common than higher-order interactions. Based on these insights, we developed an approach that automates the detection of feature interactions without the need of domain knowledge and availability of source code (cf. Chapter 8).

In two steps, we identify which features interact and which combinations of these interacting features actually cause an observable interaction. To find these combinations, we use three heuristics (pair-wise, higher-order, and hot-spot) that are based on the patterns we identified in the previous experiments. We evaluate prediction accuracy for our three heuristics of six real-world programs with the non-functional

property performance. We demonstrated that the error rate of predictions decreased from 20.3 %, on average, for feature-wise measurement, to 4.6 %, on average, when using all heuristics. Since these heuristics aim at measuring only few additional variants, we found a sweet spot between measurement effort, accuracy, and generality of our approach.

We conclude that by using our prediction model in combination with the automated feature-interaction detection, we fulfill all thesis goals: We achieve nearly perfect predictions (over 95 %, on average, for performance), we need only few measurements (below 1 % of all variants for highly customizable programs), and we support black-box programs (i.e., we do not rely on domain knowledge or source-code analysis). Since our prediction model abstracts from concrete implementation and allows us to have different mappings and aggregation functions, we support all automatically measurable properties. To the best of our knowledge, this is the first time that a prediction approach achieves this high accuracy, is applicable for diverse non-functional properties, and supports even black-box programs.

In the following, we summarize our contributions.

9.2. Contributions

We showed that it is indeed possible to predict measurable non-functional properties of highly customizable programs with justifiable measurement effort and for black-box programs.

Our contributions are the following:

1. **Measurement of Non-Functional Properties.** Measuring non-functional properties is a challenging task, because different non-functional properties often require different measurement techniques, and different programs require different customization methods to generate a variant to be measured. For the first time, we showed a generally applicable measurement strategy that is independent of implementation techniques, programming languages, works for black-box programs, and achieves nearly perfect prediction results. Furthermore, it can be used in realistic environments and does not need to use the source code of a program.
2. **Automated Feature-Interaction Detection.** Feature interactions cause problems in many situations and domains, because they cause an unexpected system behavior. Many approaches were published to detect feature interactions, but they require domain knowledge or availability of code or other development artifacts. In contrast, we developed a novel automated detection approach that identifies feature interactions in black-box programs using three

heuristics. Detecting these interactions reduced the average error rate of our predictions by 15 %, which results in accurate predictions.

We believe that similar patterns exist also for other domains (not only for non-functional properties). We showed how to identify such patterns and how to develop from these insights heuristics that allow us to detect feature interactions in an automated manner. Hence, we believe that related research communities profit from our research methodology and even from our identified patterns to detect feature interactions in their respective domains.

3. **Prediction of Non-Functional Properties.** We contribute a prediction model to the research community, which is extensible and adaptable to different non-functional properties. The central elements are *terms*, which quantify the influence of features and feature combinations on non-functional properties. We outlined that, when predicting non-functional properties, we need to identify only the relevant terms in our prediction model. Furthermore, we can use different aggregation functions, such as multiplication, addition, and maximum, to address different properties (e.g., maintainability vs. performance). The model is extensible in the way that the terms abstract from the concrete realization. That is, we proposed to use two measurements to compute the corresponding delta to have a concrete value per term.

   Furthermore, we currently extend our model, such that terms do not represent concrete values, but workload-dependent cost functions to quantify the influence of features on non-functional properties. That is, we aim at considering the workload in our predictions. Similarly, our prediction model can be extended to represent hardware components or environmental influences using terms. Although this is future work, we laid the base with our prediction model to these new important research fields.

4. **Analysis of Customizable Programs and Non-functional Properties.** We provide our measurement raw material online to support other researchers in developing prediction approaches and other analyses based on customizable programs. For the first time, such a quantity of raw material is presented. On realistic programs, we did a brute-force approach as base line, although it cost a huge time investment. That is, we thoroughly evaluated all our approaches: we spent about five months of measurement with multiple computers to conduct the experiments presented in this thesis, which underlines the feasibility of our approach. We recognized already three research groups using our data. Considering the short amount of time at which this data is public, it clearly shows the need for such quantitative data and that we make an important contribution by making them available.

An important aspect during our evaluations was to use real-world programs. We contribute to the real world to make it possible to, for example, generate a variant of

SQLite with a specific performance and footprint. We show which features contribute positively or negatively to a non-functional property and to what extent.

We conclude that we found a *scalable and efficient* method to *accurately* predict *different* non-functional properties of customizable *black-box* programs.

9.3. Future Work

Our work lays the foundation for different research directions in the area of non-functional properties for customizable programs. Moreover, we plan to extend our work in the area of self-adaptive systems, because these systems have to cope with varying (mostly) non-functional requirements, leading to frequent reconfiguration. In the following, we highlight some promising directions.

9.3.1. Feature Libraries

Rather than optimizing non-functional properties by finding an optimized configuration, we can also optimize a program by refactoring. We proposed to use *feature libraries* as means to optimize a program for a user-specified non-functional property [Siegmund et al., 2010b]. Depending on the property to be optimized, we use a different feature library. A feature library contains algorithms of reoccurring functionality that are optimized for a specific non-functional property. As an example, we suggested the use of an energy feature library, which contains energy-efficient sorting algorithms (e.g., JouleSort [Rivoire et al., 2007]). Moreover, the library has features that control hardware devices in an energy-efficient manner. For example, we proposed to automatically switch the WLAN in a smart phone on and off depending on the functionality that is executed in the customizable program.

The energy features (e.g., to control hardware usage) have to be weaved into the client program, which is actually executed on a system. We proposed different weaving approaches based on aspectual feature modules [Apel et al., 2008] and aspect-oriented programming [Kiczales et al., 1997] to integrate the energy features. In future work, we extend these techniques to incorporate also other non-functional properties, such as performance. The combination with the ability to quantify the influence of a feature on a non-functional property allows us to selectively choose which features of the feature library to integrate in the client program.

9.3.2. Workload-Aware Prediction of Non-Functional Properties

A crucial factor for the prediction of non-functional properties is the workload of a program. The workload specifies which features are used how extensively at runtime. In our future work, we aim at describing the influence of features on non-functional

properties depending on the workload. That is, we determine cost functions for workload parameters, parameterize these cost functions using curve-fitting approaches, and build a prediction function. This prediction function is composed of the parameterized cost functions of each selected feature and identified feature interactions.

In Figure 9.1, we show initial results to predict a variant's non-functional properties with respect to variable workload parameters. In this example, we predict performance of Berkeley DB C version for three workload parameters:

- **Selects.** We chose selects (i.e., "gets" for key-value databases) as the first workload parameter, because selects are the most common form of using a database. Here, we are interested in how the features of Berkeley DB affect the response time of a select. For example, how does *Encryption* slow the response time.
- **Inserts.** We chose inserts (i.e., "puts" for key-value databases) as the second workload parameter, because, together with selects, we can describe a complete real-word workload.
- **Database size.** Finally, we select database size as a workload parameter, because it affects the response times of select and insert operations. That is, depending on how much data are already stored in a database, retrieving a certain data set requires more time.

With these three parameters, we can build cost functions; that is, what is the cost of a select for a certain size of a database when feature *Encryption* is selected.

In Figure 9.1, we show the changing response times (y-axis) for different workload parameters of selects (x-axis from center to left) and inserts (x-axis from center to right). We see that with an increasing number of inserts and selects, the response time increases, too. Since we provide only preliminary results, concrete values do not matter in Figure 9.1. Here, only the overall shape of the distribution of the measured and predicted performance is of interest.

At the top left corner, we depict the shape of the predicted cost function for this workload. At the bottom left corner, we show the error rate for our prediction against the measurement. As we can see, the overall function is appropriately approximated, but requires additional measurement points at the edges to be more accurate. Nevertheless, we show this figure to illustrate in which directions our research will follow. That is, it is possible to provide workload-aware predictions of non-functional properties.

The challenge of a workload-aware prediction is to find efficient sampling techniques for parameterization of the cost functions. Furthermore, this technique requires knowledge about the polynomial structure of cost functions. We aim at automatically identifying the structure of these cost functions. Here, the integration of code-metric analysis seems to be beneficial. Identifying a certain structure in the program may allow developers to automatically approximate the structure of a cost

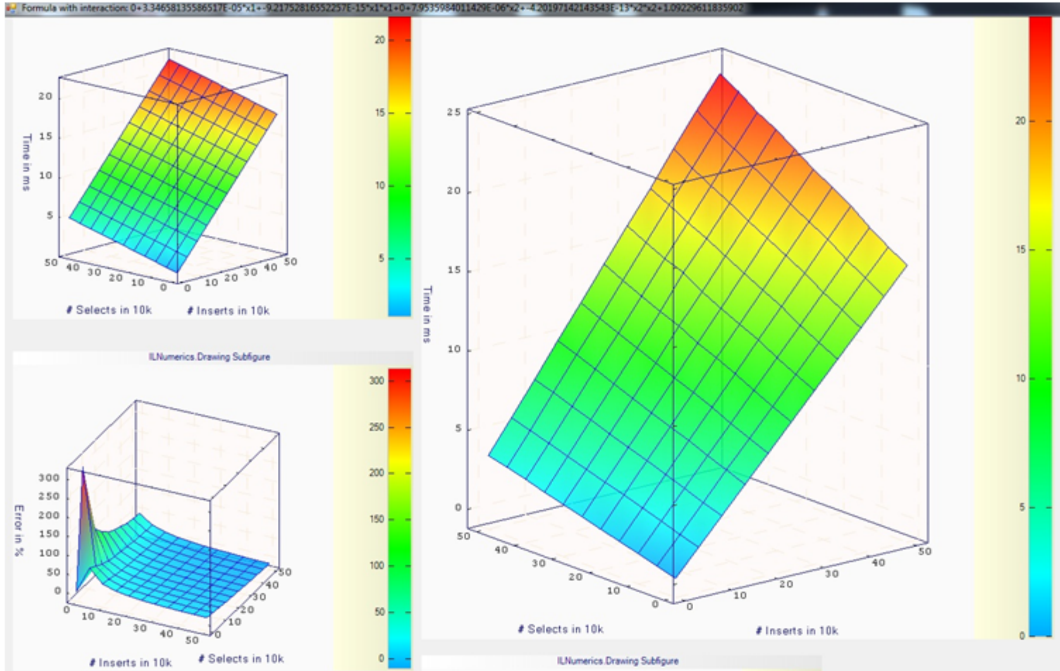


Figure 9.1.: Initial results to predict a variant's non-functional properties depending on workload parameters.

function, which then only needs to be parameterized. Similarly, the used hardware needs to be described with appropriate models. We analyze existing languages and tools, such as *LCFG* and *Karl* to be integrated in our prediction model.

9.3.3. Software Measures and Non-Functional Properties

The work in this thesis targets black-box programs. While this extends applicability of our approach, it may sacrifice prediction accuracy. Hence, in a new line of research, we aim at improving our predictions by using static code analysis to find non-functional feature interactions and to find patterns that degrade or improve certain properties.

As a first step, we plan to empirically identify which software measures (e.g., code metrics for internal and external feature dependencies) are good predictors for non-functional properties. This work is in line with the DFG funded research project *Pythia* (techniques and prediction models for sustainable product-line engineering).[1] Pythia aims at utilizing software-product-line analysis to improve predictions of non-functional properties.

[1] `http://www.infosun.fim.uni-passau.de/spl/pythia/`

Together, we compare these predictors with the empirical studies performed in this thesis to detect which architectural pattern and which variability mechanism are likely to cause feature interactions and degrade or improve certain properties, such as performance. Based on this, we plan to develop code-transformation techniques that optimize a certain non-functional property. These code transformations are partially based on previous work, in which we suggested the use of refactorings to improve performance [Siegmund et al., 2010a]. The combination with feature libraries seems to be beneficial for this line of research.

9.3.4. Self-Adaptive Systems

An important trend in current research is the development of self-adaptive systems [Sawyer et al., 2010]. Self-adaptive systems change their functionality and architecture depending on the current environment, context, and requirements. Hence, an adaptive system must provide means to customize itself according to ever changing requirements.

Important run-time requirements are the optimization of certain non-functional properties. For example, sensor networks have often self-adaptive nodes, which change their role depending on the context and non-functional property that must be optimized [Siegmund et al., 2009b]. Nodes can change their role to save as much energy as possible to extend life-time of the hardware and the overall network. Other roles are performance optimized to provide fast response times of network queries. Finally, sensor networks contain aggregation nodes, which improve reliability of data.

In previous work, we already investigated the possibility of reconfiguring a program at run-time to optimize different non-functional properties [Siegmund et al., 2009a, Rosenmüller et al., 2011b]. In future work, we combine our prediction model, including the feature-interaction detection, with self-adaptive systems. TOD This allows a program to build and extended the prediction model with monitoring data at run-time. That is, with each reconfiguration, we enrich our prediction model by measuring additional terms, such that our predictions become more accurate and a self-adaptive system finds to each context the optimal configuration. In this line of research, we are currently working on simulating these systems in the virtual reality to test and train reconfiguration for changing environmental influences [Rosenmüller et al., 2011a].

A. List of Case Studies

To evaluate our approach, we conducted a series of experiments. In the following, we describe the programs and software product lines we used in these experiments in detail. We describe attributes, such as origin, feature models, and features.

A.1. Case Studies for Footprint

LinkedList

Developed by:	Martin Kuhlemann (translated to Java by Norbert Siegmund)
Developed as:	software product line
Domain:	component
Origin:	academic
Language:	Java
Customization:	composition (Jak)
Features:	18
Variants:	492
Lines of code:	2 595

LinkedList was originally developed by Martin Kuhlemann in FeatureC++, a language extension to C++ to support feature-oriented programming. This version was translated to Java by Norbert Siegmund using the language Jak, an extension to Java to support feature-oriented programming. Further features of alternative sorting algorithms were added to allow users to generate a linked list component, which is optimized either for performance or for memory consumption. Features of LinkedList are:

- AbstractElement: Represents an abstract object, which can be stored in the list.
- AbstractSort: Represents an abstract sorting algorithm.
- BackwardInterator: Iterates the list from the end to the head.
- Base: Basic list functionality.
- BubbleSort: Implements the bubble-sort algorithm.
- Element A, B, and C: Represent alternative concrete objects to be stored in the list.
- ForwardIterator: Iterates the list from head to tail.
- InsertionSort: Implements the insertion-sort algorithm.

- Measurement: Includes basic monitoring functionality.
- MemorySize: Monitoring code to measure current main-memory consumption.
- MergeSort: Implements the merge-sort algorithm.
- Performance: Monitoring code to measure performance for the currently selected sorting algorithm.
- Print: Prints the elements of the list.
- QuickSort: Implements the quick-sort algorithm.
- TCP/IP: Sends monitoring data to a network address.

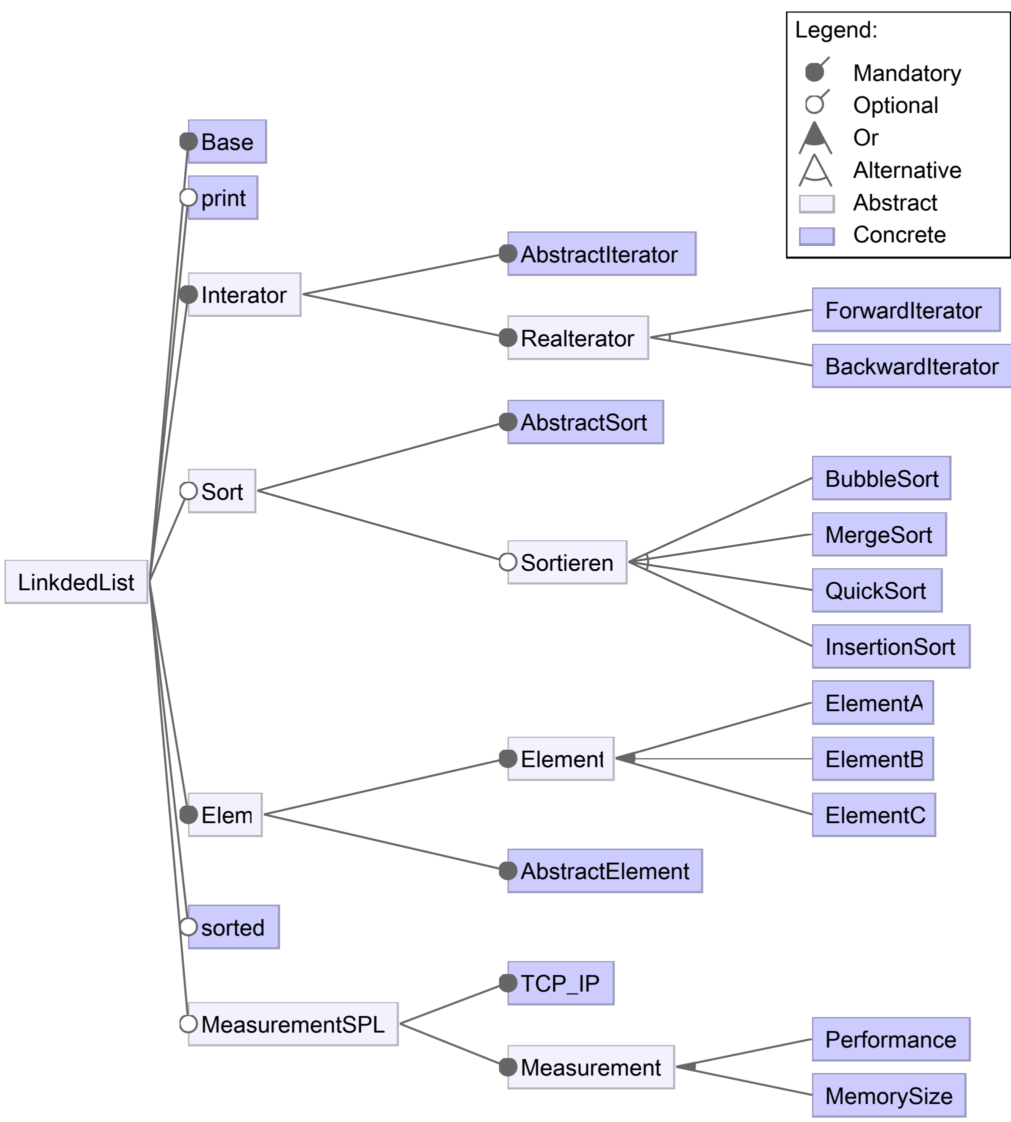


Figure A.1.: Feature model of LinkedList.

Prevayler

Developed by:	Klaus Wuestefeld and others
Developed as:	single library
URL:	`http://www.prevayler.org`
Domain:	persistence library
Origin:	industrial
Language:	Java
Customization:	conditional compilation
Features:	5
Variants:	24
Lines of code:	4 030

Prevayler is a customizable object-persistence library. It can be embedded into existing Java programs to enable efficient in-memory data storage. Prevayler was not initially developed as a customizable program, but it was refactored by other researchers Godil and Jacobsen [2005] such that features were extracted to be user-selectable. We use a version in which variability is realized with preprocessor annotations.

- Base: Basic functionality.
- Censor: Allows roll backs of transactions.
- Gzip: Compression support.
- Monitor: Enables monitoring functionality.
- Replication: Supports duplicating contents between a server and multiple clients..
- Snapshot: Enables to do snapshots of the current database status.

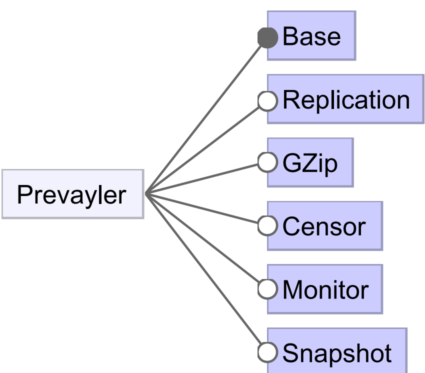


Figure A.2.: Feature model of Prevayler.

ZipMe

Developed by:	Akihiko Kusanagi and others
Developed as:	single library
URL:	`http://zipme.sourceforge.net/`
Domain:	compression library
Origin:	industrial
Language:	Java
Customization:	composition (Jak)
Features:	8
Variants:	104
Lines of code:	4 874

ZipMe is an open-source Java library, which allows developers to compress data. It was refactored by other reserachers [] to make library customizable. In our used version, it has the following features:

- CRC: Implements CRC32 data checksum of a data stream.
- ArchiveCheck: Checks archives for checksums.
- GZIP: GZIP compression algorithm.
- Adaptation: Exception support.
- Adler32Checksum: Includes checksums in the archive.
- Compress: Standard compression engine.
- Extract: Standard extraction engine.

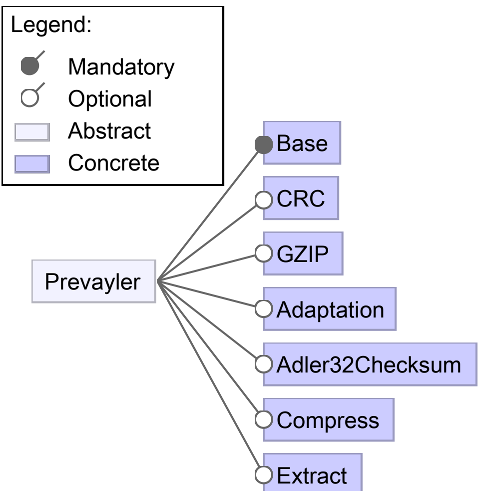


Figure A.3.: Feature model of ZipMe.

PKJab

Developed by:	student project at University of Passau
Developed as:	software product line
URL:	`http://zipme.sourceforge.net/`
Domain:	messenger
Origin:	academical
Language:	Java
Customization:	composition (FeatureHouse)
Features:	11
Variants:	73
Lines of code:	5 016

PKjab was developed during a student project. It allows asynchronous messaging between remote computers.

- Base: Basic messaging functionality.
- ContactListSource: Implementation of a contract list.
- ServerRoster: Integrates the XMPP protocol.
- Composing and SendComposing: Basic messaging support and notification of typing a message.
- History: Persistent storage of messaging history.
- Timestamps: Support for timestamps for message sending.
- ThemeSelection: Support for different user-interface themes.

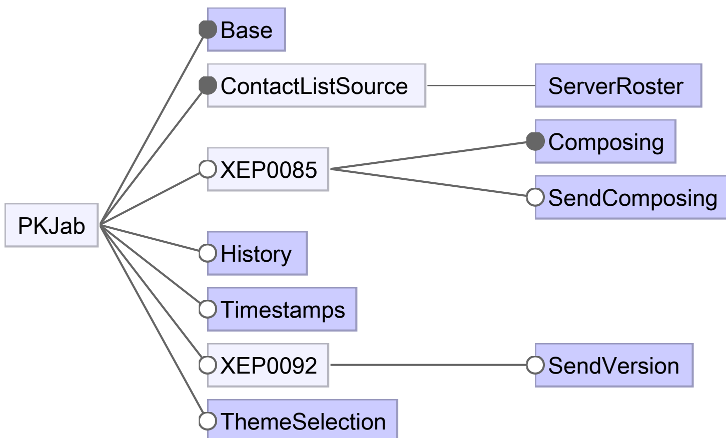


Figure A.4.: Feature model of PKJab.

SensorNetwork

Developed by:	Marko Rosenmüller
Developed as:	software product line
Domain:	simulation
Origin:	academical
Language:	C++
Customization:	composition (FeatureC++)
Features:	26
Variants:	3 240
Lines of code:	7 303

The SensorNetwork is a software product line, which simulates different interconnecting sensor nodes. The product line can be customized for different sensor types (sensor node, aggregation node, entrance node) and with varying functionality (e.g., data storage and stream processing). It was developed in a research project to demonstrate flexibility of feature bindings [Rosenmüller et al., 2011b].

- Http: Provides access to sensor nodes via the http protocol.
- Communication: Provides different communication schemes with its child features (i.e, client, server, multicast).
- ConnectSensors: Connects the sensors in the simulation such that data is transferred.
- Routing: Different routing schemes (i.e., routing by query or routing via connected sensors).
- Data: Provides two methods to handle data (i.e., aggregate data or stream data).
- Threading: Provides single and multi threading for sensors.
- Sensor: Includes a simple simulation program on the sensor.
- Storage: Supports different database systems for storing data on a node.
- Statistics: Collects communication statistics.

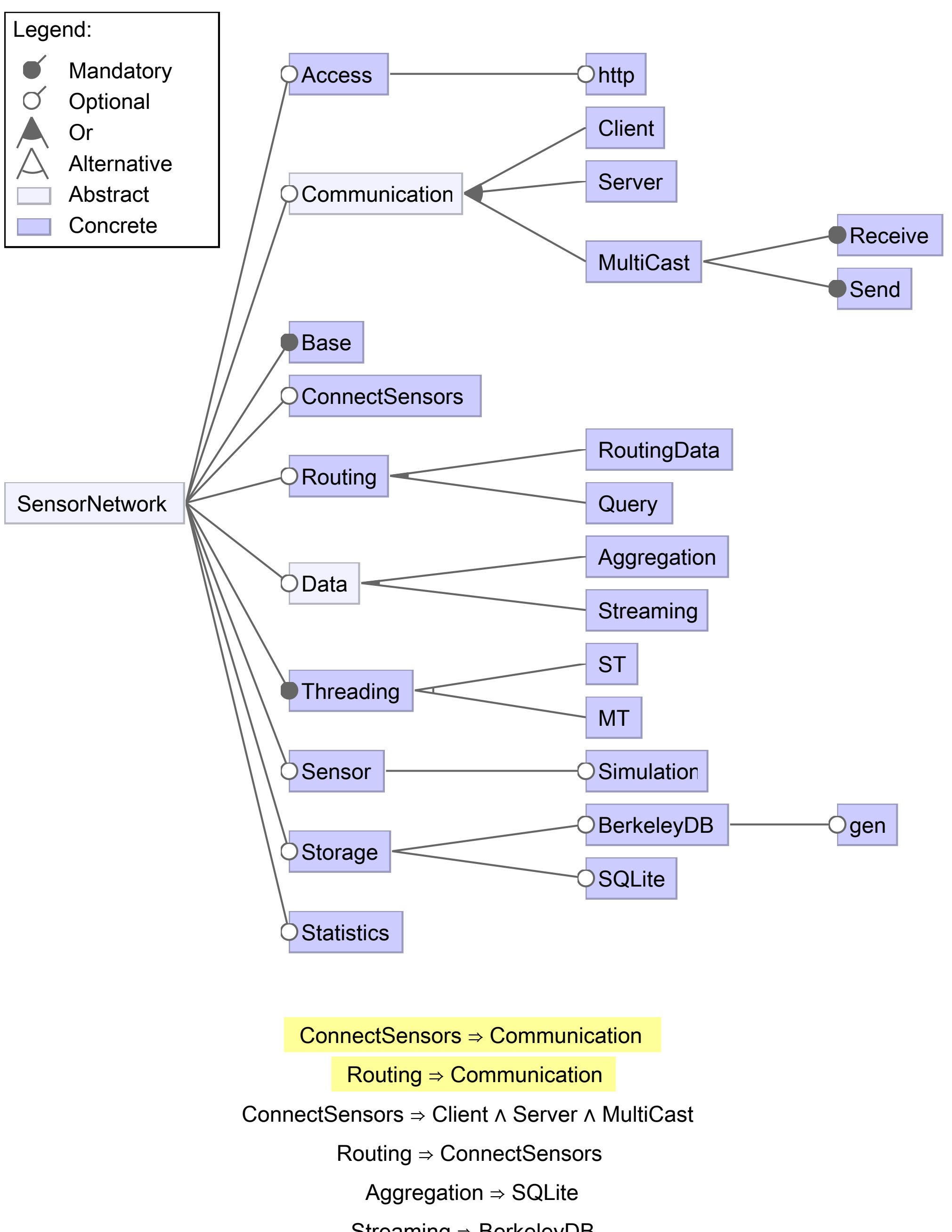


Figure A.5.: Feature model of SensorNetwork.

Violet

Developed by:	Cay S. Horstmann and Alexandre de Pellegrin
Developed as:	software product line
URL:	`http://alexdp.free.fr/violetumleditor/page.php`
Domain:	diagram editor
Origin:	industrial
Language:	Java
Customization:	composition (FeatureHouse)
Features:	100
Variants:	10^{20}
Lines of code:	19 379

Violet is a graphical editor, which supports different types of diagrams (e.g., UML diagram, state-chart diagram). It was originally not developed as a customizable program, but refactored in a student project at the University of Passau. It supports a large number of features. We describe only a subset of them in the following:

- ObjectDiagram: Support for UML object diagrams.
- SequenceDiagram: Support for sequence diagrams.
- StateDiagram: Support for state-chart diagrams.
- ClassDiagram: Support for UML class diagrams.
- UseCaseDiagram: Support for use-case diagrams.
- File: Implements the file menu with features as open file, save file, and export image.
- Edit: Allows to edit diagrams.
- View: Provides basic functions for the drawing area (e.g., zooming, clip area, show grid on screen).
- Window: Realizes Windows look-and-feel buttons.
- Additional: Implements various functions, such as command-line support, image filters, preferences, version checker.

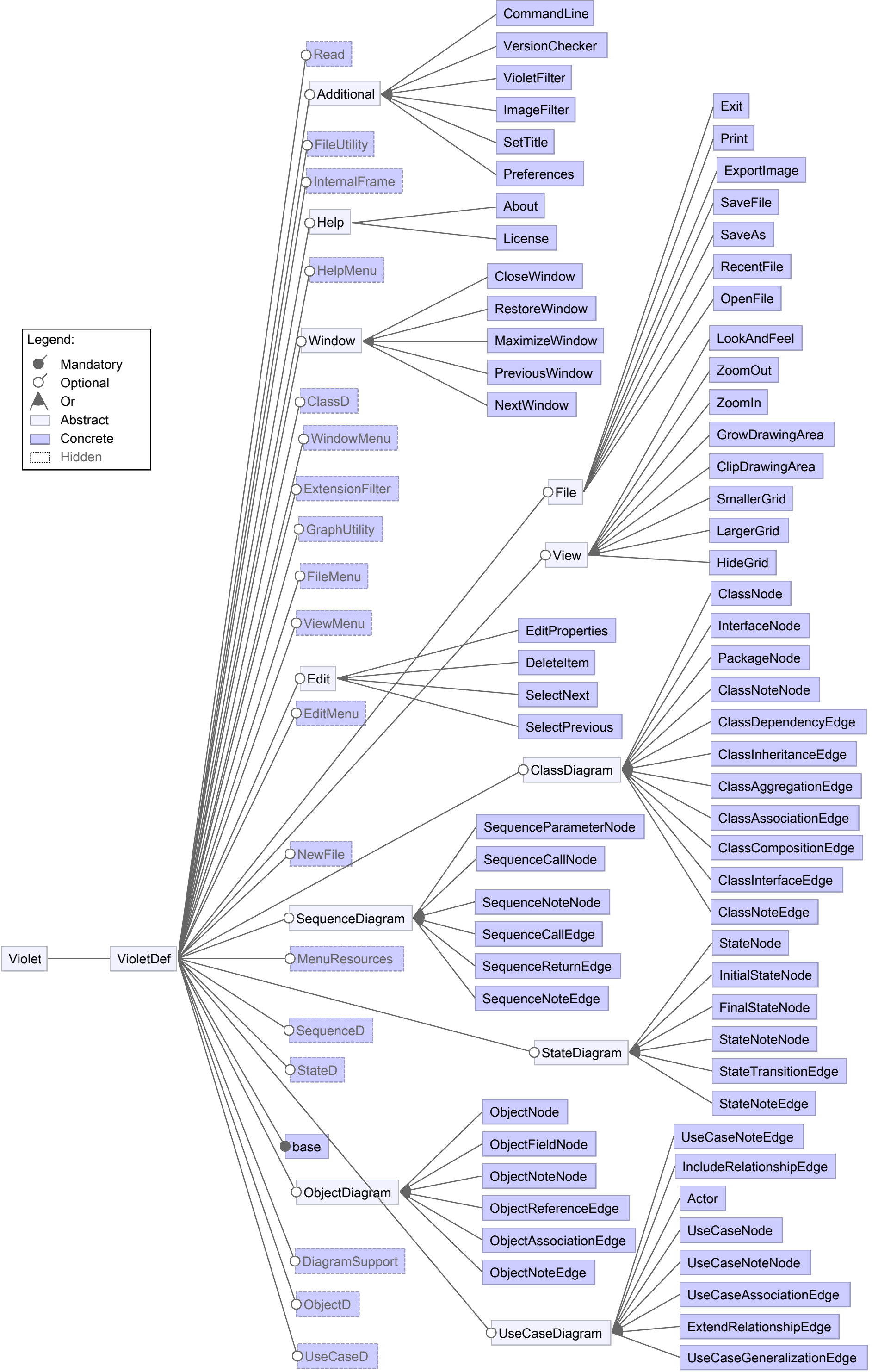


Figure A.6.: Feature model of Violet.

File ∨ NewFile ∨ Edit ∨ View ∨ Window ∨ Help ∨ Additional ⇒ MenuResources ∧ GraphUtility ∧ ExtensionFilter ∧ FileUtility ∧ InternalFrame ∧ Read
File ∨ NewFile ⇒ FileMenu
Edit ⇒ EditMenu
View ⇒ ViewMenu
Window ⇒ WindowMenu
Help ⇒ HelpMenu
ClassDiagram ∨ SequenceDiagram ∨ StateDiagram ∨ ObjectDiagram ∨ UseCaseDiagram ⇒ DiagramSupport ∧ NewFile
ClassDiagram ⇒ ClassD
SequenceDiagram ⇒ SequenceD
StateDiagram ⇒ StateD
ObjectDiagram ⇒ ObjectD
UseCaseDiagram ⇒ UseCaseD
RecentFile ⇒ OpenFile
SaveFile ⇒ SaveAs
ImageFilter ⇒ ExtensionFilter ∧ ExportImage
VioletFilter ⇒ ExtensionFilter
CommandLine ⇒ OpenFile
ClassDependencyEdge ∨ ClassInheritanceEdge ∨ ClassAggregationEdge ∨ ClassAssociationEdge ∨ ClassCompositionEdge ∨ ClassInterfaceEdge ⇒ ClassNode
ClassNoteEdge ⇒ ClassNoteNode
SequenceCallEdge ∨ SequenceReturnEdge ⇒ SequenceCallNode
SequenceNoteEdge ⇒ SequenceNoteNode
StateTransitionEdge ⇒ StateNode
StateNoteEdge ⇒ StateNoteNode
ObjectReferenceEdge ∨ ObjectAssociationEdge ⇒ ObjectNode
ObjectNoteEdge ⇒ ObjectNoteNode
UseCaseAssociationEdge ∨ ExtendRelationshipEdge ∨ IncludeRelationshipEdge ∨ UseCaseGeneralizationEdge ⇒ Actor
UseCaseNoteEdge ⇒ UseCaseNoteNode

Figure A.7.: Boolean constraints of Violet.

Berkeley DB (Version 4.4.20)

Developed by:	University of California, Berkeley
Developed as:	customizable embedded database system
URL:	`http://www.oracle.com/technetwork/products/berkeleydb/`
Domain:	database
Origin:	industrial
Language:	C
Customization:	conditional compilation
Features:	8
Variants:	256
Lines of code:	209 682

Berkeley DB was originally developed by University of California. Developers of Berkeley DB founded the company Sleepycat Software, which added new features. Each release version introduced a new major feature: 1.x introduced key/value storage, 2.x introduced locking functionality to support concurrent data access, 3.x added transaction support, such as logging functionality and recovery, 4.x introduced replication to improve availability of data. Today, Oracle maintains three versions of Berkeley DB: Berkeley DB C version (which is more or less the original version), Berkeley DB Java version with a complete different code base, and Berkeley DB XML. For our footprint evaluation, we use the C version 4.4.20 of Berkeley DB. We slightly modified the code to ease generation of customized variants. That is, we moved functionality from the make file that was responsible to tailor the program to `#ifdefs` in the source code. This way, the preprocessor removes code of unselected features and we compile only selected features.

- Base: Basic database functionality.
- Crypto: Encrypts data pages and log files using advanced encryption standard (AES).
- Hash: Implements a hash-based search index.
- Queue: Realizes list and queue data structures.
- Replication: Replicates data to improve availability.
- Verify: Verifies the integrity of all the database.
- Sequence: Support for sequential data structures.
- Diagnostic: Diagnostic functions.
- Statistics: Statistics about search indexes, stored data, etc.

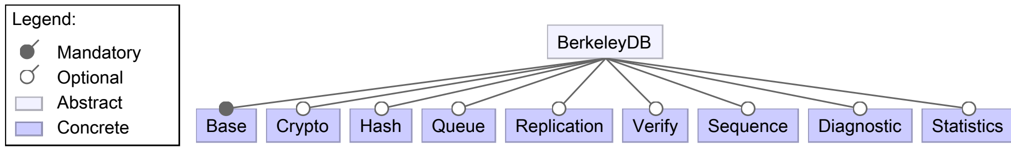


Figure A.8.: Feature model of Berkeley DB.

SQLite (Version 3.7.4)

Developed by:	D. Richard Hipp and others
Developed as:	customizable embedded database system
URL:	`http://sqlite.org`
Domain:	database
Origin:	industrial
Language:	C
Customization:	conditional compilation
Features:	88
Variants:	ca. 10^{23}
Lines of code:	305 191

SQLite is a customizable relational database system implemented to be small and fast. It is ACID-compliant and supports most of the SQL standard. SQLite is likely the most widely used SQL-based database in the world with over 500 million deployments. It is mainly used for embedded systems (e.g., smart phones), application software (e.g., Mozilla Firefox), and other programs (e.g., PHP, McAfee, and Skype). The source code is under public domain. We selected features from the available compilation options. Since there are over 80 compilation options, we do not list them here, but refer to SQLite's website, which provides a detail description: `http://www.sqlite.org/compile.html`.

Slackware Linux Kernel

Developed by:	Patrick Volkerding and others
Developed as:	customizable operating system
URL:	`http://www.slackware.com/`
Domain:	operating system
Origin:	industrial
Language:	C
Customization:	conditional compilation
Features:	25 (selected by domain expert)
Variants:	ca. $3{\cdot}10^{24}$
Lines of code:	13 005 842

Slackware Linux is a Linux derivate, which aims at simplicity and ease of use. It was released in April 1993 by Patrick Volkerding. Since it is Linux compliant, it supports a large number of features (called packages). A domain expert selected 25 features that can be arbitrary configured to compile a tailor-made kernel.

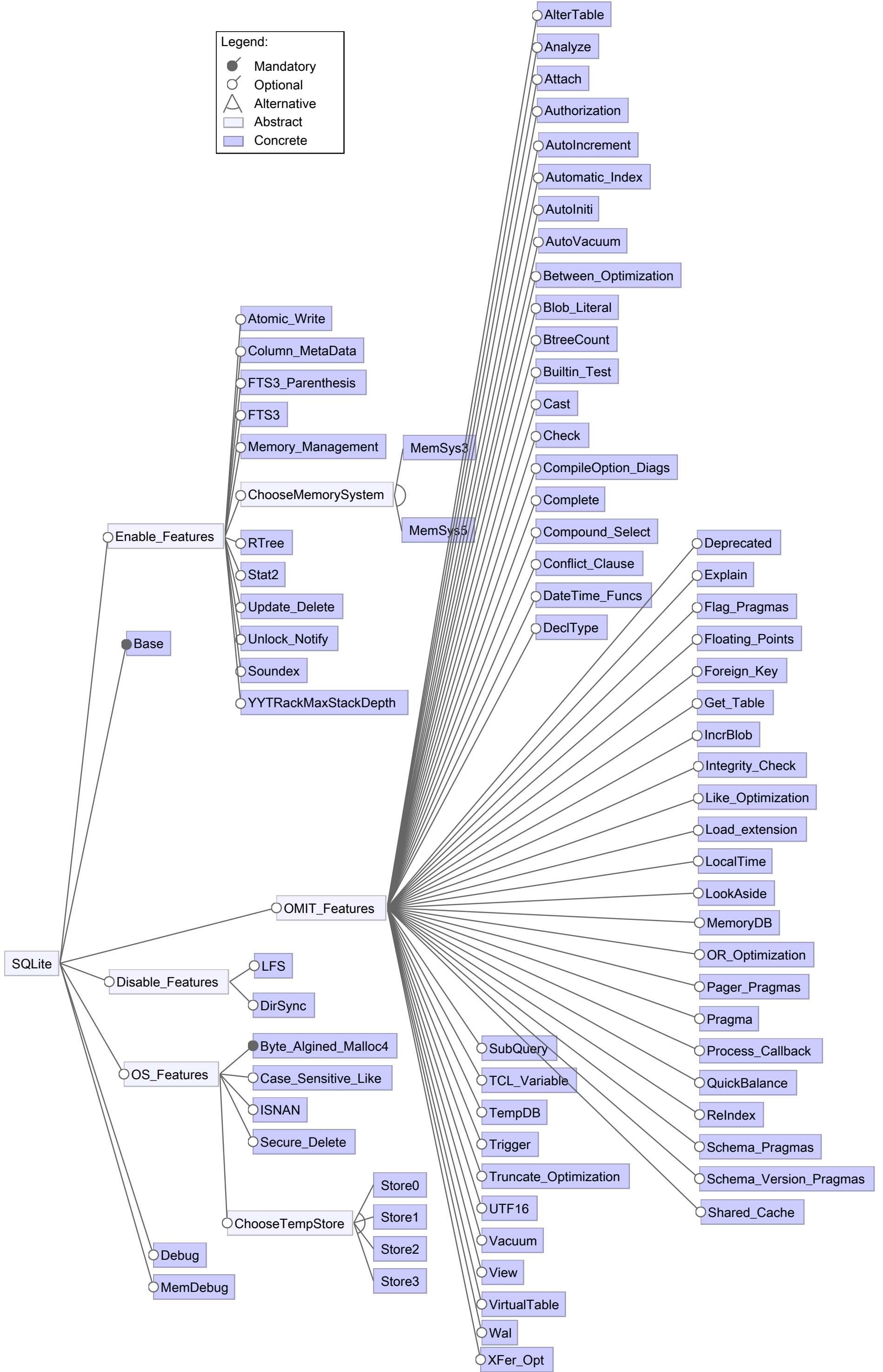


Figure A.9.: Feature model of SQLite.

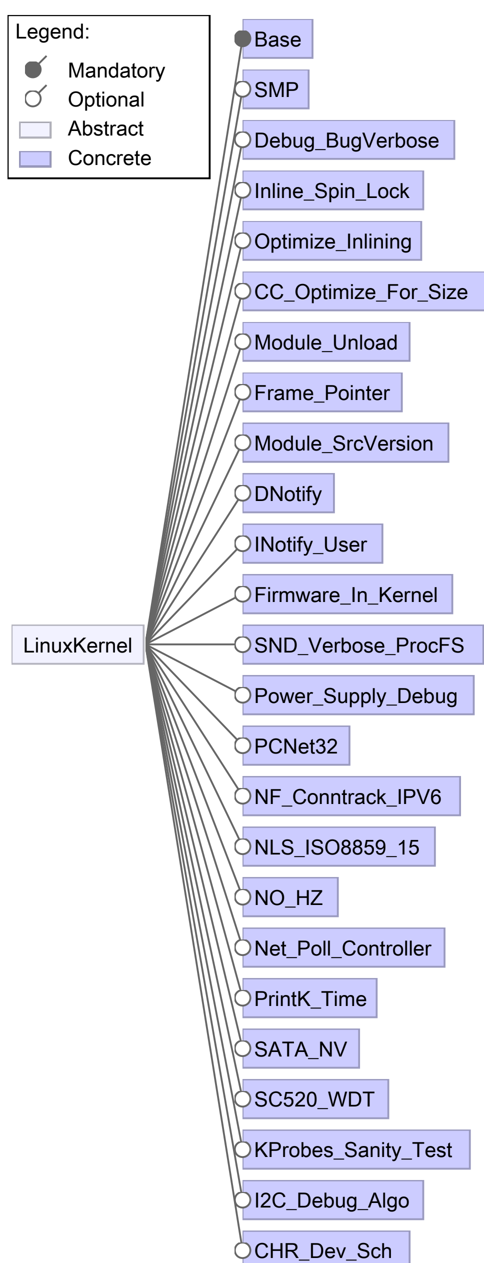


Figure A.10.: Feature model of selected features of the Linux kernel.

A.2. Case Studies for Main-Memory Consumption

Curl

Developed by:	Daniel Stenberg and others
Developed as:	command-line tool
URL:	`http://http://curl.haxx.se/`
Domain:	data transfer
Origin:	industrial
Language:	C
Customization:	command-line parameters
Features:	13
Variants:	768
Lines of code:	52 341

Curl is a tool to transfer data with URL syntax and was released in 1997. It supports various protocols, such as FTP, HTTP, and SCP. Curl can be customized via command-line parameters.[1]

- Compressed: Request a compressed response from a server.
- Dump_Header: Writes the protocol to a file.
- Include: Includes the HTTP-header in the output.
- No_Buffer: Disables buffering of the output stream.
- Trace: Enables a full trace dump of all incoming and outgoing data.
- Verbose: Makes the fetching more verbose/talkative.
- Silent: Set curl silent.
- Raw: Disables all internal HTTP decoding of content.
- HTTP1_0: Forces curl to issue its requests using HTTP 1.0.
- Progress_Bar: Visualizes a simple progress bar.
- No_KeepAlive: Disables the use of keepalive messages on a TCP connection.

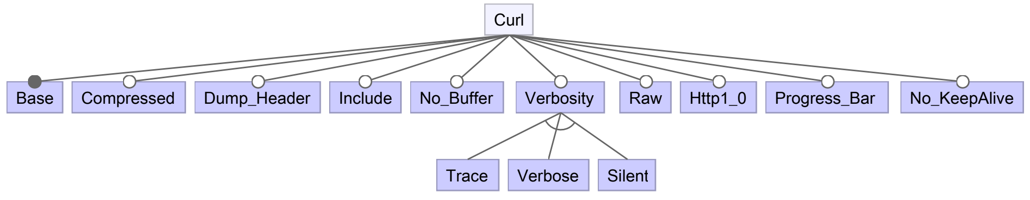


Figure A.11.: Feature model of Curl.

[1] See website for a full list of customization options: `http://curl.haxx.se/docs/manpage.html`.

LLVM

Developed by:	University of Illinois at Urbana-Champaign
Developed as:	research infrastructure to investigate dynamic compilation
URL:	`http://llvm.org/`
Domain:	compiler infrastructure
Origin:	industrial
Language:	C++
Customization:	command-line parameters
Features:	11
Variants:	1 024
Lines of code:	47 549

LLVM stands for low level virtual machine. It is a compiler infrastructure to optimize programs at compile-time, link-time, and run-time. To this end, LLVM implements layers in a compiler system, in which it take intermediate form (IF) code from a compiler and emitting an optimized IF. Since its publication in 2000, LLVM was extended with several front-ends to support different programming languages and is now used in Apple's development system. All customizations options can be found here: `http://llvm.org/docs/Passes.html`.

- Time_passes: Records the amount of time needed for each pass and print it to standard error.
- Gvn: Enables blobal value numbering (GVN) to assign unique value numbers to variables and expressions (across basic blocks) having the same static value.
- InstCombine: Combines redundant instructions.
- Inline: Specifies Whether inlining is used as a compile-time optimization.
- Jump_Threading: Finds distinct threads of control flow running through a basic block and optimizes them.
- SimplifyCfg: Simplifies the CFG (performs dead code elimination and basic block merging).
- Sccp: Enables sparse conditional constant propagation.
- Print_used_Types: Prints the internally used types.
- IPSccp: Enables inter-procedual sparse conditional constant propagation.
- IV_Users: Induction variable users,
- Licm: Enables loop invariant code motion.

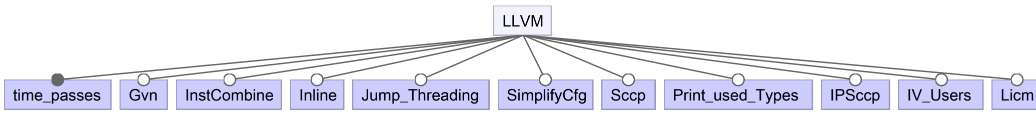


Figure A.12.: Feature model of LLVM.

x264

Developed by:	Laurent Aimar and others
Developed as:	software library
URL:	http://www.videolan.org/developers/x264.html
Domain:	video encoding
Origin:	industrial
Language:	C
Customization:	command-line parameters
Features:	16
Variants:	1 152
Lines of code:	45 743

x264 is a free video encoding library. It implements h264 encoding and many other encoding algorithms. x264 is customizable via command-line parameter or via API calls when used as a library. A complete list of parameters can be found here: http://mewiki.project357.com/wiki/X264_Settings.

- No_asm: Disables all CPU optimiztions.
- No_8x8dct: Disables intelligent adaptive use of 8x8 transforms in I-frames.
- No_cabac: Disables context adaptive binary arithmetic coder stream compression and uses instead the less efficient context adaptive variable length coder.
- No_deblock: Disables the loop filter (coding time vs. quality).
- Rc_LookAhead: Sets the number of frames to use for mb-tree rate control and vbv-lookahead.
- No_fast_pskip: Disables early skip detection on P-frames (quality increase vs. performance degrade).
- Ref: Sets the size of the decoded picture buffer.
- No_mixed_refs: Select refs on a per-macro block basis (reduces quality).
- No_MBtree: Disable macro block tree rate control.
- No_wightb: Disables weighing of references in B-frames.

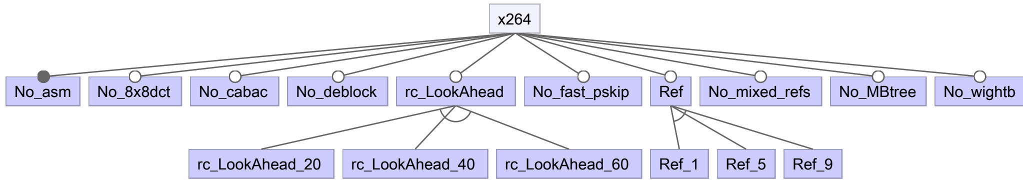


Figure A.13.: Feature model of x264.

Wget

Developed by:	Giuseppe Scrivano and others
Developed as:	software package
URL:	http://www.gnu.org/software/wget/
Domain:	retrieving files
Origin:	industrial
Language:	C
Customization:	command-line parameters
Features:	16
Variants:	5 120
Lines of code:	34 880

Wget is a free software package to download contents and retrieve files from web servers using multiple Internet protocols, such as HTTP, HTTPS and FTP. Customization is done via a command-line interface. A full list of options can be found here: `http://www.gnu.org/software/wget/manual/html_node/Wgetrc-Commands.html`.

- Output_File: Specifies an output file for the download.
- Server_Response: Prints the HTTP and FTP server responses.
- No_Clobber: Disables overwriting of already downloades files.
- No_DNS_Cache: Disables caching of DNS lookups.
- Verbosity: Defines the amount of messages.
- No_Directories: Ignores directories.
- No_Host_Directories: Ignores directories at Host side.
- No_Http_Keep_Alive: Disables keep alive messages.
- No_Cookies: Disables cookies.
- Save_Headers: Saves file headers.
- Convert_Links: Enables the conversion of non-relative links locally.
- Strict_Comments: Escaping of characters in comments.

Berkeley DB (Version 4.4.20)

The basic version is the same as for footprint measurements. We included features that set different cache and page sizes.

SQLite (Version 3.7.4)

The basic version of SQLite is the same as for footprint measurements. We removed features that do not affect performance or main-memory consumption of the used benchmark. We added features to set different cache sizes.

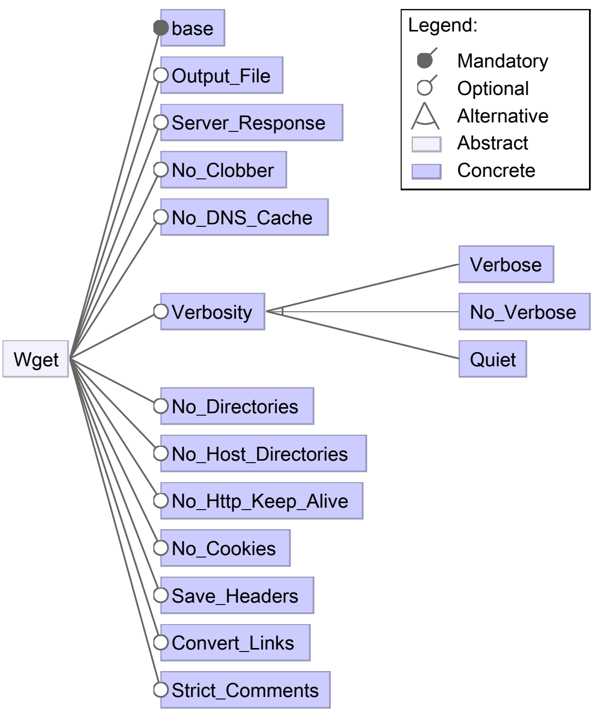


Figure A.14.: Feature model of Wget.

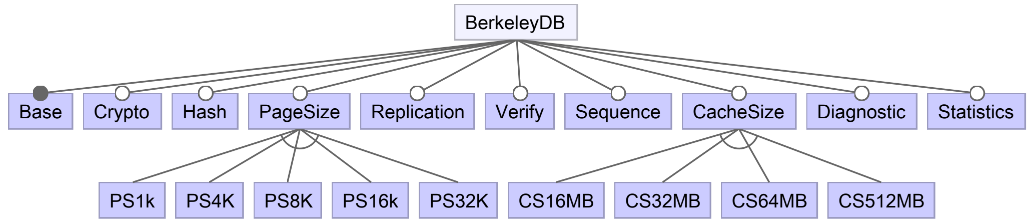


Figure A.15.: Feature model of Berkeley DB for main-memory consumption and performance measurements.

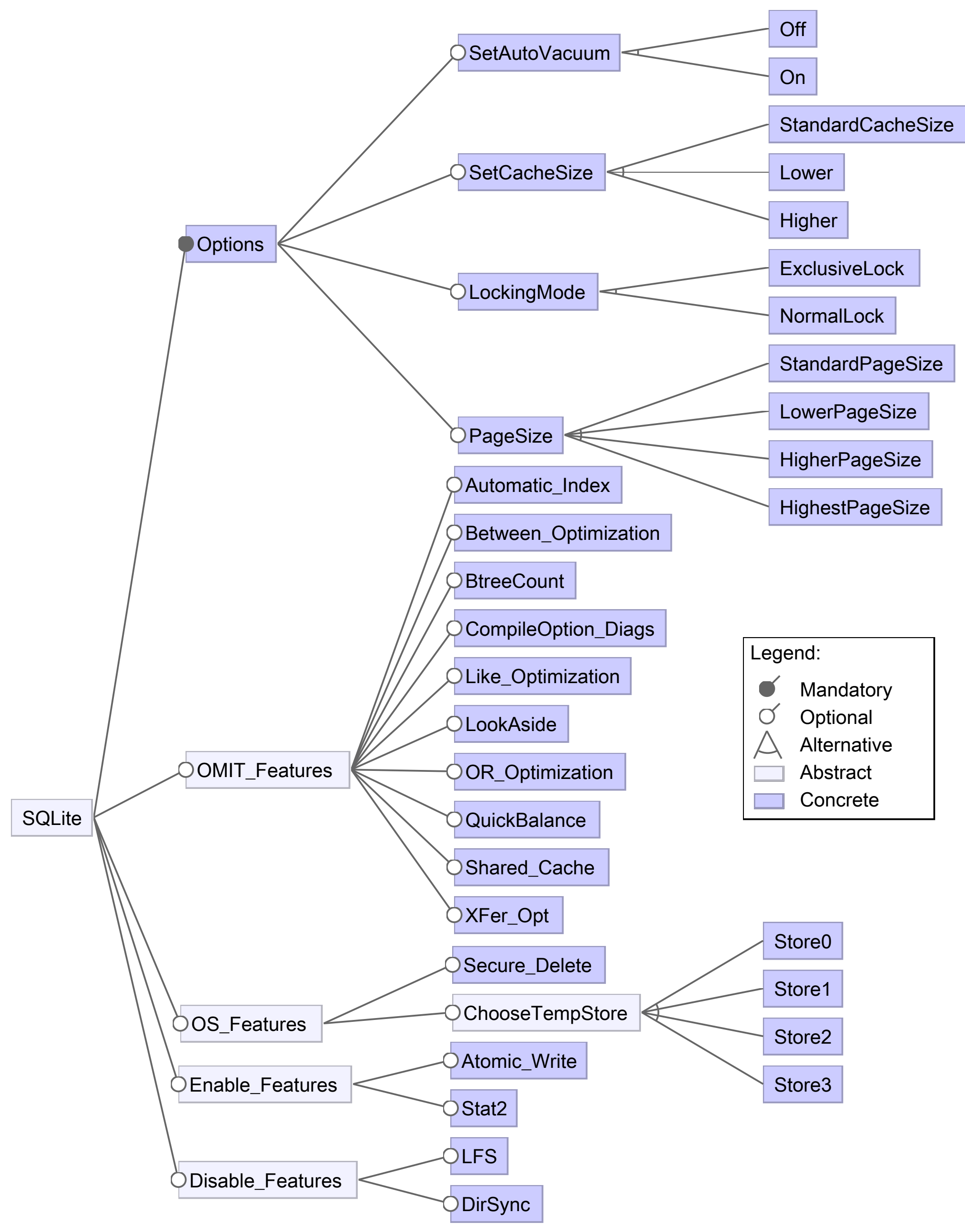


Figure A.16.: Feature model of SQLite for main-memory consumption and performance measurements.

A.3. Case Studies for Performance

Most of the case studies were already presented in the previous sections. We describe only the two new programs: Berkeley DB Java version and Apache.

Berkeley DB JE

Developed by:	Oracle
Developed as:	non-customizable embedded database
URL:	`http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html`
Domain:	database
Origin:	industrial
Language:	Java
Customization:	conditional compilation
Features:	32
Variants:	400
Lines of code:	42 596

Berkeley DB Java edition is an open source persistent layer for Java objects with full ACID compliance. Hence, it supports transactional storage including object-oriented information, such as object graphs and objects in collections. However, unlike the C version, the Java version is not customizable. Hence, we refactored this version and extracted 32 features. Since it is a legacy system, there are many dependencies among these 32 features such that there are only 400 different variants. We extracted features for alternative versions of input/output functionality (i.e., disk storage), for different btree features, such as Evictor pattern (to free resources), alternative cache sizes, and verifier functionality. Furthermore, we extracted different tracing levels and statistics as individual features.

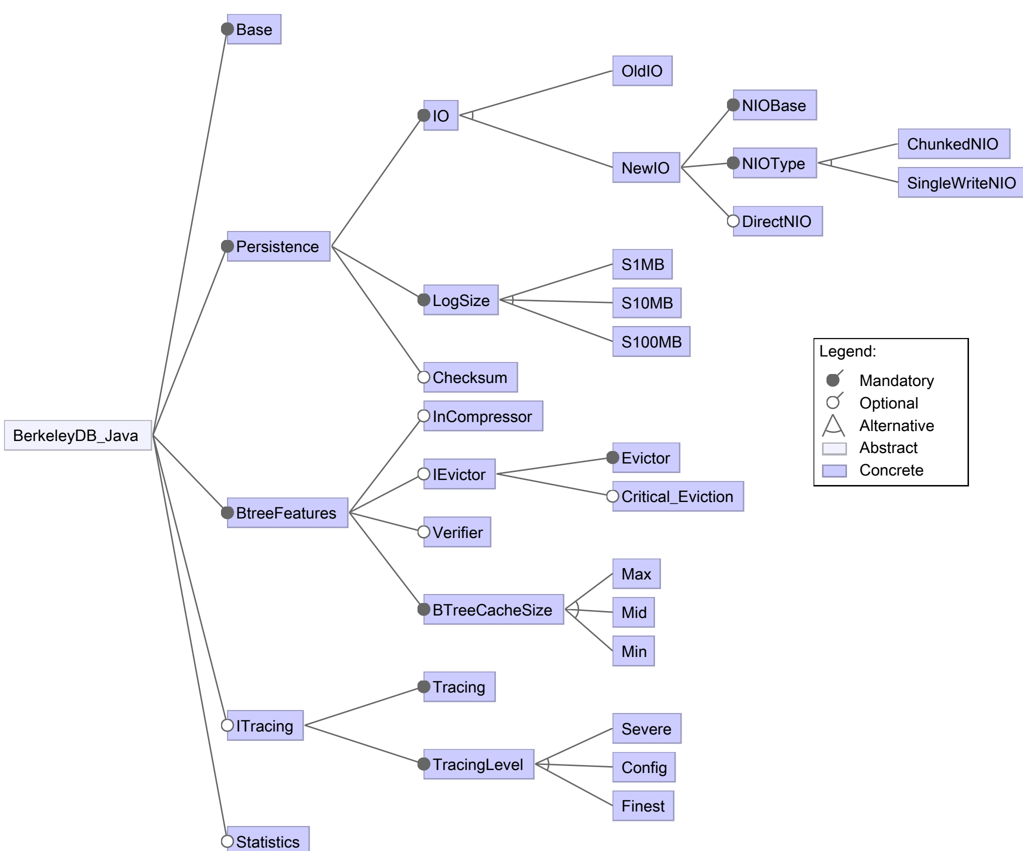


Figure A.17.: Feature model of Berkeley DB Java version.

Apache

Developed by:	Rober McCool and Apache Software Foundation
Developed as:	customizable web server
URL:	`http://httpd.apache.org/`
Domain:	HTTP web server
Origin:	industrial
Language:	C
Customization:	configuration files
Features:	9
Variants:	192
Lines of code:	230 277

The Apache web server is one of the most widely used web servers. It is open source and customizable via a configuration file. See the website for all customization options: `http://httpd.apache.org/docs/current/en/mod/core.html`.

- HostnameLookups: Enables DNS lookups, such that host names can be logged.
- KeepAlive: Enables persistent and long running connections.
- EnableSendfile: Avoids distinct read and send operations.
- FollowSymLinks: Apache follows symbolic links in a directive.
- AccessLog: Logs the data access on the server.
- ExtendedStatus: Enables an extended status information for each request.
- InMemory: Enables memory mapping.
- Handle: Enforces the server to operate all data with a single handle.

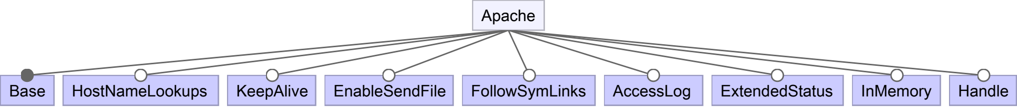


Figure A.18.: Feature model of Apache.

Remaining Programs

Berkeley DB CE, SQLite, LLVM, and x264 are identical case studies as used for the main-memory measurement.

Bibliography

Adil A. Abdelaziz, Wan M.N. Wan Kadir, and Addin Osman. Comparative analysis of software performance prediction approaches in context of component-based system. *International Journal of Computer Applications*, 23(3):15–22, 2011.

Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for Microsoft SQL server 2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121. VLDB Endowment, 2004.

Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *Proceedings of the International Conference on Software and Systems Engineering and their Applications (ICSSEA)*. CNAMCMSL, 2003.

Vander Alves, Daniel Schneider, Martin Becker, Nelly Bencomo, and Paul Grace. Comparative study of variability management in software product lines and runtime adaptable systems. In *Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–17. ICB Research Report, University of Duisburg-Essen, 2009.

Theodore Anderson and Jeremy Finn. *The New Statistical Analysis of Data*. Springer, 1996. ISBN 0-387-94619-5.

Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *Proceedings Workshop on Eclipse Technology eXchange*, pages 67–72. ACM, 2004.

Ana I. Anton. *Goal identification and refinement in the specification of software-based information systems*. PhD thesis, 1997.

Sven Apel and Dirk Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceeding of the International Conference on Software Engineering (ICSE)*, pages 421–430. ACM, 2011.

Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140. Springer, 2005.

Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE, 2009.

Sven Apel, Christian Kästner, Armin Grölinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3): 251–300, 2010a.

Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting dependences and interactions in feature-oriented design. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–170. IEEE, 2010b.

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering (TSE)*, 2012. to appear; submitted 21 Oct 2010, accepted 29 Nov 2011.

Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310, 2004.

Albert L. Barabási, Réka Albert, and Hawoong Jeong. Mean field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272: 173–187, 1999.

Victor R. Basili. Software modeling and measurement: The goal/question/metric paradigm. Technical Report CS-TR-2956 (UMIACS-TR-92-96), University of Maryland at College Park, 1992.

Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.

Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. In *Proceedings of the International Conference on Generative Programming (GPCE)*, pages 13–22. ACM, 2011.

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 491–503. Springer, 2005.

David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134. Lero Technical Report 2007-01, 2007.

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6): 615 – 636, 2010.

Antonia Bertolino and Raffaela Mirandola. Towards component-based software performance engineering. In *Proceedings of ICSE Workshop on Component-based software engineering*, pages 1–6, 2003.

Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. Macleod, and Michael J. Merritt. *Characteristics of Software Quality*. Elsevier, 1978. ISBN 0444851054.

Goetz Botterweck, Daren Nestor, André Preußner, Ciarán Cawley, and Steffen Thiel. Towards supporting feature configuration by interactive visualization. In *Proceedings of Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 125–131. IEEE, 2007.

Leo Breiman, Jerome Friedman, Richar Olshen, and Charles Stone. *Classification and Regression Trees*. Wadsworth, 1984.

Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of software product lines with delta-oriented slicing. In *Proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOO)*, pages 61–75. Springer, 2011.

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

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks and Isdn Systems*, 41:115–141, 2003a.

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer and Networks*, 41(1):115–141, 2003b.

Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Using feature models for developing self-configuring smart homes. In *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 179–188. IEEE, 2009.

Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *Journal of Systems and Software*, 74(1):35–43, 2005.

Lawrence Chung and Julio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual Modeling: Foundations and Applications*, volume 5600 of *LNCS*, chapter 19, pages 363–379. Springer, 2009. ISBN 978-3-642-02462-7.

Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 1999. ISBN 0792386663.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. ISBN 0201703327.

David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13 (5):83–88, 1996.

Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0201309777.

Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34. IEEE, 2007.

Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 266–283. Springer, 2004.

Dave Zubrow and Gary Chastek. Measures for software product lines. Technical Report CMU/SEI-2003-TN-031, Software Engineering Institute, 2003.

Alan M. Davis. *Software requirements: Objects, functions, and states*. Prentice Hall, 1993. ISBN 9780138057633.

Florian Deissenboeck, Elmar Juergens, Klaus Lochmann, and Stefan Wagner. Software quality models: Purposes, usage scenarios and requirements. In *Proceedings of Workshop on Software Quality*, pages 9 –14. IEEE, may 2009.

Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23:62–70, 2006.

Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 592–603. IEEE, 2009.

David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the Internatial Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 57–76. ACM, 2007.

Martin Glinz. On non-functional requirements. In *International Conference on Requirements Engineering*, pages 21–26. IEEE, 2007.

Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The COMQUAD component model: Enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of the International Conference on Aspect-oriented software development (AOSD)*, pages 74–82. ACM, 2004.

Irum Godil and Hans-Arno Jacobsen. Horizontal Decomposition of Prevayler. In *Proceedings of the International Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100. IBM Press, 2005.

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

Jens Happe. Predicting mean service execution times of software components based on Markov models. *Quality of Software Architectures and Software Quality*, 3712: 53–70, 2005.

Jens Happe, Heiko Koziolek, and Ralf Reussner. Facilitating performance predictions using software components. *IEEE Software*, 28(3):27 –33, 2011.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of statistical learning: Data Mining, Inference, and Prediction.* Springer, 2009.

Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, pages 149–155. ACM, 2012.

Frank Hunleth and Ron K. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 38–45. ACM, 2002.

International Organization for Standardization (ISO). Part 1: General overview. In *Information technology - Software product evaluation*, ISO/IEC 14598-1, 1999.

International Organization for Standardization (ISO). Software engineering - Product quality, Part 1: Quality model. In *JTC 1/SC 7 - Software and systems engineering*, ISO/IEC 9126-1, 2001.

Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2nd edition, 2007. ISBN 9780387682815.

Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering Methodology*, 21(3):14:1–14:39, 2012.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Chirstina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

Chang Hwan Peter Kim, Christian Kästner, and Don Batory. On the modularity of feature interactions. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–34. ACM, 2008.

James C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.

Tomoji Kishi and Natsuko Noda. Formal verification and software product lines. *Communications of ACM*, 49(12):73–77, 2006.

Samuel Kounev and Christofer Dutz. QPME: A performance modeling tool based on queuing petri nets. *SIGMETRICS Performance Evaluation Review*, 36(4):46–51, 2009.

Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, 2010.

Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

Jaejoon Lee, Kyo Kang, and Sajoong Kim. A feature-based approach to product line production planning. In *Software Product Lines*, volume 3154 of *LNCS*, pages 137–140. Springer, 2004.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling Interactions in Feature-Oriented Designs. In *Proceedings of the International Conference on Feature Interactions (ICFI)*, pages 178–197. IOS Press, 2005.

Jia Liu, Don Batory, and Christian Lengauer. Feature-oriented refactoring of legacy applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM, 2006.

Roberto Lopez-Herrejon and Sven Apel. Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 422–437. Springer, 2007.

Rafael Lotufo, Steven She, Thorsten Berger, Andrezj Wasowski, and Krzysztof Czarnecki. Evolution of the Linux kernel variability model. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 136–150. Springer, 2010.

Kanti V. Mardia, John T. Kent, and John M. Bibby. *Multivariate Analysis (Probability and Mathematical Statistics)*. Academic Press, first edition, 1980. ISBN 0124712525.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, 2(4):308–320, 1976.

Jim A. Mccall, Paul K. Richards, and Gene F. Walters. Factors in software quality. Volume I. Concepts and definitions of software quality. Technical Report ADA049014, General Electric Co Sunnyvale California, November 1977.

Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276. ACM, 2009.

Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature interaction: The security threat from within software systems. *Progress in Informatics*, 5:75–89, 2008.

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM (CACM)*, 15(12):1053–1058, 1972.

Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. ISBN 3540243720.

Keith P. Pomakis and Joanne M. Atlee. Reachability analysis of feature interactions: A progress report. *SIGSOFT Software Engineering Notes*, 21:216–223, 1996.

Gilda Pour. Component-based software development approach: New opportunities and challenges. In *Proceedings of the International Conference on Technology of Object-Oriented Languages (TOOLS)*, pages 376–383. IEEE, 1998.

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

Christian Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and Systems Modeling*, 3(3):221–234, 2004.

Rick Rabiser, Deepack Dhungana, and Paul Grünbacher. Tool support for product derivation in large-scale product lines: A wizard-based approach. In *Proceedings of the International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 119–124. IEEE, 2007.

Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 131–140. ACM, 2011.

Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: A balanced energy-efficiency benchmark. In *Proceedings of the International Conference on Management of Data (COMAD)*, pages 365–376. ACM, 2007.

Suzanne Robertson and James Robertson. *Mastering the requirements process*. ACM Press, 1999. ISBN 0-201-36046-2.

Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering – An International Journal*, 18(2):163–197, 2011a.

Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM Press, 2011b.

Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional variability modeling. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–20. ACM Press, 2011c.

Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for RE for self-adaptive systems. In *Proceedings of the International Conference on Requirements Engineering Conference (RE)*, pages 95 –103, 2010.

Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the International Conference on Requirements Engineering (RE)*, pages 139–148. IEEE, 2006.

Sergio Segura. Automated analysis of feature models using atomic sets. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceeding of the International Conference on Software Engineering (ICSE)*, pages 461–470. ACM, 2011.

Norbert Siegmund, Martin Kuhlemann, Marko Rosenmüller, Christian Kästner, and Gunter Saake. Integrated product line model for semi-automated product derivation using non-functional properties. In *Proceedings of Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 25–31. ICB Research Report, University of Duisburg-Essen, 2008a.

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, and Gunter Saake. Measuring non-functional properties in software product lines for

product derivation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE, 2008b.

Norbert Siegmund, Mario Pukall, Michael Soffner, Veit Köppen, and G. Saake. Using software product lines for runtime interoperability. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 1–7. ACM, 2009a.

Norbert Siegmund, Marko Rosenmüller, Guido Moritz, Gunter Saake, and Dirk Timmermann. Towards robust data storage in wireless sensor networks. *IETE Technical Review*, 26(5):335–340, 2009b.

Norbert Siegmund, Martin Kuhlemann, Sven Apel, and Mario Pukall. Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 115–122. ICB Research Report, University Duisburg-Essen, 2010a.

Norbert Siegmund, Marko Rosenmüller, and Sven Apel. Automating energy optimization with features. In *Proceedings of the International Workshop on Feature-oriented Software Development (FOSD)*, pages 2–9. ACM, 2010b.

Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo Giarrusso, Sven Apel, and Sergiy Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 160–169. IEEE, 2011.

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012a.

Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo Giarrusso, Sven Apel, and Sergiy Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology (IST)*, 2012b.

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20: 487–517, 2012c.

Julio Sincero, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in software product lines. In *Proceedings of Software Product Line Conference (SPLC), Doctoral Symposium*, pages 167–173. IEEE, 2007.

Julio Sincero, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. Approaching non-functional properties of software product lines: Learning from products. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 147–155. IEEE, 2010.

Stanley S. Stevens. On the theory of scales of measurement. *Sciences*, 103(2684): 677–680, 1946.

Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One size fits all? part 2: Benchmarking studies. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 173–184, 2007.

Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109 –111, 2002.

Craig Taube-Schock, Robert J. Walker, and Ian H. Witten. Can we avoid high coupling? In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 204–228. Springer, 2011.

Rasha Tawhid and Dorina Petriu. Automatic derivation of a product performance model from a software product line model. In *SPLC*, pages 80–89. IEEE, 2011.

Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE, 2009.

Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE, 2011.

Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schäfer, and Gunter Saake. Analysis strategies for software product lines. Technical report, University of Magdeburg, Nb.: FIN-04-2012, 2012.

Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *International Symposium on Requirements Engineering (RE)*, pages 249–262. IEEE, 2001.

Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 129–140. IEEE, 2007.

Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting highly optimal architectural feature sets with filtered Cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

Karl Eugene Wiegers. *Software Requirements.* Microsoft Press, 2 edition, 2003. ISBN 0735618798.

Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the International Conference on Testing Communicating Systems: Tools and Techniques (TestCom)*, pages 59–74. Kluwer, B.V., 2000.

Ian H. Witten and Eibe Frank. *Data mining: Practical machine learning tools and techniques*. Elsevier, Morgan Kaufman, 2. edition, 2005. ISBN 0-12-088407-0.

Murray Woodside and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.

Sherif Yacoub. Performance analysis of component-based applications. In *Software Product Lines*, volume 2379 of *LNCS*, pages 1–5. Springer, 2002.

Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the International Symposium on Requirements Engineering (RE)*, pages 226–235. IEEE, 1997.

Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1087–1097. VLDB Endowment, 2004.