Technische Universität Braunschweig

Department of Computer Science

Master's Thesis

Assessing Performance Evolution Of Configurable Software Systems

Author:

Stefan Mühlbauer

August 14, 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

Contents

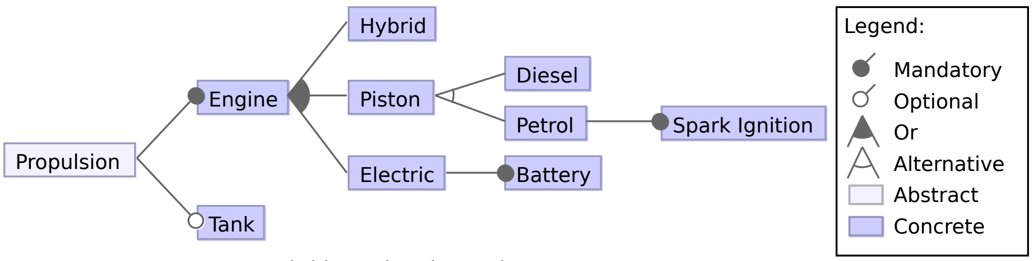
1. Introduction

Configurable Systems Modern software systems often need to be customized to specific customer needs. Customizable software, for instance, enable greater flexibility in supporting variable hardware platforms or tweaking system performance. To make software systems configurable and customizable they ship with a variety of *configuration options*, also called *features*, to select from (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. Configuration can be accommodated at different stages of the software life-cycle, either at *compile-* or *build-time* when the software is build or at *load-time* before the software is actually used. The main difference between applying configuration at either stage besides how variability is implemented is the effect on the software system. At compile-time parts of the code are explicitly in- and excluded from the variant derived from the configuration, whereas at customization at load-time usually just en- or disables functionality for one run while all functionality is available. Examples for configurable software systems range from small open-source command-line tools to mature ecosystems like Eclipse or even operating systems like the Linux kernel with more than 11.000 options (Dietrich et al., 2012).

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 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; 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 valid configuration. Moreover, feature diagrams provide a syntax for two different types of feature groups, *or-groups* or *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 such 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 Fig. 1.1. In this example an imaginary vehicle propulsion can be configured with eight valid configurations. The vehicle requires an engine, thus, 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.



Hybrid ⇒ Electric ∧ Piston

Piston ⇒ Tank

Fig. 1.1. Feature diagram for a feature model with eight valid configurations; two cross-tree constraints are specified as propositional formulas over features

For a configurable system configurations are primarily chosen to meet functional and non-functional requirements. Nonetheless, the selection of certain combinations of multiple features, of course, can happen to cause unexpected and undesired side effects as well. This so-called *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. A commonly referred to example of a feature interaction, as drafted by Calder et al. (2003), describes interactions in telecommunication networks. Given two independent features CallForwarding and CallWaiting, where CallForwarding forwards a call from a busy line to a line that is available, and where CallWaiting notifies a busy line when a call is on hold. In isolation their behavior is well-defined, but if both features are selected their oppositional behavior becomes problematic. If no precedence of one feature has been specified, the network might end up in race conditions or other unexpected behavior. That is, to avoid this feature interaction, for instance, precedence constraints must be implemented or the selection of both features must be mutually exclusive.

Thesis Statement and Scope A software system's performance depends on the functionality offered, the respective implementation, program load and the resulting execution. While feature interactions not necessarily cause the software system to break severely in all cases, its overall performance can degrade for corner cases or specific configurations as the feature selection influences the execution (Foo et al., 2010; Heger et al., 2013; Nguyen et al., 2014). That is, the choice of features as well shapes the performance of a software system.

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, where all options can be selected independently, the number of valid configurations grows exponentially with every feature added and likely 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 to assessing properties of configurable software systems exhaustively for each valid configuration is in general conceived infeasible. Despite this mathematical limitation many feasible approaches to static analysis for highly 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).

The aspect of performance of configurable software systems has gained more attention recently, even though from a practitioners view, according to Molyneaux (2014), for the most part performance testing is still not accommodated to an acceptable degree in the development process. Assessing performance for configurable systems incorporates obtaining knowledge about the performance of every valid configuration.

In the recent past a variety of approaches to model, learn and predict performance behavior of configurable software system have emerged. The scheme behind these approaches is the conception of performance modeling as an optimization problem, i.e., to recover and approximate performance behavior as a function of the selection of configuration options. Genetic algorithms (Guo et al., 2011; Sayyad et al., 2013) have shown reasonable results, yet are not able to handle constraints like mutual exclusion. Siegmund et al. (2012) proposed a method to predict performance for arbitrary variants following an approach for automated detection of feature interaction (Siegmund et al., 2012). Following their approach, in 2015 they proposed performance-influence models as a means to analyze and predict performance for configurable software systems (Siegmund et al., 2015). A performance-influence model attempts to approximate the influence of both single features and interacting features on the software systems' performance. The approach has shown a reasonably low error rate for several real-world applications and allows prediction of system performance for arbitrary configuration variants.

Going a step further, actively maintained configurable software systems evolve: Variability models may change as software has to adapt to changes in the functional requirements it is meant to meet. Patching and upgrading a software system affects the architecture or implementation that is likely to become inconsistent and degrade over time. While there exists substantial work on understanding the evolution of configurable systems, e.g., documenting common symptoms of architectural decay (Passos et al., 2015; Zhang et al., 2013) or attempting to classify patterns for variability evolution (Seidl et al., 2012; Peng et al., 2011; Passos et al., 2012), there is little we know so far about the evolution of performance or non-functional properties in configurable systems.

To get a better understanding software evolution and to address performance regression problems it is inevitable to continue studying the performance and performance evolution of configurable systems. In practice, all aforementioned approaches to model and predict performance behavior for a configurable software system require exhaustive records of performance measurements to learn from. Even though valid configurations can be sampled to some extent (Apel et al., 2013; Siegmund et al., 2015), assessing a single version of a configurable software system still demands a large number of valid configurations to be measured. In addition, to study the performance evolution of configurable software systems a history or series of performance models is required. That is, assessing performance evolution of configurable systems is infeasible without automated tool support.

The goal of this thesis is to provide a theoretical and practical foundation for exhaustive performance measurements of configurable software systems and series thereof. We contribute a guideline of and tool support for performance measurements for configurable and evolving software systems. Our research objectives and desired outcomes are

- a literature overview and discussion regarding software evolution and performance testing, especially with respect to the presence of variability and
- a practical tool for performance measurement for multiple revisions of configurable software systems.

Thesis Outline The Thesis is organized as follows. Chapter 2 recalls the background topics of the thesis theme, including software evolution, the foundations and statistical aspects of performance testing, variability model synthesis, and recent approaches to performance modeling. In Chapter 3 we present the overall measurement process and discuss the methods used for our performance measurement tool as well as its limitations. In Chapter 4 we evaluate practical aspects of our tool with respect to practicality and discuss the results thereof. Finally, Chapter 5 concludes the thesis and gives an outlook on possible future work.

2. Background

This chapter is intended to recapitulate the background of the thesis theme. In Sec. 2.1, we recall the evolution of software systems with respect to architecture and variability. In Sec. 2.2 we outline the characteristics of software performance, practical testing and measurement strategies as well as some statistical background necessary to analyze, interpret and compare performance assessment. In Sec. 2.3 we present recent approaches for feature model extraction from existing software systems and code artifacts. Finally, in Sec. 2.4 we recall and compare in detail different approaches to model and predict performance behavior for configurable software systems.

2.1 Evolving Software

Software Evolution The first notion of a software systems' 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. 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, like 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, 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 and maintainability as well as evolvability can degrade as software evolves.

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 regarding 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 and, therefore, knowledge about the architecture becomes unavailable. Although the unfavorable effects of software evolution do not necessary 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. They identify variability-related directives and clusters of those to 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* or interdependencies 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, e.g., 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 driver. 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.

Variability Evolution Apart from architecture evolution, the variability offered by software systems evolves as well. For configurable software systems (or software product lines; these terms are not equivalent, but every SPL is a configurable software system) evolution steps will not only affect artifacts in the solution space, yet also be visible in changes in the respective variability models. Although the variability aspect of software evolution has not been drawn as much attention to as has been on 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 as well as changing 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. identify major causes for variability evolution, comprising a) new driving contexts emerging, b) weakened 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 so far has been proposed as all mentioned work above focuses on those patterns that appeared to be relevant for the respective case study.

2.2 Performance Regression and Testing

- What is “performance”? (Molyneaux, 2014)
- Ideas, concepts and strategies (Molyneaux, 2014; Fleming and Wallace, 1986; Woodside et al., 2007)
- Performance regression testing (Nguyen et al., 2014; Foo et al., 2010) and root cause detection (Heger et al., 2013)

2.3 Feature Model Synthesis

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 (or software product line) 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), overly complex configurability (Xu et al., 2015) or configuration constraints originated in different layers of a software system, e.g.m build constraints or compiler constraints (Nadi et al., 2015).

Second, variability models have emerged to be a useful means in domain and domain analysis prior to developing a software system. As variability models group and organize 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 detailed feature summary.

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 address 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.

2.3.1 Feature Extraction

The first objective in recovering a variability model from a configurable system is to determine the set of available configuration options to select. 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.

Rabkin and Katz (2011) proposed a static, yet heuristic approach to extract configuration options along with respective types and domains. They exploit the usage of configuration APIs. Their approach works in two stages and commences with extracting all code sections where configuration options are parsed. Subsequently, 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, they 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 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 options' domain. The extraction method by Rabkin and Katz (2011) renders to be precise, but is limited, for instance, when an option leaves the scope of the source code. Nonetheless, for the systems they evaluated they recovered configuration options that were not documented, only used for debugging or even not used at all.

2.3.2 Constraint Extraction

The second, or an additional step in recovering a variability model is the extraction of configuration constraints. An approach proposed by Zhou et al. (2015) focuses on the extraction of file presence conditions from build files using symbolic execution. A more comprehensive investigation of configuration constraints and their origin is provided by Nadi et al. (2014, 2015). They propose an approach based on variability-aware parsing and 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, thus. 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 was only to recover 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.

2.3.3 Reverse Engineering of Feature Hierarchy

Besides recovering configuration options and respective constraints, to reverse engineer a feature model, one further step is required. The recovered knowledge needs a tree-like hierarchy, detection of feature groups and cross-tree constraints to be an acceptable feature diagram (Kang et al., 1990). While several approaches to the recover 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. 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 an semi-automated and interactive approach to synthesize a feature hierarchy. Their approach comprises three steps: 1) Specifying a feature hierarchy, 2) detecting and selecting feature groups, and 3) adding a cross-tree constraint formula to the feature model.

1. Their approach 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 depicts 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 the descriptions of features u and v 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 by with by the frequency of w in the description of feature p.

   The first ranking, called Ranked-Implied-Features (RIF), for each feature f ranks 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 ranking, a user selects for each feature 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.

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 incident adjacent 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. Those cliques 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 follow this distinction between mutex- and xor-groups and just use the term alternative-group mentioned in Sec. 1.

3. The cross-tree constraints 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 in the same manner 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.

2.4 Performance Modeling

- Genetic algorithms (Guo et al., 2011)
- Variability-aware modeling (Guo et al., 2013)
- via feature-interaction and performance influence models (Siegmund et al., 2012, 2015)

3. Performance Measurement Setup

3.1 Performance Measurement Infrastructure

3.1.1 Repository Retrieval

3.1.2 Configuration Generation

(Batory, 2005)

3.1.3 Sampling strategies

3.1.4 Benchmarks

3.1.5 Measurements with `GNU time`

3.1.6 Measurement Aggregation

3.2 Experiment Optimizations

3.3 Preprocessing Strategies

3.3.1 Variabilty Model Extraction

3.3.2 Build Mechnism Extraction

3.3.3 Release Commit Extraction

3.4 Case Study Corpus

4. Case Study Evaluation

5. Conclusion

Bibliography

Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., and Rummler, A. (2008). An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 67–76. IEEE.
Cited on page 7.

Andersen, N., Czarnecki, K., She, S., and Wąsowski, A. (2012). Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 106–115. ACM.
Cited on page 9.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-642-37521-7.
Cited on pages 1, 2, and 4.

Bakar, N. H., Kasirun, Z. M., and Salleh, N. (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 page 7.

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714, pages 7–20. Springer.
Cited on pages 1 and 11.

Breivold, H. P., Crnkovic, I., and Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40.
Cited on page 5.

Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141.
Cited on page 2.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co.
Cited on page 1.

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

Fleming, P. J. and Wallace, J. J. (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 7.

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41. IEEE.
Cited on pages 3 and 7.

Guo, J., Czarnecki, K., Apely, S., Siegmundy, N., and Wasowski, A. (2013). Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 301–311. IEEE Press.
Cited on page 10.

Guo, J., White, J., Wang, G., Li, J., and Wang, Y. (2011). A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208–2221.
Cited on pages 3 and 10.

Heger, C., Happe, J., and Farahbod, R. (2013). Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38. ACM.
Cited on pages 3 and 7.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
Cited on pages 1, 9, and 10.

Molyneaux, I. (2014). *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2 edition.
Cited on pages 3 and 7.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151. ACM.
Cited on page 8.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (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 7 and 8.

Nguyen, T. H., Nagappan, M., Hassan, A. E., Nasser, M., and Flora, P. (2014). An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 232–241. ACM.
Cited on pages 3 and 7.

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press.
Cited on page 5.

Passos, L., Czarnecki, K., and Wąsowski, A. (2012). Towards a catalog of variability evolution patterns: the Linux kernel case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM.
Cited on pages 3, 6, and 7.

Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., and Valente, M. T. (2015). Feature scattering in the large: a longitudinal study of Linux kernel device drivers. pages 81–92. ACM Press.
Cited on pages 3 and 6.

Peng, X., Yu, Y., and Zhao, W. (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 3, 6, and 7.

Perry, D. E. and Wolf, A. L. (1991). Software architecture. *Submitted for publication*.
Cited on pages 5 and 6.

Rabkin, A. and Katz, R. (2011). Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140. ACM.
Cited on pages 7 and 8.

Sayyad, A. S., Ingram, J., Menzies, T., and Ammar, H. (2013). Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 465–474. IEEE.
Cited on page 3.

Seidl, C., Heidenreich, F., and A\s smann, U. (2012). Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 76–85. ACM.
Cited on pages 3, 6, and 7.

She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 461–470. IEEE.
Cited on page 9.

Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. pages 284–294. ACM Press.
Cited on pages 3, 4, and 10.

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., and Saake, G. (2012). Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177. IEEE.
Cited on pages 3 and 10.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):1–45.
Cited on page 3.

Woodside, M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE.
Cited on page 7.

Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. (2015). Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. pages 307–319. ACM Press.
Cited on page 7.

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., and Savolainen, J. E. (2013). Variability evolution and erosion in industrial product lines: a case study. In *Proceedings of the 17th International Software Product Line Conference*, pages 168–177. ACM.
Cited on pages 3 and 6.

Zhou, S., Al-Kofahi, J., Nguyen, T. N., Kästner, C., and Nadi, S. (2015). Extracting configuration knowledge from build files with symbolic analysis. In *Proceedings of the Third International Workshop on Release Engineering*, pages 20–23. IEEE Press.
Cited on page 8.