Master's Thesis

Assessing Performance Evolution Of Configurable Software Systems

Untersuchungen zur Performanz-Evolution von konfigurierbaren Software-Systemen

Author:

Stefan Mühlbauer

October 5, 2017

Advisors:

Prof. Dr.-Ing. Ina Schaefer
Technische Universität Braunschweig · Institute for Software Engineering and
Automotive Informatics

Prof. Dr.-Ing. Norbert Siegmund
Bauhaus University Weimar · Chair for Intelligent Software Systems

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Braunschweig, October 5, 2017   _____

Statement of Originality

This thesis has been performed independently with the support of my supervisors. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, October 5, 2017   _____

Contents

1. Introduction

Configurable Software Systems.   Modern software systems often need to be customized to satisfy user requirements. Configurable software, for instance, enables greater flexibility in supporting varying hardware platforms or tweaking system performance. To make software systems configurable and customizable, they exhibit a variety of *configuration options*, also called *features* (Apel et al., 2013). Configuration options range from fine-grained options that tune small functional- and non-functional properties to those that enable or disable entire parts of the software system. The selection of configuration options can be accommodated at different stages, either at *compile-* or *build-time* when the software is built or at *load-time* before the software is actually used. Compile-time variability usually governs what code sections get compiled in the program. Runtime-variability allows to configure the system during execution, which is needed, for example, for context-sensitive systems. For instance, compile-time variability can be realized by excluding code sections from compilation using preprocessor annotations (Hunsen et al., 2016) or by assembling the code sections to compile incrementally from predefined code modules depending on the feature selection (Schaefer et al., 2010). In contrast to that, load-time variability controls which code sections can be visited during execution. Configurations for load-time variability can be specified using configuration files, environment variables, or command-line arguments. Many software systems are configurable, examples range from small open-source command-line tools to mature ecosystems including Eclipse or even operating systems, such as the Linux kernel with more than $11,000$ options (Dietrich et al., 2012).

Configuration options for software systems are usually constrained (e.g., are mutually exclusive, imply or depend on other features) to a certain extent. In the worst case though, at which all options can be selected independently, the number of valid configurations grows exponentially with every feature added, and exceeds the number of atoms in the entire universe once we count 265 independent features. Hence, even for a small number of features, any naive approach for assessing emergent properties of configurable software systems exhaustively for each valid configuration is conceived infeasible. Despite this mathematical limitation, many feasible approaches to static analysis for configurable systems emerged. Those variability-aware approaches enable, for instance, type checking in the presence of variability by exploiting commonalities among different variants (Thüm et al., 2014).

To meet functional and non-functional requirements, users aim at finding the optimal configuration of a configurable software system. However, this task is non-trivial and has shown to be NP-hard (White et al., 2009). The main driver for the complexity are feature interactions. A *feature interaction* is an "emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved" (Apel et al., 2013)

and can make development and maintenance of a configurable system an error-prone task.

To illustrate feature interactions, consider the following example (Siegmund et al., 2015): A software system, say a file server, is used to store files in a database and provide access upon request. The system provides two features, encryption and compression. In isolation, both file en- or decryption and file (de-)compression demand an expectable fraction of memory and processor time. The performance behavior for the software system though may vary if both features are selected. For instance, if a file is encrypted and compressed (or vice versa), we can expect the operation to demand less resources since an compressed file is likely to be of smaller size than the decompressed original. This is a positive example for a feature interaction, where the performance behavior although being benefitting, is unexpected.

Performance Behavior. The term "performance" with respect to software and software systems is not precisely defined and differs from an end user's and developer's perspective. According to Molyneaux (2014), from a user's perspective "a well-performing application is one that lets the end user carry out a given task without any undue perceived delay or irritation". However, to accurately assess performance, from a practitioner's perspective, performance is outlined by measurements called *key performance indicators* (KPIs) which relate to non-functional requirements (Molyneaux, 2014). The set of KPIs includes availability of a software system, its response time, throughput, and resource utilization. Availability comprises the amount of time an application is available to the user. Response time describes the amount of time it takes to process a task. Throughput describes the program load or number of items passed to a process. Resource utilization describes the used quota of resources required for processing a task.

The performance behavior of a software system depends on the functionality offered, the respective implementation, program load, the underlying hardware system, environment variables, and the resulting execution. Since configuration options control what and how functionality is executed, we concentrate here on this source of performance. While feature interactions not necessarily cause the software system to break severely in all cases, its overall performance can become unfavorable for corner cases or specific configurations as the feature selection influences the execution. That is, the choice of features influences the performance of a software system, too.

Performance and Evolving Software. Actively maintained software systems evolve with every modification made, every version released, and patch provided. Modifications usually introduce new functionality to the system, but functionality might as well be divided into smaller modules to provide more fine-grained configuration options. When features are removed from the software system, the corresponding functionality might remain in the code base or options are merged (Apel et al., 2013).

There exists substantial work on understanding the evolution of configurable systems, for instance, with respect to software architecture (Zhang et al., 2013; Passos et al., 2015) or variability (Seidl et al., 2012; Peng et al., 2011; Passos et al., 2012). As software evolves, the code base which is subject to modifications and the overall architectural quality can degrade. Common symptoms of architectural degradation are code tangling and scattering (Zhang et al., 2013; Passos et al., 2015), which lead to less cohesive and stricter coupled code. The more the code base is constrained and interdependent, the more software can become "brittle" (Perry and Wolf, 1991), less flexible, harder to adapt, and

therefore harder to evolve. Evolution of software, especially with respect to variability, is essentially driven by and can be conceived as adapting a software system to changed requirements and contextual changes (Peng et al., 2011). That is, (potential) degradation of software quality as software evolves is often a phenomenon due to decisions trading quality assurance (QA) and maintenance tasks with meeting requirements and schedules (Guo et al., 2011). The metaphor of *technical debt* (Guo et al., 2011) which is commonly used to describe this trade-offs and corresponding costs, outlines the risk that postponed maintenance tasks pose to software evolution. Although every deferred maintenance or QA task may save some cost, it also could have unveiled software defects in the first place. Technical debt implies both interest, so to speak, the potential damage of a defect depending on its severity, as well as the probability of incurring interest. A defect can be severe, yet fixable with reasonable effort and cost. However, the aforementioned symptoms of architectural degradation and deferring maintenance render bug-fixing to become more and more expensive.

Besides the aspects of software evolution discussed above, the evolution of performance for software systems has gained more attention recently. In practice though, quality assurance with respect to performance is still conducted to an unsatisfactory extent, or accommodated too late in the development process, according to Molyneaux (2014). Thus, postponed maintenance and QA with respect to performance is likely to a driving factor for degradation of performance quality, or simply called *performance regression.*

While performance discipline has emerged as a discipline of or target in software testing, qualitative root-cause analysis, for the most part, is still conducted manually (Molyneaux, 2014). However, there exists work on automated root-cause analysis for performance bugs, such as measuring the execution time of unit tests, whereby an increased execution time indicates performance regression, and the corresponding unit test helps isolating the root-cause thereof (Heger et al., 2013; Nguyen et al., 2014b). In conclusion, we see that, to better assure good software performance, more knowledge about performance behavior needs to be available, ideally, earlier in the development process.

Performance Prediction. For configurable software systems, performance behavior can be more complex and dependent on the feature selection, as we have seen with the example for feature interactions above. Similarly, quality assurance for configurable software systems is far from exhaustively testing all possible configurations, but rather close to only testing a selection of configurations sampled with respect to certain constraints. Sampling strategies might stress feature interactions, such as pair-wise sampling (Siegmund et al., 2012). However, all samples are selected with the intention to learn as much as possible about the entire system from a small sample set of variants. So to speak, a sampling strategy is "optimal" if for a resulting sample set, the probability of missing an arbitrary (relevant) feature interaction, is minimal.

While performance testing is apparently useful, recently, a number of techniques to model and predict performance behavior for arbitrary configurations have emerged (Siegmund et al., 2012, 2015; Guo et al., 2013; Sarkar et al., 2015). The underlying optimization problem of performance-prediction models is to find an accurate estimator $\hat{f}$ for a function f describing a performance property depending on the feature selection. Performance properties can be estimated without performance measurements, for instance by inferring performance properties from software models (Woodside et al., 2007); measurement-based approaches for configurable systems address this optimization problem. The proposed

approaches include learning performance behavior with decision trees (Guo et al., 2013; Sarkar et al., 2015), learning a frequency-based representation of the target function (Zhang et al., 2015), or learning the influence of single features and all performance-relevant feature interactions (Siegmund et al., 2012, 2015). All approaches have shown promising error rates for several real-world applications and allow prediction of system performance for arbitrary configurations. To create performance-prediction models, all approaches demand samples of performance measurements for multiple configurations.

1.1 Problem Statement

The assessment of performance evolution requires a series of performance-prediction models describing performance behavior for a series of versions of a configurable software system. Assessing the performance behavior for a single version of a configurable software system entails a number of necessary and preliminary tasks. These tasks can even become more complicated once a series of versions is assessed, instead of a single one:

☐ *Feature Model Synthesis:* Not all configurable software systems do explicitly exhibit a variability model which is, however, required to derive all valid variants (Rabkin and Katz, 2011; Nadi et al., 2015). While substantial work exists on reverse engineering variability models from code (Rabkin and Katz, 2011; She et al., 2011; Zhou et al., 2015; Nadi et al., 2015) or non-code artifacts (Alves et al., 2008; Andersen et al., 2012; Bakar et al., 2015), many techniques still involve manual decisions (She et al., 2011) and domain knowledge (Nadi et al., 2015). Moreover, variability models evolve as part of the software (Peng et al., 2011), vary from version to version, and therefore, require repeated reverse engineering steps.

☐ *Configuration Translation:* The translation of a valid configuration to a configuration artifact such as a configuration file or a list of command-line arguments may differ from system to system. This step may be automated, but one still needs to detect how configurations are read by the software system one wants to study.

☐ *Automated Integration:* The sampe applies to the infrastructure to compile or build a software system since there exist many possible build tools such as makefiles or sbt. Again, the build process can be automated, but one needs to detect and specify the build mechanism used.

☐ *Version History Sampling:* To study performance evolution we need to specify which snapshots or versions of a software system are relevant to describe its performance behavior over time. While detecting releases and release candidates should be straightforward, one might, for instance, be interested in the performance evolution including snapshots between two releases. As not all snapshots though are likely to compile, classifying defect snapshots can still be tedious work.

☐ *Performance Assessment Setup:* The accurate assessment of performance evolution requires a suitable testing setup. The methodology required for assessing performance among others requires the selection of suitable performance metrics and corresponding benchmarks, means to record measurements, and repeat experiments easily as well as proper ways to interpret and compare results.

Goals and Thesis Structure. The goal of this thesis is to provide a theoretical and practical methodology to enable exhaustive performance measurements of configurable software systems and over their version history. That is, we contribute a guideline of and tool support for performance measurements of configurable and evolving software systems. Our research objectives and desired outcomes are

- a literature overview regarding software evolution, feature model synthesis, and performance assessment,

- a methodology to assess performance evolution with respect to the aforementioned challenges, and

- a practical tool for performance measurement for multiple revisions of configurable software systems.

We also propose an approach of automatically detecting promising versions of the configurable software systems for performance measurements and evaluate whether our assumptions hold.

The thesis is organized as follows. Chapter 2 provides the background to the relevant topics discussed in this thesis, including variability modeling, software evolution, feature model synthesis, performance assessment, statistical basics to summarize data records, and performance prediction models. In Chapter **??**, we propose our measurement methodology and discuss the methods used for our performance measurement tool as well as its limitations. In Chapter **??**, we evaluate several aspects of our tool with respect to practicality and discuss the results thereof. Finally, Chapter **??** concludes the thesis and gives an outlook on possible future work.

1.2 Performance Evolution Assessment

The methodology described in this thesis is organized with respect to three dimensions of configurable software systems: variability, version history, and performance. The schematic overview of the performance evolution assessment process shown in Figure 1.1 outlines the major aspects of each dimension.
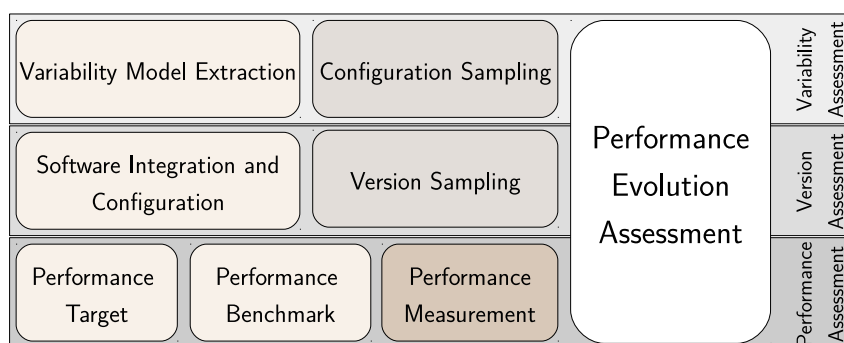
Fig. 1.1. Overview of the performance evolution assessment process

First, objectives related to the variability of a configurable software system are considered. This includes the extraction and comprehension of knowledge about the system's

variability. econd, we address objectives which emerge with evolving software, for instance whether a system compiles for a specific version, or whether a version is a promising candidate for detecting performance changes. Finally, we describe in detail the assessment of performance which comprises the selection of suitable performance benchmarks as well as a selection of appropriate statistical means to summarize, compare, and evaluate performance statistics in order to derive meaningful insights.

In addition to the three aforementioned dimensions in Figure 1.1, performance evolution assessment can also be conceived as consisting of three different categories of tasks, as the different colorizations might suggest. First, for a configurable software system, knowledge about it needs to be gathered, and clear performance goals need to be specified. Second, a sample set of configurations along with versions need to be selected. Since covering all variants and versions is not feasible, a sampling strategy needs to be chosen which is likely to uncover performance changes. Finally, the practical performance measurements are conducted and evaluated as mentioned above.

2. Background

This chapter presents necessary background to the individual topics used in this thesis. In Secion 2.1, we recapitulate basic concepts, notations and terminology regarding variability modeling, and Sec. 3.1.2 presents methodologies to recover variability models from source code. In Sec. 2.2, we outline different aspects of software evolution. Sec. 2.3 discusses the assessment and testing of performance and Sec. 5.3 reviews statistical basics to describe and compare performance measurements. Finally, Sec. 2.4 recalls methodology to predict performance behavior for configurable systems.

2.1 Variability Modeling

The design and development of configurable software systems is conceptually divided into *problem space* and *solution space* (Czarnecki and Eisenecker, 2000). The problem space comprises the abstract design of features that are contained in the software system as well as constraints defined among features, such as dependencies or mutual exclusion. The solution space describes the technical realization of features and the functionality described by and associated with features, e.g., implementation and build mechanisms. That is, features cross both spaces since they are mapped to corresponding code artifacts.

A common way to express features and constraints in the problem space is to define a *variability model*, or *feature model*, which subsumes all valid configurations (Kang et al., 1990; Thüm et al., 2009; Apel et al., 2013). There are different and equivalent syntactical approaches to define feature models, for instance, a propositional formula F over the set of features of the configurable software systems (Batory, 2005). In this case, a configuration is valid with respect to the feature model if and only if F holds for all selected features being true and all unselected features being false, respectively. However, a more practical and more commonly used way to express feature models are graphical tree-like *feature diagrams* (Apel et al., 2013). In a feature diagram, features are ordered hierarchically, starting with a root feature and subsequent child features. By definition, the selection of a child feature requires the parent feature to be selected as well. Child features can either be labeled as *optional* features or *mandatory* features; the latter ones need to be selected in every configuration. Moreover, feature diagrams provide a syntax for two different types of feature groups, *or-groups* and *alternative-groups*. For an or-group, at least one of the group's features needs to be selected for a valid configuration, whereas for an alternative-group exactly one out of the group's mutually exclusive features must be selected. In addition to the feature hierarchy, constraints, which cannot be expressed by the tree-like structure, are referred to as *cross-tree constraints*. Cross-tree constraints, depending on the notation, are depicted by arrows between two features or simply added to the feature diagram as a propositional formula. For two features f_1 and f_2, a cross-tree constraint

means that for feature f_1 to be selected, either the selection of f_2 is required/implied or excluded.

An introductory example for the syntax and semantics of feature diagrams is provided in Figure 2.1. In this example, an imaginary vehicle propulsion can be configured with eight valid configurations. The vehicle requires an engine, and therefore feature Engine is mandatory. At least one out of the three features Hybrid, Piston and Electric needs to be selected. For a piston engine, we can select either the feature Diesel or Petrol. A petrol engine requires additional ignition sparks in contrast to a Diesel engine. For an electric engine, we require a battery, hence, the feature Battery is mandatory. In addition, the feature model specifies two cross-tree constraints: First, the feature Tank is optional, yet once a piston engine is selected, we require a tank. Second, if we want to use the Hybrid functionality (e.g., use both electric and piston engine simultaneously), we require to have both a piston and an electric engine.
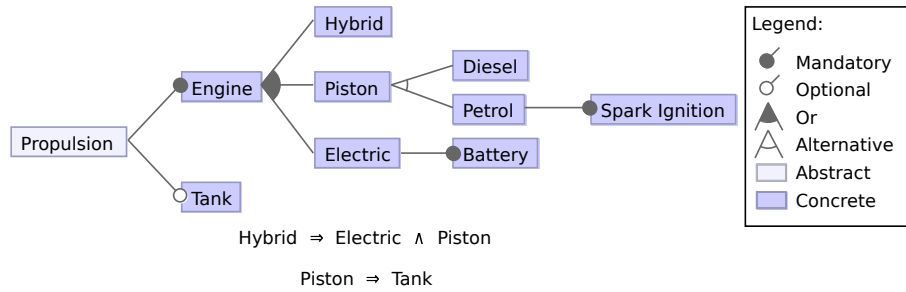
Fig. 2.1. Feature diagram for a feature model with eight valid configurations; two cross-tree constraints are specified as propositional formulas over features

2.2 Software Evolution

The first notion of the software development process is usually developer-centered and merely focuses on software being designed, implemented, tested, and eventually being released and deployed. Maintainability is a generally recognized software quality property to look after, and maintenance is, of course, essential to every successful software system Liggesmeyer (2009). Nonetheless, less attention is given to the ability to adapt a software system to changing requirements (evolvability) rather than maintaining it to keep functionality working (Parnas, 1994). Software evolution and evolvability, such as software itself are manifold. Software evolves in many ways ranging from maintenance (refactoring, bug-fixes, and patches) to adapting to changed requirements (adding, removing, reorganizing functionality, and variability). Modern software systems not only often ship with a variety of configuration options to select from, they also employ routines to be build and sometimes even make use of or are part of platforms, such as apps or plugins. That is, software evolution affects all aforementioned aspects, maintainability as well as evolvability can degrade as software evolves.

2.2.1 Software Erosion

The negative symptoms of software evolution which are referred to as "architectural erosion" (Breivold et al., 2012) have been addressed by many researchers. Most of existing

research so far though focuses on evolution with regard to software architecture (Breivold et al., 2012). The main driving factors leading to symptoms of decay identified by Perry and Wolf (1991) are architectural erosion and architectural drift. While architectural drift subsumes developers' insensitivity when not following a systems architecture or respective guidelines while making changes, architectural erosion subsumes ignoring and violating the existing software architecture. Parnas (1994) argues that as software evolves, software is maintained and evolved by developers who are not necessarily familiar with the initial architectural design. Therefore, knowledge about the architecture can become unavailable as software evolves. Although the unfavorable effects of software evolution do not break a system necessarily and imminently, the software becomes "brittle" (Perry and Wolf, 1991) as maintainability as well as evolvability degrade. Concrete symptoms of software erosion on the implementation level have been documented.

Zhang et al. (2013) have studied erosion symptoms for a large-scale industrial software product line with compile-time variability using preprocessor directives. The authors identify variability-related directives and clusters of those that tend to become more complex as the software evolves. The negative effects, or symptoms of software erosion are described as, but not limited to *code replication* and inter-dependencies between code elements, such as *scattering* and *tangling.* Code scattering describes the phenomenon of code belonging to a certain feature being scattered across multiple units of implementation, such as modules, whereas code tangling means that code from different and potentially unrelated features in entangled within a single module.

Passos et al. (2015) have studied the extent of usage of scattering for device-drivers in the Linux kernel. Despite scattering being quite prevalent, their findings suggest that the kernel architecture is robust enough to have evolved successfully. Nonetheless, platform drivers in the Linux kernel seem more likely to be scattered than non-platform drivers. They conclude that this is a trade-off between maintainability and performance: a more generalized and abstract implementation for platform-drivers in this case could possibly avoid scattering, yet refactorings in this manner did not seem to be necessary or worth the effort yet.

2.2.2 Variability Evolution

Apart from architecture evolution, the variability offered by software systems evolves as well. For configurable software systems, evolution steps will not only affect artifacts in the solution space, yet also be visible in changes in the respective variability models in the problem space. Although the variability aspect of software evolution has not gathered as much attention as architecture in the past, more and more research has emerged recently to address and understand variability evolution.

Peng et al. (2011) proposed a classification of variability evolution patterns that conceives evolution as adaption to changing (non-)functional requirements and contexts. For a context in that sense, two categories exist. A driving context determines, whether a variability model and respective variants can meet functional requirements in the first place. A supporting context by definition determines how non-functional properties are strengthened or weakened. Any changed requirement is likely to change the contexts for a software systems variability model and, therefore, will make adaptations of the variability model necessary. Within their classification method, Peng et al. (2011) identify major causes for variability evolution, comprising (a) new driving contexts emerging, (b) weak-

ened supporting contexts (for instance, due to new non-functional requirements), and (c) unfavorable trade-offs for non-functional properties.

To understand single evolutionary steps, several catalogs of variability evolution patterns have been proposed. Peng et al. (2011) present three patterns, where either a new feature is added, a mandatory feature becomes optional, or a mandatory/optional feature is split into alternative features. Seidl et al. (2012) suggest a catalog of patterns for co-evolution of variability models and feature mappings that additionally introduces code clones, splitting a feature into more fine-grained sub-features, and feature removal as evolution patterns. In addition, Passos et al. (2012) have studied variability evolution in the Linux kernel and present a catalog of patterns where features are removed from the variability model, but remain a part of the implementation. Their catalog, among others, includes feature merges, either implicit (optional feature merged with its parent) or explicit.

The classification proposed by Peng et al. (2011) is a general and formalized approach that, as well as Seidl et al. (2012) and Passos et al. (2012), describes elementary evolution patterns which can be composed to more complex patterns. Nonetheless, no comprehensive catalog of variability evolution patterns so far has been proposed.

2.3 Assessing Performance

While the last three sections covered software evolution and variability modeling, we now step forward to the topic of software performance. This section will outline the term performance with respect to software systems as well as to possible measurements. We provide a brief look at the general performance testing setup and the required prerequisites, including suitable benchmarks. Finally, we provide the statistical background to summarize, interpret and compare performance measurements accurately.

2.3.1 What is Performance?

The performance of software systems is, like software quality, primarily a matter of perspective. While an end user might consider practical aspects to be more important, from a developer's perspective, performance relates to and is best described by non-functional properties (Liggesmeyer, 2009; Molyneaux, 2014). While functional properties subsume what exactly a software system does, non-functional properties describe how (good or bad) a software system is at providing the functionality offered (Liggesmeyer, 2009). The notion of good and bad in this sense corresponds to non-functional requirements (NFR), that is, software with "good" performance behavior does not violate its NFRs. The categories of NFRs that shape performance behavior are manifold. According to Molyneaux (2014), the categories or *key performance indicators* (KPIs) include

□ availability

□ response time,

□ throughput,

□ resource utilization, and in a broader scope also

☐ capacity.

Time-related KPIs are availability and response time, whereby availability describes the time or time ratio that the software is available to the end user, and response time subsumes the time it takes to finish a request or operation. Throughput as a category subsumes the program behavior with respect to program load, such as hits per second for a Web application or amount of data processed per second. Resource utilization describes the extent to which a software system uses the physical resources (CPU time, memory, and disk, or cache space) of the host machine. Finally, from a Web-centered perspective, capacity describes measurements with respect to servers and networks, such as network utilization (traffic, transmission rate) and server utilization, such as resource limitations per application on a host server (Molyneaux, 2014).

Consequently, the assessment of performance requires a context or testing target that corresponds to the assessed system under test (SUT). For instance, for a simple command-line compression tool, suitable KPIs are response time and throughput, whereas performance for an online shop Web application is better outlined by availability and capacity.

2.3.2 Performance Testing

The first step in performance testing, prior to defining relevant KPIs and metrics, is to specify a system operation or *use case* (Woodside et al., 2007) to assess performance for. A typical use case includes a well defined task or workload to process, expected behavior, outcome, and performance requirements as previously discussed. For the SUT, however, we require a version that does compile or, in case it is interpreted, is syntactically correct (Molyneaux, 2014). With regard to performance assessments as part of the development process, a code freeze should be obtained since measurement results are likely to become meaningless for later versions. In addition to that, the machine or setup used for performance measurement should ideally be as close to the production environment as possible, but at least be documented to compare different runs (Molyneaux, 2014).

Finally, one or more benchmarks need to be selected to simulate the program load for the respective use case. A benchmark, all in all, needs to be representative, i.e., should relate to the use case or requirement one wants to validate. While benchmarks for file compression usually include multiple different types of media data (text, sound, pictures) such as the Canterbury corpus[1], Web applications can be exposed to handling with a number of simulated users at the same time (Molyneaux, 2014). Benchmarks have often been standardized within research and engineering communities to provide comparable performance measurements. A popular example is the Software Performance Evaluation Corporation (SPEC), a consortium providing a variety of benchmarks such as the CPU2000[2] processor benchmark consisting of programs with both floating point and integer operations. Moreover, benchmarks, in a sense of repeatable program load, can be obtained from load generation software such as ApacheBench[3] for the Apache Web server or simply by measuring performance for test cases (Heger et al., 2013; Nguyen et al., 2014b).

Performance testing heavily relies on tool support, especially for repeating test cases and recording measurements. Since the tool solutions for performance testing vary from

---

[1]The Canterbury corpus can be found here, `http://corpus.canterbury.ac.nz/`.

[2]The benchmark description can be found at `https://www.spec.org/cpu2000/`.

[3]Manual page of the Apache tool `ab`, `http://httpd.apache.org/docs/2.4/en/programs/ab.html`.

domain, scale, and purpose, we will outline only the general tool architecture. Molyneaux (2014) describes four primary components for a performance testing setup: a scripting module, a test management module, a load injector, and an analysis module. A scripting module handles the generation or repetition of use cases which, for instance, can be recorded prior to the test for Web applications. A test management module creates and executes a test session, whose program load is generated by one or more load injectors. A load injector can provide a benchmark by generating items to process, or can simulate a number of clients for a server-side application. An analysis module, finally, collects and summarizes data related to the performance testing target. We present more on summarizing and comparing recorded results in the Section **??**.

2.4 Performance Prediction Models

In the previous section, we referred to performance, or in detail, the KPIs, as possible testing targets, which we validate against non-functional requirements. However, in a broader sense, software performance has become an aspect of software engineering referred to as *software performance engineering* (SPE) (Woodside et al., 2007). A lot of effort has been spent to study and describe performance behavior as well as to improve performance quality. Besides the analysis of concerns or requirements with respect to performance, SPE comprises performance testing as well as performance prediction (Woodside et al., 2007). Performance prediction aims at modeling and estimating performance behavior for different use cases or configurations. The first approaches to performance prediction models describe the underlying software system component or operation from which performance estimations are then deduced. These so-called *model-based prediction models* enable a performance estimation early in the development process since no actual performance measurement is required (Woodside et al., 2007). In contrast to model-based approaches, measurement-based approaches have emerged. These *measurement-based prediction models* are based on a sample of performance measures which are used to learn a software system's performance behavior (Woodside et al., 2007). More recently, measurement-based prediction models that emphasize variability have been proposed, of which we will discuss three approaches in the remainder of this section.

Learning Performance In essence, learning and predicting performance behavior for a configurable system means nothing different than finding an approximation $\hat{f}$ for a function $f : C \to \mathbb{R}$ where C is the set of configurations and $f(c) \in \mathbb{R}$ with $c \in C$ is the corresponding performance measurement. The accuracy of the approximation $\hat{f}$ describes how the estimated performance $\hat{f}(c)$ deviates from the actually measured performance $f(c)$. Different approaches to construct such an approximation, referred to as a *performance prediction model*, emerged recently. All approaches utilize two samples of configurations and corresponding performance measurements to build and validate the model. A training sample is used to learn the approximation from, whereas a testing sample is used to assess the prediction error rate of the previously learned approximation.

A straightforward methodology to learn performance behavior was proposed by (Guo et al., 2013). The authors utilize classification-and-regression-trees (CART), akin to decision trees, to derive a top-down classification hierarchy for a given sample. The approach supports progressive (and random) sampling, i.e., the performance model is constructed

several times while the size of the training sample is successively increased. The CART is constructed by recursively partitioning the sample into smaller segments which best describe a class. It is worth mentioning that the estimation using CART is not limited to binary configuration options, but supports numeric features as well. Moreover, the approach by (Guo et al., 2013) does not produce any further computation overhead besides the measurement effort and the construction of a CART.

A different approach to modeling performance behavior was proposed by Zhang et al. (2015). The authors propose an approach to construct performance prediction models based on a Fourier description of the performance-describing function f, which maps each configuration to a corresponding performance measurement. The principal idea is to approximate a Fourier series approximating the function f, and learning all coefficients of the Forier series terms.

The number of terms is exponential in the number of configuration options. The main characteristic of this approach is that, prior to learning a performance prediction model, a desired level of prediction accuracy can be specified. That is, the approach automatically chooses the sample size required to learn the prediction model accordingly.

A third approach to model and predict performance behavior, *SPLConqueror*, was proposed by Siegmund et al. (2012). The authors describe performance behavior for a configuration as the accumulated influence of features, from which the configuration is composed, and respective feature interactions. To estimate the influence of single features and feature interactions, the authors propose a number of sampling strategies. Single features are assessed by comparing the performance of two different configurations per feature: For a feature f, two valid configurations are compared, whereby both configurations have the minimal number of features selected that are not excluded by the selection of f. While for one configuration f is selected, it is deselected for the other one. The difference between the performance measures for both configurations is the estimate $\Delta_{min(f)}$ for the performance influence of feature f. Feature interactions that are performance-relevant are detected automatically in a similar manner. The main idea is, that a feature f is more likely to interact with other features, the more features are selected along with f. Therefore, if f interacts, the influence of feature f, $\Delta_{min(f)}$ differs from the influence of f, when estimated using configurations where the maximal number of features selectable together with f are used. Based on the set of interacting features, three heuristics are employed to detect feature interactions. Simple feature interactions are detected using pair-wise sampling (Siegmund et al., 2012; Apel et al., 2013), whereas higher-order feature interactions are often composed from simpler ones, or centered around a small subset of hot-spot features (Siegmund et al., 2012). Finally, for an arbitrary configuration, the performance can be predicted by the sum of influence estimations per feature and feature interaction.

The idea of predicting performance by estimating the influence of feature and feature interactions was further developed by Siegmund et al. (2015) by proposing performance-influence models for both binary and numeric configuration options. A performance influence, similar to *SPLConqueror*, predicts performance by computing the sum of previously learned influence estimates. The novelty of this approach lies in the way the terms describing the influence of feature and feature interactions are learned and not derived from a sampling strategy. Instead of a single constant performance influence estimate, each term can contain of a number of functions, such as linear, quadratic, or logarithm functions, or compositions thereof. This does not only allow the developer to incorporate domain

knowledge by preselecting functions. Moreover, but also by introducing combinations of non-linear functions and learning the coefficients using linear regressions, it enables learning a non-linear function. The algorithm commences with the selection of terms and successively extends and reduces the term selection to increase the prediction accuracy. The outcome, similar to Siegmund et al. (2012), is a linear function whose terms represent the estimated influence of features and feature interactions on performance.

All aforementioned approaches rely on performance measurements and, to some extent, also on a sampling strategy. While the approaches by Siegmund et al. (2012, 2015) essentially describe sampling strategies, the approach proposed by Zhang et al. (2015) is able to work with random samples, and the approach of Guo et al. (2013) allows a smaller sample to be extended progressively (Guo et al., 2013). As performance measurements entails an expensive and time-consuming process, Sarkar et al. (2015) have investigated different sampling strategies in this domain. The authors compare progressive sampling, where sample sizes are increased successively until the learned prediction model performs accurately enough, and projective sampling, where the optimal sample size is estimated based on the expected learning curve, with respect to cost and prediction accuracy. They advocate the use of projective sampling over progressive sampling. In addition, the authors propose and evaluate an own sampling heuristic to select an initial sample for progressive sampling. This sampling heuristic is based on feature frequencies and outperforms t-wise sampling, such as pair-wise sampling (Sarkar et al., 2015).

3. Variability Assessment

To assess performance evolution for configurable software systems, it is inevitable to assess and understand variability of such systems with respect to various aspects: *First*, to actually assess single variants, knowledge about the variability model is required to configure the software systems accordingly. *Second*, obtaining knowledge about feature usage and implementation can provide meaningful insights. For instance, knowing that most variable code is dependent on larger numbers of features might be useful information when selecting a sampling strategy. Similarly, knowing which feature combinations are frequently involved in conditioning program functionality and behavior can help understand configuration-related defects. *Last*, since the number of configurations for most configurable software systems is infeasible, variability assessment faces performance- and scalability-related problems. That is, assessments are usually conducted constrained by limited resources.

This chapter describes the first preliminary steps in our methodology and addresses the following questions. Section 3.1 reviews the synthesis of variability models as well as useful analyses of systems' variability; section 3.2 describes means to generate configurations from variability models, and section 3.3 dicusses different sampling strategies.

3.1 Untangling Variability

The idea of designing and developing configurable software systems is driven by the separation of different concerns, expressed as features of a software system. A configurable software system is either assembled at compile-time with respect to its configuration, or behaves according to a configuration read at load-time. While for both flavors of variability there exist various ways of realization, research has proposed a variety of variant-generating implementation techniques for compile-time variability, ranging from simple preprocessor directives to features as separate modules (Kästner et al., 2009). A complete survey of means to implement configurable software systems would likely exceed the scope of this section, yet we intend to discuss in this section which information regarding variability is required or useful to our methodology, and how it can be obtained.

3.1.1 Family-based Analysis

As the number of variants for configurable software systems is usually infeasible, naive analyses of configurable systems are not trivial. Any static analysis can only assess one variant at a time, as well as dynamic analyses, which can only follow one execution path. In contrast to that, recently, extended analysis techniques, which are aware of variability of the systems studied, have emerged (Thüm et al., 2014). In particular, these *family-based*

analyses avoid redundant computation, such as visiting a code section multiple times, and exploit artifacts shared by multiple variants (Thüm et al., 2014). Besides more efficient analysis, family-based methods incorporate knowledge about valid feature combinations (Thüm et al., 2014) and, therefore, connect analysis results with a context, such as feature combinations, for which the findings hold. Family-based methods have been widely used across various domains and can provide useful information when assessing configurable software systems.

Variability-aware Parsing. Kästner et al. (2011) have proposed a framework, *Type-Chef*, to enable the construction of variability-aware parsers. A variability-aware parser, like ordinary parsers, systematically explores a program to return an abstract representation of the parsed program. This parse tree, or abstract syntax tree is the basis for compilers to translate a program, or for further static analyses including type checking. For a code base with variability expressed by preprocessor directives, which are evaluated prior to compilation, a variability-aware parser, however, is able to derive a parse tree considering all variants in a single run. A parse tree usually consists of nodes representing syntactical features of the parsed program. The parse tree returned by a variability-aware parser, moreover, comprehends variable segments of a program and will include them with respect to their presence conditions. For instance, a class may contain a function sort, for which two different implementations exist. While there might be numerous variants, the parse tree of the class will contain a node with two children, one for each implementation; higher numbers of alternative implementations are expresses by nesting further nodes. In that way, variable and invariant program segments can be separated.

While the approach of Kästner et al. (2011) handles undisciplined usage of preprocessor directives, such as splitting function parameter lists, variable types, or expressions, Medeiros et al. (2017) have proposed an approach to avoid and conservatively refactor those cases. The authors propose a catalog of refactoring templates, which describe transformations from undisciplined usage of preprocessor annotations to disciplined ones. With respect to variability-aware parsing, disciplined usage is conceived as using preprocessor annotations only to segment statements, but not to segment a single syntactical unit, such as expressions (Medeiros et al., 2017).

Staged Variability. Besides variability-aware parsing, Nguyen et al. (2014a) have applied symbolic execution (King, 1976; Darringer and King, 1978) to unwind variability for PHP Web applications. Web applications are staged, i.e., even though it can be configured at load-time, the applications is as well variable with respect to input received at run-time. For instance, consider WordPress, a popular content management system (CMS) implemented in PHP, can be extended with a number of plug-ins. However, the content of a website presented to the user also depends on information retrieved from a database, and user input. Consequently, a dynamic PHP Web application is staged in a sense that it generates configurable HTML templates which are rendered at run-time. The authors utilize a symbolic execution to explore all possible execution paths. Each user input or database query is considered a symbolic value which is propagated through each script. By keeping track of the (partially symbolic) HTML output and organizing it in a DOM-like structure, their approach approximates the HTML output, which subsequently can be tested, for instance for validity (Nguyen et al., 2011).

Similarly, Lillack et al. (2014) have applied taint-analysis to configurable software

systems to track the influence of configuration options read at load-time. Their static analysis approach taints every value resulting from reading a configuration parameter as well as every value resulting from a computation that involves previously tainted values. That way, lines of code that are possibly depending on configuration options are detected.

Build System Variability. Build System Variability. Apart from configuring software systems using preprocessor annotations, the assembling of a configurable software system can as well be orchestrated by its underlying build system. While preprocessor annotations virtually separate code fragments of different features, for instance, build systems can physically exclude files from compilation. This variability enabled by build systems, in particular of Makefiles has been subject to a couple of analysis approaches. Tamrawi et al. (2012) have proposed Symake, a symbolic execution engine to evaluate Makefiles. On top of Symake, Zhou et al. (2015) use symbolic execution to analyze Makefiles and derive file presence conditions, stating under which feature selection a file is included or excluded from compilation. The work of Al-Kofahi et al. (2016) addresses a more advanced build system, GNU Automake. Automake describes a staged build process, where a Makefile can be specified on a higher level, and is subsequently compiled to an actual Makefile. The authors' aim to provide a variability-aware representation of all possible Makefiles to enable further analyses of the build process.

3.1.2 Reverse Engineering Variability Models

A variability model as an abstraction of functionality of a software system is required, or at least of great interest, in many contexts. *First*, not every configurable system provides an explicit representation of its variability model. The reasons for inexplicit or absent configuration specification are manifold. They can range from poor or inconsistent documentation (Rabkin and Katz, 2011) to overly complex configurability (Xu et al., 2015), or configuration constraints originated in different layers of a software system, such as build constraints or compiler constraints (Nadi et al., 2015). *Second*, variability models have emerged to be a useful means in domain analysis prior to developing a software system. As feature diagrams group and organize features (representing functionality), synthesizing a variability model has shown to be applicable to extract features and constraints from functional requirements. In addition, by comparison of product specifications for an existing market domain, variability models can provide a detailed feature summary (Alves et al., 2008; Bakar et al., 2015).

For this thesis, we focus on the first aspect of synthesizing variability models as our work addresses the assessment of already existing configurable software systems. Nonetheless, many techniques employed in the aforementioned second aspect consider similar problems, yet rely on natural language artifacts rather than code artifacts (Alves et al., 2008; Bakar et al., 2015). The following section recalls work on extracting configuration options and constraints from source code as well as the organization of constraints into feature hierarchy and groups. The further assessment of configurable systems requires a well-defined and sound variability model.

The first objective in recovering a variability model from a configurable system is to determine the available configuration options to select from. In addition, for further configuration, the type of each configuration option (e.g., boolean, numeric, or string) and the respective domain of valid values needs to be specified.

Feature Extraction. Rabkin and Katz (2011) proposed a static, yet heuristic approach to extract configuration options along with respective types and domains. Their approach exploits the usage of configuration APIs and works in two stages. It commences with extracting all code sections where configuration options are parsed. Next, configuration names can be recovered as they are either already specified at compile-time or can be reconstructed using string analysis yielding respective regular expressions. Moreover, the authors employ a number of heuristics to infer the type of parsed configurations as well as respective domains. First, the return type of the parsing method is likely to indicate the type of the configuration option read. Second, if a string is read initially, the library method it is passed to can reveal valuable information about the actual type. For instance, a method *parseInteger* is likely to parse an integer value. Third, whenever a parsed configuration option is compared against a constant, expression, or value of an enum class, these might indicate valid values or at least corner cases of the configuration option domain. The extraction method by Rabkin and Katz (2011) is precise, but limited, for instance, when an option leaves the scope of the source code. Nonetheless, for the systems studied, the authors were able to recover configuration options that were not documented, only used for debugging or even not used at all.

Constraint Extraction. A more comprehensive investigation of configuration constraints and their origin is provided by Nadi et al. (2014, 2015). They use variability-aware parsing to infer constraints by evaluating make files and analyzing preprocessor directives. Inferred constraints result from violations of two assumed rules, where (a) every valid configuration must not contain build-time errors and (b) every valid configuration should result in a lexically different program. While the first rule aims at inferring constraints that prevent build-time errors, the second one is intended to detect features without any effect, at least as part of some configurations. Their analysis one the one hand emerged to be accurate in recovering constraints with 93 % for constraints inferred by the first rule and 77 % for second one respectively. On the other hand, their approach recovered only 28 % of all constraints present in the software system. Further qualitative investigation, including developer interviews, lead to the conclusion that most of existing constraints stem from domain knowledge (Nadi et al., 2015).

Feature Hierarchy Recovery Besides recovering configuration options and respective constraints, to reverse engineer a feature model, one further step is required. The recovered knowledge needs to be organized in a tree-like hierarchy with feature groups specified and CTCs explicitly stated to derive a valid feature diagram (Kang et al., 1990). While several approaches for recovering the feature model hierarchy have been proposed, we are primarily interested in finding a hierarchy for knowledge obtained from source code. Other scenarios, as already stated in the opener of this section, are based on product descriptions or sets of valid configurations (Aleti et al., 2013; Bakar et al., 2015). In the remainder of this subsection, we will focus on organizing features and constraints extracted from source code. For further reading, Andersen et al. (2012) present algorithms for structuring feature diagrams for three different scenarios including the ones previously mentioned.

Given an extracted set of features along with corresponding descriptions and recovered constraints among the features, She et al. (2011) propose a semi-automated and interactive approach to synthesize a feature hierarchy. Their approach comprises three tasks.

First, an overall feature hierarchy based on feature implications is specified. Second, potential feature groups are detected and manually selected. Finally, the feature diagram is extended with remaining CTCs. A detailed description of the algorithm is presented below:

1. The algorithm commences with finding a single parent for each feature and, thus, specifying a tree-like feature hierarchy. Based on the given constraints, a directed acyclic graph (DAG) representing implication relationships among features, a so-called *implication graph*, is constructed. Every vertex in the implication graph represents a feature and edges are inserted for each pair of features (u, v), where $u \implies v$ holds with respect to the given constraints.

   In addition to the implication graph, the algorithm for each feature computes two rankings of features that are likely to be the respective parent feature. The two rankings both employ the feature descriptions. Feature descriptions are compared for similarity using a similarity metric. For two features p and s, the similarity is defined as the weighted sum of the inverse document frequencies $idf(w)$ for the words that both descriptions of features p and s share. The idf-ranking for a word w is the logarithm of the number of features divided by the number of features whose description contains w. Each idf value is weighted with the frequency of w in the description of feature p.

   The first ranking, called Ranked-Implied-Features (RIF), for each feature f ranks all features by their similarity to f in an descending order, but prioritizes those features that are implied according to the previously computed implication graph. The second ranking, called Ranked-All-Features (RAF) is similar to RIF, yet less strict since implied features are not prioritized. Given these rankings, a user for each feature selects a suitable parent feature from the RIF or RAF ranking. The idea behind providing two separate rankings, according to She et al. (2011) is that the given extracted constraints can be incomplete and, thus, not all relevant implications are contained in the implication graph.

2. After the feature hierarchy is specified, another auxiliary graph, a mutex graph, similar to the implication graph, is constructed. The *mutex graph* is an undirected graph with features as vertices and edges between two features u and v, if $u \implies \neg v$ and $v \implies \neg u$ hold with respect to the given constraints. That is, all adjacent features are mutually exclusive. Based on this mutex graph, all maximal cliques (subsets of vertices that all are connected with each other) among the vertices with the same parent are computed. All features within such a clique are mutually exclusive and share the same parent and represent mutex- or alternative-groups. She et al. (2011) introduce an additional constraint to extract xor-groups that require one of the groups' features to be selected if the parent is selected. This distinction is in line with the initial description of feature diagrams by Kang et al. (1990), but not all descriptions of feature diagrams follow this distinction between mutex- and xor-groups and just use the term alternative-group discussed in Sec. 2.1.

3. CTCs for the feature diagram are extracted from the given configuration constraints. Since CTCs are constraints that could not be represented by the feature hierarchy (implication) or alternative-groups (exclusion), the derivation of CTCs follows this

idea. The set of cross-tree implications is derived by removing all edges that are part of the feature hierarchy from the initially constructed implication graph. The set of cross-tree exclusions is derived similarly from the mutex graph by removing all edges among vertices of all mutex-groups. To make the feature model sound, the given configuration constraints, reduced to those clauses that are not already entailed by the diagram, can be added as an additional CTC formula to the feature diagram (She et al., 2011).

The approach by She et al. (2011) provides a practical algorithm to synthesize a feature diagram, yet has some limitations we need to consider. First, the approach is not able to detect or-groups as defined in Sec. 2.1. Second, the approach does introduce a root feature. Finally, the approach does not distinguish between mandatory and optional features. Implicitly, all features that do not have a parent feature are optional and all features that have a parent feature are by default mandatory. She et al. (2011) evaluated the algorithm with both complete and incomplete variability knowledge (feature names, descriptions and constraints). While the algorithm has shown to be practical, detecting features whose parent was the root-feature was difficult since, due to the transitive property of implication, it is implied by each feature of the feature model.

3.1.3   Practical Remarks

□ Scenarios based on two factors:

    − documentation of configurability

    − visibility of variability

3.2   Configuration Generation

Apart from having knowledge about the variability of a configurable software system, either in the form of a set of constraints among features, or a feature diagram, we also need to derive all configurations, but at least meaningful sample sets thereof, from the variability model. Variability models can be expressed in various forms, such as propositional formulas or context-free grammars (CFG) (Batory, 2005), as well as constraint satisfaction problems (CSP) (Benavides et al., 2005a,b). Accordingly, the all configurations represent solutions to propositional formulas or CSPs, and valid words for CFGs respectively. That is, the generation of all configurations with respect to the variability model is equivalent to finding a solution or word set for the aforementioned representations of a variability model. In the following, we look into how variability models can be encoded as a CSP and describe in detail the configuration generation using CFGs.

3.2.1   Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) in the context of variability modeling describes a set of options ranging over finite domain as well as a set of constraints which restrict which values variables can take simultaneously (Benavides et al., 2005a). Fo a binary option b, the respective domain $dom(b)$ simply is $\{0,1\}$. For a numeric option, the respective domain $dom(n)$ is a finite set of legal values with a minimum and a maximum value, say

$\{v_{min}, v_1, v_2, \ldots, v_{max}\}$. A solution $s : O \rightarrow dom(o_1) \times dom(o_2) \times \ldots \times dom(o_{|O|})$ to a CSP is an assignment of options $o_i \in O, i \in \mathbb{N}$ to values of their respective domain, such that all constraints are satisfied simultaneously (Benavides et al., 2005a).

A solution to a CSP is found by systematically checking for different selections of values whether all constraints are satisfied. There exists a large number of ready-to-use SAT and CSP solvers, yet we are not covering CSP solution here since it is beyond the scope of your thesis. For further reading, (Benavides et al., 2007) present a tool with extensive analysis support for various different presentations of variability models.

To encode a variability model as a CSP, Benavides et al. (2005a) describe the following transformation rules:

- For a parent feature f and a child feature f', a mandatory relationship is expressed as $f \Leftrightarrow f'$, and an optional relationship is expressed as $f' \implies f$.

- For a parent feature f and child features f_i, where $i = 1, 2, \ldots n$, an or-group is expressed as $f \Leftrightarrow f_1 \vee f_2 \vee \ldots \vee f_n$, and an alternative-group is expressed as

$$\bigwedge_{i=0}^{n} f_i \Leftrightarrow (f \wedge \bigwedge_{j \in [0,n] \setminus i} \neg f_j)$$

.

To also consider numeric options, the domain of a numeric option n can be conceived as an alternative-group since only one value from the domain can be selected at a time. Hence, each value of the domain $dom(n)$ can be conceived as a binary option. If value $v \in dom(n)$ is selected, this states $n = v$, otherwise $n \neq v$.

3.2.2 Grammar Expansion

Besides trying to find solution for satisfiability problems, the expression of variability models as context-free grammars (CFGs) enables the derivation of configurations directly from a CFG by expanding it. A first description of transformation rules, yet only for feature diagrams with binary options, was was proposed by Batory (2005). A hierarchical feature diagram can be recovered from a set of constraints using the algorithm of She et al. (2011) as explained in section 3.1.2. In the following we describe how a feature diagram with both binary and numeric features can be transformed to a context-free grammar, and how configurations can be derived subsequently.

Def. 3.2.1 (Context-free Grammar)**.** *A context-free grammar is a tuple $G = (N, T, S, P)$, consisting of a set of non-terminal symbols N, a set of terminal symbols T, a start word $S \in (N \cup T)^*$, and a set of productions $P \subseteq N \times (N \cup T)^*$. The set L_G describes the language of the grammar G and comprises all valid words $w \in L_G$ which can be derived from the start word $S \in L_G$ by applying productions a finite number of times to it.*

Following the Definition. 3.2.1, to derive all configurations for a given feature diagram, the idea is to first translate it to a context-free grammar. In order to do so, especially with respect to handling numeric options, we introduce an extended definition for a CFG, a configuration generation grammar.

Def. 3.2.2 (Configuration Generation Grammar). *A configuration generation grammar is a context-free grammar $G = (N, T, S, P)$ whose elements are constructed from a feature diagram as follows.*

- *All features represent non-terminal symbols N, which can be divided into two disjoint sets, binary non-terminal symbols $F_{\mathcal{B}}$ and numeric nont-terminal symbols $F_{\mathcal{N}}$. That is, $N = F_{\mathcal{B}} \cup F_{\mathcal{N}}$ and $F_{\mathcal{B}} \cap F_{\mathcal{N}} = \varnothing$.*

- *Similarly, the set of terminal symbols T is consists of two different sets, the binary terminal symbols $T_{\mathcal{B}}$ and the numeric terminal symbols $T_{\mathcal{N}}$, so that $T = T_{\mathcal{B}} \cup T_{\mathcal{N}}$ and $T_{\mathcal{B}} \cap T_{\mathcal{N}} = \varnothing$ with*

$$T_{\mathcal{B}} = \bigcup_{b \in F_{\mathcal{B}}} \{b_0, b_1\} \tag{3.1}$$

$$T_{\mathcal{N}} = \bigcup_{n \in F_{\mathcal{N}}} \bigcup_{v_i \in dom(n)} \{n_{v_i}\}. \tag{3.2}$$

- *All productions P are constructed from the hierarchy specified in the given feature diagram, the binary, and the numeric features. In our definition of a configuration grammar, each word w is expressed as a subset of (non-)terminal symbols, i.e., $w \subseteq (N \cup T)$. A word is a terminal word, if and only if it does not contain any non-terminal symbol. Accordingly, the set of productions is $P \subseteq N \times (N \cup T)$, and a production $p = (u, v) \in P$ is applied to a word w by removing non-terminal symbol u from word w and merging words v and w. Hence, the new word w' is defined as $w' = (w \setminus u) \cup v$.*

  The productions $P = P_H \cup P_F$ are constructed from the following disjoint two sets of productions:

$$P_H = \{(p, \{c, c_1\}) | p, c \in N \land c \implies p\} \tag{3.3}$$

$$P_F = \bigcup_{f \in (F_{\mathcal{B}} \cup F_{\mathcal{N}})} \bigcup_{v \in dom(f)} \{(f, v)\} \tag{3.4}$$

- *Finally, the start word $S \subseteq (N \cup T)$ consists the non-terminal representing the top-level feature in the given feature diagram. The set of respective configurations is described by all words which can be generated by a finite number of applications of productions to the start word S.*

Based on definition 3.2.2, we specify the following algorithm 1 to expand the configuration step by step and, thus, derive all valid words from the grammar. The algorithm repeatedly derives new words by applying every production possible to previously derived words. Once a word is terminal, i.e., does not contain any non-terminal symbol, it is returned and not expanded any further. The algorithm terminates if there is no word left

to expand in the queue of partial words.

Data: Configuration generation grammar $G = \{N, T, S, P\}$, and cross-tree
constraints C of the given variability model
Result: All configurations of the given variability model.

words = Queue();
words.enqueue(S);
while *words is not empty* **do**
    current = words.dequeue();
    if *current does violate any cross-tree constraint* **then**
        continue;
    end
    for $n \in N$ **do**
        if $n \in current$ **then**
            for $u \in \{u | (n, u) \in P\}$ **do**
                new = current;
                new = new $\setminus n$;
                new = new $\cup u$;
                if $N \cup new = \varnothing$ **then**
                    yield new;
                else
                    words.enqueue(new);
                end
            end
            break;
        end
    end
end

Algorithm 1: Expansion algorithm for a configuration generation grammar.

In addition to deriving all configurations from a variability model, the algorithm can also be used to derive partial configurations, such as binary-only configurations. To do so, the numeric features need to be removed from the set of non-terminal symbols. The only limitation of this algorithm is that, conceptually, it requires all numeric options to be mandatory features. This is due to the unspecified semantics of a numeric option being un- or deselected.

3.3 Configuration Sampling

When assessing properties for configurable software systems, it is infeasible to consider every possible variant. As previously stated in Section 2.x, with variability, interactions between features can emerge and can be the root cause for configuration-related faults. Hence, faults may only be uncovered under specific configurations. To not exhaustively assess all variants, a variety of strategies to select a sample set of configurations have been proposed. Every sampling strategy in the context of configurable software system is designed with respect to a *coverage criterion* (Apel et al., 2013). For instance, most feature

interactions happen to be pair-wise interactions, i.e., interactions between two features. Based on this finding, t-wise sampling is one widely used strategy, where configurations are part of the sample set, so that all t-tuples of features are selected together. Thus, all t-wise feature interactions are covered.

3.3.1 Binary Features

Most sampling strategies in the context of configurable software systems target binary features. Popular sampling strategies for sampling configurations of binary features include, but are not limited to the following (Apel et al., 2013; Medeiros et al., 2016):

□ *Feature coverage*: Configurations are selected, so that each feature is selected in at least one configuration (Apel et al., 2013).

□ *Most-enabled-disabled*: Configurations are selected, so that a maximal and minimal number of features is enabled (Medeiros et al., 2016).

□ *One-enabled*: Configurations are selected, so that each feature is enabled at a time (Abal et al., 2014). Similarly, with a *one-disabled* strategy, configurations are chosen, so that each feature is deselected at a time.

□ *Statement-coverage*: Configurations are selected, so that each variable block of code (for compile-time variability) is at least enabled in one variant (Tartler et al., 2014).

□ *T-wise sampling*: Configurations are selected, so that all t-tuples of all (binary) features are included in at least one configuration (Williams and Probert, 1996). That is, the upper bound for the sample size is $\binom{|F|}{t}$ for features F and $t \in \mathbb{N}$.

Medeiros et al. (2016) have compared different sampling strategies, among other things with respect to resulting sample size and fault detection rate. *Most-enabled-disabled* results in the smallest sample size, whereas *t-wise* sampling, especially for a greater t yields the largest samples. Regarding the detection of faults, *statement-coverage* performed poorly, whereas *t-wise* sampled samples, especially with a greater t unveiled most faults.
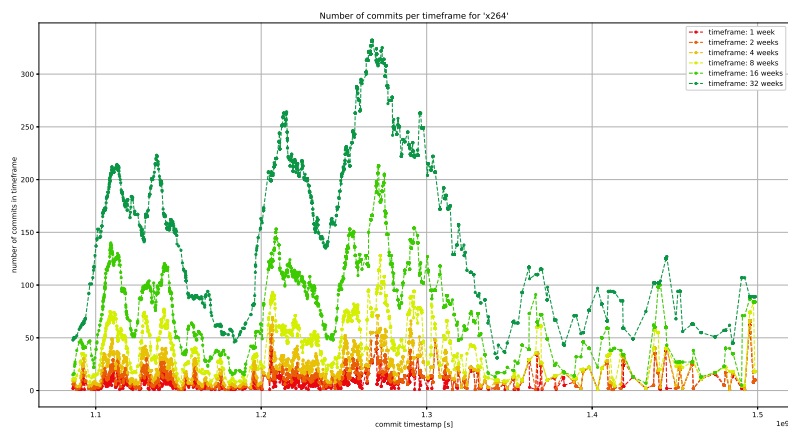
3.3.2 Numeric Features

□ Full factorial designs are infeasible

□ Numeric options usually only tune features and do not interact

□ Evaluation of game designs by Siegmund et al. (2015)

3.3.3 Configuration Sampling in Practice

4. Version Assessment

Number of commits per timeframe for 'xz'

Number of commits per timeframe for 'x264'
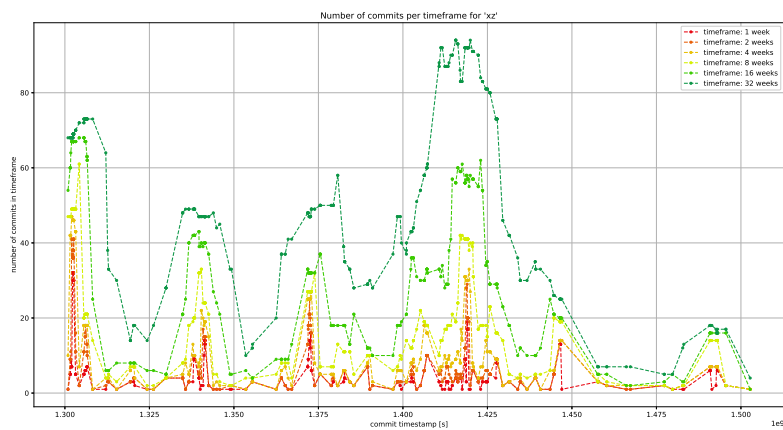
5. Performance Assessment

5.1 Performance Benchmarks

How performance benchmarks should be chosen. . .

5.2 Profiling

Which tools exist to profile programs, such as `GNU time`, . . .

5.3 Statistical Considerations

In this section, we now discuss the appropriate statistical means to summarize, compare and interpret measurement results. For the remainder of this section, we will refer to the following two scenarios. First, to obtain robust results, the assessment of a single variant needs to be repeated multiple times. Consequently, to report a single result per metric, the measurements for a single variant need to be summarized. Second, the assessment of performance for a variant may comprise several use cases, for instance file compression and decompression for a compression software. Therefore, performance measures aggregated from different benchmarks need to be summarized accurately.

5.3.1 Measures of Central Tendency

For each test run of a software system or variant, we obtain a single-valued measurement per performance metric. Since we repeat each test run n times per variant, we obtain a data record X with n measurements $X = X_1, X_2, \ldots, X_{n-1}, X_n$. While the arithmetic mean is commonly considered the right way to summarize data records and report a representative average value, we need to be more cautious with how to summarize data records (Fleming and Wallace, 1986; Smith, 1988). From a statistical perspective, the intention of summarizing a data record is to find a measure of central tendency what is representative for the data record. While the arithmetic mean is an appropriate method for many cases, there exist other means to summarize data records. Moreover, there exist a number of criteria for when to use which means to summarize a data record.

The first question when summarizing a data record is to ask what the data actually describe and what we intend to express with our summary. For a data record X, we can define a relationship we would like to conserve while replacing each single X_i with an average value $\bar{x}$. Based on this relationship, we can derive the appropriate method to summarize our record. For instance, our data record X describes the time elapsed

for a test case and we want the keep the following relation, saying that the sum of all measurements $\sum_{i=1}^{n} X_i$ is equal to the total time elapsed T, defined as

$$T = \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \bar{x}.$$

Based on the term above, we can derive the definition of the method appropriate to summarize our data record with respect to the conserved relation what is the *arithmetic mean*, defined as

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} X_i. \tag{5.1}$$

Consider another example, similar to the one above, where the test case is a load test with a predefined number of users and the measurements $X = X_1, X_2, \ldots, X_{n-1}, X_n$ are measured as hits per second. Again, we have different measurements we want to summarize with respect to a relation to conserve. Each user drops the same number of requests T, whereby the hit rate X_i and the respective elapsed time $t_i = \frac{T}{X_i}$ vary. We now conserve the relationship that the total number of requests for the data record is the sum of all hit rates X_i times the time elapsed t_i. Therefore, we want to substitute each hit rate X_i with an average value $\hat{x}$ so that the aforementioned relationship is conserved:

$$n \cdot T - \sum_{n}^{i=1} X_i t_i = 0 \Leftrightarrow n \cdot T - \hat{v} \sum_{n}^{i=1} t_i = 0 \Leftrightarrow n = \hat{x} \sum_{n}^{i=1} \frac{1}{X_i}$$

Similar to Eq. 5.1 we can derive the definition for the summarization method to use from the equation above what is the *harmonic mean*, defined as

$$\hat{x} = n \cdot \left(\sum_{i=1}^{n} \frac{1}{X_i} \right)^{-1} = \frac{n}{\frac{1}{X_1} + \frac{1}{X_2} + \ldots + \frac{1}{X_{n-1}} + \frac{1}{X_n}}. \tag{5.2}$$

We see that the summarization methods presented in Eq. 5.1 and 5.2 are useful for different types of data records, and should be used accordingly. The arithmetic mean is suitable for records, where the total sum of single measurements has a meaning, whereas the harmonic mean is suitable for measured rates or frequencies (Smith, 1988).

While the aforementioned measures of central tendency should be appropriate for most cases, they are not always the best choice though since the arithmetic and harmonic mean are heavily influenced by extreme observations (Shanmugam and Chattamvelli, 2015). For instance, for the data record $X = 1, 2, 3, 30$, three of four values are smaller than the arithmetic mean $\bar{x} = 9$. One option is to explicitly exclude outlier values from the data record. The so-called *trimmed mean* is obtained by truncating a upper and/or lower percentage t of the data record and, consequently, computing the (arithmetic or harmonic) mean for the remaining data record (Shanmugam and Chattamvelli, 2015).

While this method is suitable to omit the effects of outliers, one still needs to specify which upper and/or lower percentage t needs to be truncated. Moreover, the use a (trimmed) mean requires the data records' frequencies to be distributed symmetrically around its mean. A probability distribution is skewed (and therefore asymmetric) if,

graphically speaking, its histogram is not symmetric around its measure of central tendency. A simple method to measure the skewness of a probability distribution is *Bowley's measure* (Shanmugam and Chattamvelli, 2015), defined as

$$B_S = \frac{(Q_3 - M) - (M - Q_1)}{Q_3 - Q_1} = \frac{(Q_3 + Q_1 - 2M)}{Q_3 - Q_1}, \tag{5.3}$$

where Q_1 and Q_3 denote the first and third quartile, and M denotes the median of the probability distribution. The quartiles Q_i with $i \in \{1, 2, 3\}$ are defined as the values of a data record X, so that $\frac{i}{4}$ of the values of X are smaller than Q_i. The *median* is defined as Q_2, i.e., a value $M \in X$ so that half of the values in X are smaller than M.

The median itself is a more robust measure of central tendency than the aforementioned ones since it is less influenced by outliers and can be used for both skewed and symmetric data records (Shanmugam and Chattamvelli, 2015). For a given ascendingly ordered data record $X = X_1, X_2, \ldots, X_{n-1}, X_n$, we can compute the median as follows:

$$\text{Median}(X) = \begin{cases} X_{\frac{n+1}{2}} & \text{if } n \text{ is odd} \\ \frac{1}{2}\left(X_{\frac{n}{2}} + X_{\frac{n}{2}+1}\right) & \text{if } n \text{ is even} \end{cases} \quad \text{with } X_i \leq X_j \text{ and } i < j \leq n \tag{5.4}$$

5.3.2 Measures of Dispersion

Last, we take a look at different measures of spread or dispersion. Most commonly used are the *variance* σ^2 and the *standard deviation* $\sigma = \sqrt{\sigma^2}$ defined along the mean μ of a probability distribution as

$$\sigma^2 = \frac{\sum_{i=1}^{n}(X_i - \mu)^2}{n} \tag{5.5}$$

Similar to the (arithmetic or harmonic) mean, the variance and the standard deviation are heavily influenced by extreme observations (Shanmugam and Chattamvelli, 2015) and not the best choice in all cases. Instead, two more robust measures are the *median absolute deviation* (MAD) and the *inter-quartile range* (IQR). The MAD is defined as the median of the absolute deviations from the probability distributions' median (Molyneaux, 2014), or, defined as follows:

$$\text{MAD}(X) = \text{Median}\left(|X - \text{Median}(X)|\right) \tag{5.6}$$

The IQR, however, defines the range between the first and the third quartile, Q_1 and Q_3 (Shanmugam and Chattamvelli, 2015). This range is larger for a widespread data record and smaller for a data range with a narrow spread, but is not influenced by extreme observations as those outliers do not lie within the range $[Q_1, Q_3]$. The IQR is defined as

$$\text{IQR}(X) = Q_3 - Q_1. \tag{5.7}$$

5.3.3 When to use which measure?

In this section we discussed the arithmetic and harmonic mean as well as the median as a measure of central tendency, the symmetric property that is required to use the arithmetic or harmonic mean, and different measures of spread for a given data record. At the beginning, we have raised the question of how to summarize data records (a) for an experiment repeated multiple times, and (b) obtained from different benchmarks. While for case (b) the answer is to use the arithmetic or harmonic mean (depending on the quality of the measurements), for (a) the answer is a little more elaborate.

The arithmetic (or harmonic) mean can be used whenever the quality of the measurement is appropriate and the data record is symmetric. According to Shanmugam and Chattamvelli (2015), the arithmetic mean is preferable "when the numbers combine additively to produce a resultant value", such as time periods or memory sizes, whereas the harmonic mean is preferable "when reciprocals of several non-zero numbers combine additively to produce a resultant value", such as rates or frequencies (Smith, 1988). The median is a less precise estimator than the both means, yet more robust with regard to extreme observations. In addition, the MAD or IQR provide a more robust means of spread than the standard deviation.

5.3.4 Systematic Error

How different machines can be used to minimize systematic error

5.4 Summarization Strategies

□ How to summarize performance for a single version?

□ Minimal/Maximal configuration, best/worst performance measurement, . . .

6. Evaluation

7. Conclusion

Bibliography

Abal, I., C. Brabrand, and A. Wasowski
  2014. 42 variability bugs in the linux kernel: a qualitative analysis. Pp. 421–432. ACM Press.
  Cited on page 24.

Al-Kofahi, J., T. Nguyen, and C. Kästner
  2016. Escaping AutoHell: a vision for automated analysis and migration of autotools build systems. In *Proceedings of the 4th International Workshop on Release Engineering*, Pp. 12–15. ACM.
  Cited on page 17.

Aleti, A., B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya
  2013. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 39(5):658–683.
  Cited on page 18.

Alves, V., C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler
  2008. An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, Pp. 67–76. IEEE.
  Cited on pages 4 and 17.

Andersen, N., K. Czarnecki, S. She, and A. Wąsowski
  2012. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, Pp. 106–115. ACM.
  Cited on pages 4 and 18.

Apel, S., D. Batory, C. Kästner, and G. Saake
  2013. *Feature-Oriented Software Product Lines.* Berlin, Heidelberg: Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-37521-7.
  Cited on pages 1, 2, 7, 13, 23, and 24.

Bakar, N. H., Z. M. Kasirun, and N. Salleh
  2015. Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. *Journal of Systems and Software*, 106:132–149.
  Cited on pages 4, 17, and 18.

Batory, D.
  2005. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714,
  Pp. 7–20. Springer.
  Cited on pages 7, 20, and 21.

Benavides, D., S. Segura, P. Trinidad, and A. R. Cortés
  2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models.
  VaMoS, 2007:01.
  Cited on page 21.

Benavides, D., P. Trinidad, and A. Ruiz-Cortés
  2005a. Automated Reasoning on Feature Models. *CAiSE*, 5(3520):491 – 503.
  Cited on pages 20 and 21.

Benavides, D., P. Trinidad, and A. Ruiz-Cortés
  2005b. Using Constraint Programming to Reason on Feature Models. *SEKE*, Pp. 677
  – 682.
  Cited on page 20.

Breivold, H. P., I. Crnkovic, and M. Larsson
  2012. A systematic review of software architecture evolution research. *Information and
  Software Technology*, 54(1):16–40.
  Cited on pages 8 and 9.

Czarnecki, K. and U. W. Eisenecker
  2000. *Generative Programming: Methods, Tools, and Applications.* ACM
  Press/Addison-Wesley Publishing Co.
  Cited on page 7.

Darringer, J. A. and J. C. King
  1978. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60.
  Cited on page 16.

Dietrich, C., R. Tartler, W. Schröder-Preikschat, and D. Lohmann
  2012. A robust approach for variability extraction from the Linux build system. In *Pro-
  ceedings of the 16th International Software Product Line Conference-Volume 1*, Pp. 21–
  30. ACM.
  Cited on page 1.

Fleming, P. J. and J. J. Wallace
  1986. How not to lie with statistics: the correct way to summarize benchmark results.
  Communications of the ACM, 29(3):218–221.
  Cited on page 27.

Guo, J., K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski
  2013. Variability-aware performance prediction: A statistical learning approach. In
  *Proceedings of the 28th IEEE/ACM International Conference on Automated Software
  Engineering*, Pp. 301–311. IEEE Press.
  Cited on pages 3, 4, 12, 13, and 14.

Guo, Y., C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra
2011. Tracking technical debt — An exploratory case study. Pp. 528–531. IEEE.
Cited on page 3.

Heger, C., J. Happe, and R. Farahbod
2013. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, Pp. 27–38. ACM.
Cited on pages 3 and 11.

Hunsen, C., B. Zhang, J. Siegmund, C. Kästner, O. Le\s senich, M. Becker, and S. Apel
2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482.
Cited on page 1.

Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson
1990. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
Cited on pages 7, 18, and 19.

King, J. C.
1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.
Cited on page 16.

Kästner, C., S. Apel, and M. Kuhlemann
2009. A model of refactoring physically and virtually separated features. In *ACM Sigplan Notices*, volume 45, Pp. 157–166. ACM.
Cited on page 15.

Kästner, C., P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger
2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, volume 46, Pp. 805–824. ACM.
Cited on page 16.

Liggesmeyer, P.
2009. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media.
Cited on pages 8 and 10.

Lillack, M., C. Kästner, and E. Bodden
2014. Tracking load-time configuration options. Pp. 445–456. ACM Press.
Cited on page 16.

Medeiros, F., C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel
2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, Pp. 643–654. ACM.
Cited on page 24.

Medeiros, F., M. Ribeiro, R. Gheyi, S. Apel, C. Kastner, B. Ferreira, L. Carvalho, and
B. Fonseca
2017. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE
Transactions on Software Engineering.*
Cited on page *16*.

Molyneaux, I.
2014. *The Art of Application Performance Testing: Help for Programmers and Quality
Assurance*, 2 edition. O'Reilly Media, Inc.
Cited on pages *2, 3, 10, 11, 12,* and *29*.

Nadi, S., T. Berger, C. Kästner, and K. Czarnecki
2014. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, Pp. 140–151.
ACM.
Cited on page *18*.

Nadi, S., T. Berger, C. Kästner, and K. Czarnecki
2015. Where do configuration constraints stem from? an extraction approach and an
empirical study. *IEEE Transactions on Software Engineering*, 41(8):820–841.
Cited on pages *4, 17,* and *18*.

Nguyen, H. V., C. Kästner, and T. N. Nguyen
2014a. Building call graphs for embedded client-side code in dynamic web applications.
In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations
of Software Engineering*, Pp. 518–529. ACM.
Cited on page *16*.

Nguyen, H. V., H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen
2011. Auto-locating and fix-propagating for HTML validation errors to PHP serverside code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on
Automated Software Engineering*, Pp. 13–22. IEEE Computer Society.
Cited on page *16*.

Nguyen, T. H., M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora
2014b. An industrial case study of automatically identifying performance regressioncauses. In *Proceedings of the 11th Working Conference on Mining Software Repositories*,
Pp. 232–241. ACM.
Cited on pages *3* and *11*.

Parnas, D. L.
1994. Software aging. In *Proceedings of the 16th international conference on Software
engineering*, Pp. 279–287. IEEE Computer Society Press.
Cited on pages *8* and *9*.

Passos, L., K. Czarnecki, and A. Wąsowski
2012. Towards a catalog of variability evolution patterns: the Linux kernel case. In
Proceedings of the 4th International Workshop on Feature-Oriented Software Development, Pp. 62–69. ACM.
Cited on pages *2* and *10*.

Passos, L., J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente
2015. Feature scattering in the large: a longitudinal study of Linux kernel device drivers. Pp. 81–92. ACM Press.
Cited on pages 2 and 9.

Peng, X., Y. Yu, and W. Zhao
2011. Analyzing evolution of variability in a software product line: From contexts and requirements to features. *Information and Software Technology*, 53(7):707–721.
Cited on pages 2, 3, 4, 9, and 10.

Perry, D. E. and A. L. Wolf
1991. Software architecture. *Submitted for publication.*
Cited on pages 2 and 9.

Rabkin, A. and R. Katz
2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, Pp. 131–140. ACM.
Cited on pages 4, 17, and 18.

Sarkar, A., J. Guo, N. Siegmund, S. Apel, and K. Czarnecki
2015. Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Pp. 342–352. IEEE.
Cited on pages 3, 4, and 14.

Schaefer, I., L. Bettini, V. Bono, F. Damiani, and N. Tanzarella
2010. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, Pp. 77–91.
Cited on page 1.

Seidl, C., F. Heidenreich, and U. A\s smann
2012. Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, Pp. 76–85. ACM.
Cited on pages 2 and 10.

Shanmugam, R. and R. Chattamvelli
2015. *Statistics for scientists and engineers.* Hoboken, New Jersey: Wiley.
Cited on pages 28, 29, and 30.

She, S., R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki
2011. Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, Pp. 461–470. IEEE.
Cited on pages 4, 18, 19, 20, and 21.

Siegmund, N., A. Grebhahn, S. Apel, and C. Kästner
2015. Performance-influence models for highly configurable systems. Pp. 284–294. ACM Press.
Cited on pages 2, 3, 4, 13, 14, and 24.

Siegmund, N., S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake
2012. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, Pp. 167–177. IEEE.
Cited on pages 3, 4, 13, and 14.

Smith, J. E.
1988. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206.
Cited on pages 27, 28, and 30.

Tamrawi, A., H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen
2012. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering*, Pp. 650–660. IEEE Press.
Cited on page 17.

Tartler, R., C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann
2014. Static Analysis of Variability in System Software: The 90, 000# ifdefs Issue. In *USENIX Annual Technical Conference*, Pp. 421–432.
Cited on page 24.

Thüm, T., S. Apel, C. Kästner, I. Schaefer, and G. Saake
2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):1–45.
Cited on pages 1, 15, and 16.

Thüm, T., D. Batory, and C. Kastner
2009. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Pp. 254–264, Washington, DC, USA. IEEE Computer Society.
Cited on page 7.

White, J., B. Dougherty, and D. C. Schmidt
2009. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284.
Cited on page 1.

Williams, A. W. and R. L. Probert
1996. A practical strategy for testing pair-wise coverage of network interfaces. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, Pp. 246–254. IEEE.
Cited on page 24.

Woodside, M., G. Franks, and D. C. Petriu
2007. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, Pp. 171–187. IEEE.
Cited on pages 3, 11, and 12.

Xu, T., L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker
   2015. Hey, you have given me too many knobs!: understanding and dealing with over-
   designed configuration in system software. Pp. 307–319. ACM Press.
   Cited on page 17.

Zhang, B., M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen
   2013. Variability evolution and erosion in industrial product lines: a case study. In
   Proceedings of the 17th International Software Product Line Conference, Pp. 168–177.
   ACM.
   Cited on pages 2 and 9.

Zhang, Y., J. Guo, E. Blais, and K. Czarnecki
   2015. Performance prediction of configurable software systems by fourier learning (T).
   In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Con-
   ference on*, Pp. 365–373. IEEE.
   Cited on pages 4, 13, and 14.

Zhou, S., J. Al-Kofahi, T. N. Nguyen, C. Kästner, and S. Nadi
   2015. Extracting configuration knowledge from build files with symbolic analysis. In
   Proceedings of the Third International Workshop on Release Engineering, Pp. 20–23.
   IEEE Press.
   Cited on pages 4 and 17.