Hildesheimer Informatik-Berichte

Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid

Implementation Metrics for Software Product Lines

A Systematic Literature Review

September 8, 2017

Report No. 1/2017, SSE 1/17/E
ISSN 0941-3014

Software Systems Engineering • Institut für Informatik
Universität Hildesheim •Universitätsplatz 1 • D-31134 Hildesheim

Abstract

Development of Software Product Lines (SPLs) requires additional implementation concepts to manage variability and to facilitate the derivation of individual products based on a common platform. These variability implementation concepts are not required for the development of single systems and, thus, are not considered in traditional software engineering.

Metrics are well established in traditional software engineering, but are typically not applicable to SPLs as they do not address variability management. Over time, a number of specialized product line metrics have been described in literature. However, no systematic description of the characteristics of these metrics is currently available.

We conducted a systematic literature review to identify variability-aware implementation metrics, designed for the needs of SPLs. We list these metrics according to the measured artifact types: variability models, code artifacts, and combined metrics measuring both artifact types. Further, we analyze to what extent these metrics were evaluated as a basis for qualitative conclusions.

Keywords: Software product lines; software product line engineering; software metrics; software product line metrics; systematic literature review; secondary study; literature survey

Contents

List of Figures

List of Listings

List of Tables

Part I

Systematic Literature Review

Chapter 1

Introduction

Software Product Line Engineering (SPLE) has been established to minimize costs and efforts, while maximizing the quality of products in a family of software products [CN02]. This requires additional implementation concepts to manage variability and to facilitate the derivation of individual products based on a common platform. These concepts are the variability model and the variation points, i.e., elements in code, which describe the point of impact of a variation in an artifact. Both concepts are not required for the development of single systems and, thus, are not considered in traditional software engineering.

In classical software engineering, software metrics are established concepts to measure properties of software. However, they are not defined for variability management concepts of SPLs. It remains unclear to what extent classical metrics can be used in the context of SPLs and whether they are sufficient to draw any conclusions.

Here, we present a systematic literature review to identify variability-aware implementation metrics, designed for the needs of SPLs. The goal of our study is to identify existing metrics as a basis to draw qualitative conclusions on implementation properties of product lines. Thus, we focus only on variability model metrics and code metrics, which take variation points into account. We include variability model metrics, because they are linked to all levels of product line realization, including implementation. Hence, we pursue the following research questions:

RQ1 Which kind of metrics have been defined for implementation artifacts of software product lines?

- **RQ1.1** Which metrics have been defined for variability models?
- **RQ1.2** Which metrics have been defined for code artifacts, which consider variability?

RQ1.3 Which metrics have been defined, which combine information of the variability model and variability of code artifacts?

RQ2 Which correlations between these measures and qualitative characteristics have been studied?

RQ2.1 Which correlations could be proven / disproven?

RQ2.2 Which proof methods have been used to prove / disprove correlations?

In summary, the main contributions of this report are:

- We present a systematic literature review to identify the state of the art of implementation-related metrics for software product lines. This literature review comprises peer reviewed articles of the last 10 years.
- We present a catalog of variability-aware metrics for software product lines. This comprises metrics for variability models, code artifacts, and metrics measuring both artifact types. This catalog provides an overview of metrics, which are available in this domain, but also identifies open issues for the research community.
- We analyze to what extent these metrics have been evaluated to show their ability to draw conclusions about qualitative aspects of the analyzed artifacts.

This report is structured as follows. In the remainder of this part, we present our search strategy to identify relevant work. In Part II, we answer **RQ1** by presenting the identified metrics. These metrics are structured according to the measured artifact types: variability models, code artifacts, and metrics measuring both artifact types. In Part III, we answer **RQ2** by analyzing to what extent evaluations have been applied to prove their ability to serve as a basis to draw conclusions about qualitative aspects of the analyzed artifacts.

Chapter 2

Research Method

The primary goal of this study is to provide an overview of variability-aware implementation metrics for software product lines. We followed the guidelines given by Kitchenham et al. [Kit04, KBB16]. This section presents the steps followed in this study.

2.1 Search Strategy

This section details the search strategy of this literature review to identify relevant papers. We searched in two digital libraries for relevant papers: ACM[1] and ScienceDirect (SD)[2]. For ACM, we selected the option *"The ACM Guide to Computing Literature"* to collect also work printed by other publishers than ACM.

We formulated search terms to restrict the search to papers in the context of software product lines and to find papers on software metrics. The terms for these two objectives are shown in Table 2.1. We took the terms for the identification of software product line-specific papers from an existing systematic review [VFAC14]. The terms for the identification of papers on metrics are designed to cover as many papers in this area as possible. In a preliminary search, we determined that many irrelevant papers in the area of software engineering contain at least one term of both categories in the full text. For instance, "software product line" was often used in related work or as part of the references, "metric" was often used for the evaluation of a new approach. We decided to restrict the search to papers, which mention the terms either in title, abstract, or in keywords to narrow down the search

[1] `http://dl.acm.org`
[2] `http://www.sciencedirect.com`

Topic	Terms
Software product lines (context)	`"software product line" OR "software product-line" OR "software product family" OR "software product-family" OR "software family based" OR "software family-based" OR "software variability" OR "software mass customization" OR "software mass customization production lines" OR "software-intensive system"`
Metrics (object of investigation)	`metric OR metrics OR "quantitative analysis" OR "quantitative characteristics" OR "complexity of variability" OR "variability complexity"`

Table 2.1: Search terms for the identification of relevant work.

for relevant papers. The detailed instantiation of the terms from Table 2.1 for the two digital libraries are given in Appendix A.

2.2 Selection of Studies

Here, we present our inclusion and exclusion criteria of our review. We expected not too many publications in the area of variability-aware metrics for software product lines. Further, we tried to discover the maximum of metrics. For this reason, we included also workshop publications.

Inclusion Criteria

- Papers published between 2007 and 2017 (search performed at 10.01.2017).
- Peer reviewed articles of journals, conferences, and workshops.
- Papers about variability-aware implementation metrics for artifacts of software product lines. This comprises the variability model, implementation artifacts, and instantiation artifacts, like build scripts.

Exclusion Criteria

- Papers with 5 pages or less to exclude short papers.

- Literature reviews, mapping studies, and tertiary reviews, because we addressed only primary studies.
- "Classical" code metrics for single systems, i.e., metrics for system without variability. This applies also to measures on products of a software product line, after the variability has been resolved.
- Metrics which are only applicable to artifacts which are the results of phases prior to implementation and testing phases, like requirements documents or architectural design.
- Metrics addressing software evolution, for example commit analysis.
- Metrics, which measure the development process, like the quality of the development or predict the costs.

Search Process Evaluation

We checked the appropriateness of the defined search terms, inclusion, and exclusion criteria on already known relevant work. On the one side, the definitions should ensure that we are able to discover relevant work, while on the other side the definitions should help to narrow the results to a size, which is still realistically manageable. Our test set contained the following work:

T1 Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering, 39(12):1611–1640, Dec 2013.

T2 Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In Proceedings of the 20th International Systems and Software Product Line Conference, SPLC '16, pages 65–73. ACM, 2016.

T3 Andrea Leitner, Reinhold Weiß, and Christian Kreiner. Analyzing the complexity of domain model representations. In Proceedings of the 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS '12, pages 242–248. IEEE Computer Society, 2012.

T4 Roberto E. Lopez-Herrejon and Salvador Trujillo. How complex is my product line? the case for variation point metrics. In Second International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS'08, pages 97–100, 2008.

By executing our search strategy, we could identify that our result set would be manageable, but still significant. We could also identify that our search process was also able to identify all demands of the test set. However, the defined exclusion criteria would eliminate **T4**. We debated the idea of relaxing the size constraint for papers, but a preliminary analysis showed that this would significantly increase the study set, but without any significant gains. Hence, we decided to keep the exclusion criteria but to take **T4** into account. In order to avoid any distortion, we clarify the impact of **T4** on the results where appropriate.

2.3 Execution

Figure 2.1 summarizes the search process of our literature review. In the first step, we performed our search at ACM and ScienceDirect with the defined search terms of Section 2.1. We limited the search results according to the publishing date (between 2007 and 2017) and to the page size of at least 6 pages. In this step, we found 149 publications in the database of ACM and 19 publications in the database of ScienceDirect, respectively. An overview of identified results arranged by publisher and database is given in Table 2.2.

In the second step, we merged the results. 155 papers remained after removing duplicates. We scanned title, keywords, and the title of the conference/journal to check their relevancy. We filtered publications missing an explicit reference to software product lines. For instance these were papers at the conference on *"Complex, Intelligent, and Software Intensive Systems (CISIS)"*, which publications are found based on the search strings for product lines even though the papers do not address SPLE. 102 papers remained after this step for which we read the abstracts. While reading the abstract, we filtered papers for which we were absolutely sure that the papers did not address the topic of this survey. 67 papers remained after this step for which we repeated this step based on introduction and conclusion. We kept 54 papers and read them completely. At the end, only 28 papers are found to be relevant for this study. They are are presented in Part II together with paper **T4**.

Table 2.2 provides an overview of our results. We found 168 publications of 11 different publishers. However, 13 publications were listed in both digital

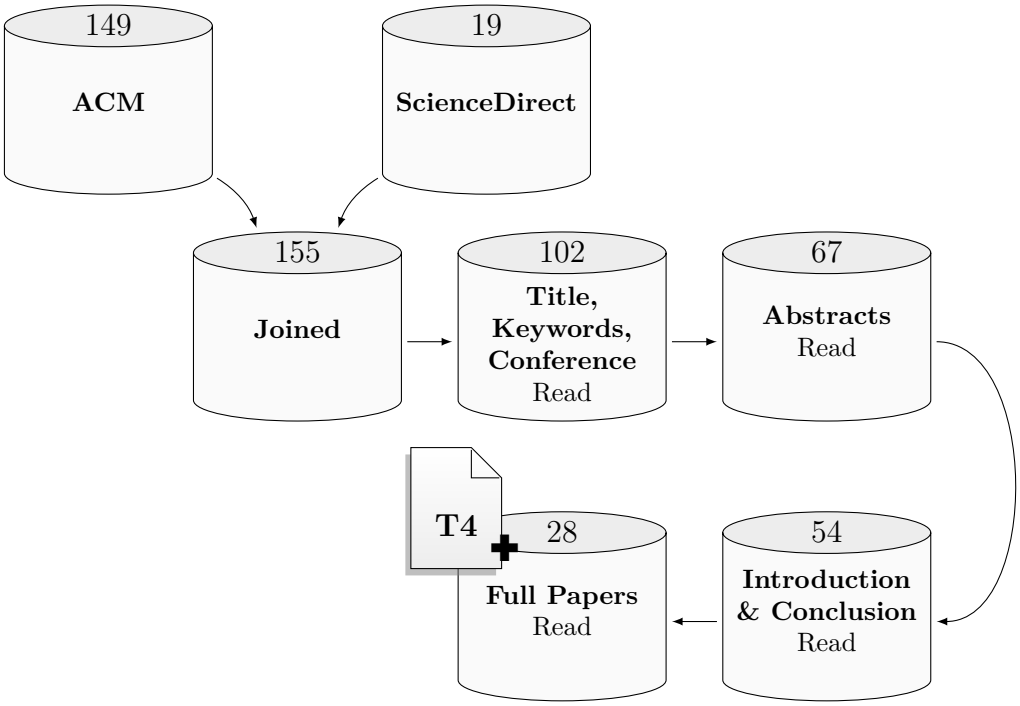

Figure 2.1: Filtering steps of paper selection.

libraries and, thus, were duplicates: These are the 4 journals published by Butterworth–Heinemann and 9 of 13 journals published by Elsevier. After the reviewing process 28 relevant publications remained. All of these 28 publications are contained in the ACM database, while only 3 publications are also contained in the ScienceDirect data base.

2.4 Study Quality Assessment

Here, we define how we assess the quality of the identified work. To answer **RQ1**, we collect all papers defining variability-ware implementations metrics for software product lines, independently of their quality. The result is presented in Chapter II. To answer **RQ2**, we assess the evaluation of the metrics. For this we define the following assessment criteria:

QA1 Does the paper evaluate the predictive power of the metric, i.e., is the metric an appropriate means to draw any kind of conclusion about the measures software? *Score:*

Publisher \ Database	Results		Relevant Publications	
	ACM	**SD**	**ACM**	**SD**
ACM	50	0	13	0
Butterworth–Heinemann	4		0	
Carnegie Mellon University	1	0	0	0
Elsevier	9	13	2	
IEEE	70	0	9	0
IOS	1	0	0	0
John Wiley & Sons	1	0	1	0
Kluwer Academic	5	0	2	0
Morgan Kaufmann	0	2	0	0
SCITEPRESS - Science and Technology Publications	2	0	0	0
Springer	6	0	1	0
Total	**149**	**19**	**28**	**3**

Table 2.2: Summary of search results per publisher (rows) and library (columns).

I The paper does not evaluate the metric at all. This applies to publications, which make use of a metric without any prove of the usefulness of the metric.

II The authors (manually) evaluate measured samples. For instance, the authors manually inspect the highest results of the metric or calculate values only for some known problems of their case study.

III The authors perform a systematic analysis based on a test set, e.g., based on a bug database.

IV A third party applied a manual assessment of the findings in addition to a systematic analysis. For instance, this may be done by developers of the system, which was used for the evaluation.

QA2 Does the paper mention that the metric was implemented? This question should answer whether the metric is suitable to measure real systems. *Score:*

I The publication does not provide any evidence that the metric was implemented, e.g., the authors provide and algorithm but do not mention whether the metric was ever realized.

II The authors claim that they implemented the metric, but do not provide any evidence of the existence, e.g., name no tool.

III The metric was evaluated based on a demo case, which was specifically developed for the evaluation. Through the explicit development of the demo case, it remains unclear if the metric is sufficient to cover all situations, which appear in the real world.

IV The metric was evaluated based on (artificial) demo cases developed by a third party. Such an evaluation shows that the metric / implementation is able to handle more than only a minimal setup as considered by the authors of the metric. However, this kind of evaluation still misses to prove that the metric is suitable for real life systems [3].

V The metric has proven to be applied on real world product lines, like Linux or on existing product lines from industry.

2.5 Data Collection and Analysis

For each publication found in the first step, we collect the full reference as provided by the libraries. Among others, this contains information about authors, title, conference/journal, year, number of pages, and keywords. This provides sufficient information for the first two filtering steps. After screening title, conference/journal, and keywords, we collected the abstracts for all of the remaining papers. The abstracts provided the necessary information for the next filtering step. After reading the abstract, we procured full-text versions of the remaining papers.

[3] According to [BSL+13, SLB+10], we treat case studies from S.P.L.O.T. [MBC09] artificial.

Part II

Metrics for Software Product Lines

Here, we present the discovered metrics of our systematic literature review to answer **RQ1**. For this, we reuse the original terminology and abbreviations of the reviewed papers as far as possible, while building up an integrated terminology. Sometimes smaller adaptions of the existing terminology are needed to avoid conflicts. For instance, for the use of features in code artifacts, there exist several different notations like feature, configuration options [FMK+16], Vars [ZBP+13], or feature constant [LAL+10, BG14]. Based on the reviewed literature, we decided to use the term *"Feature"* to refer to elements of the problem space, i.e., the variability model, while we use the term *"Feature Constant"* to refer to feature references of the solution space, i.e., code artifacts. We provide a mathematical definition for all metrics is possible. However, not all publications provide a clear definition of the presented metrics.

In Chapter 3 we present metrics defined for variability models to answer **RQ1.1**. Chapter 4 and 5 present metrics defined on code artifacts to answer **RQ1.2**. Chapter 4 introduces variability-aware metrics for classical programming languages, like C with preprocessor statements, annotated Java code, or build scripts. In Chapter 5, we show metrics designed for feature-oriented programming [ABKS13], which we treat as a special case of code metrics. Feature oriented programming languages are designed especially for product line development to separate common code from feature-specific extensions. In Chapter 6, we present metrics which are applied on the variability model and code artifacts in parallel to answer **RQ1.3**. In Chapter 7, we present general findings of our study, while gathering variability-aware implementation metrics for SPLs.

Each chapter starts with general modifiers, which may be applied to the objects of investigation to filter for specific elements of the same category. These modifiers may be used for multiple metrics to create new derivations of the listed metrics. Subsequently, we introduce the metrics independently of any evaluation. At the end of each chapter, we offer an overview of metrics presented in literature.

Chapter 3

Variability Model Metrics

In this chapter, we present metrics for variability models of software product lines. We start in Section 3.1 with modifiers, which serve as a kind of perspective to the variability model and may be applied to multiple metrics to change the elements of interest. In Section 3.2, we introduce the simplest metrics: size metrics. These metrics are used to measure different aspects of the size of the variability model. Section 3.3 presents complexity metrics and ratios based on the aforementioned metrics. In Section 3.4, we introduce metrics applied on single elements of the variability model. For instance this metrics calculate individual values for each feature in a feature model. Finally, in Section 3.5 we provide an overview of identified metrics. In this chapter, we use Figure 3.1 as integrated example to illustrate the results of metrics and the consequences of modifiers of the next section.

3.1 Modifiers

In this section, we describe identified modifiers specific to variability metrics. These modifiers can be regarded as different perspectives on the variability model and, thus, serve as a kind of filter which may be applied on multiple metrics.

3.1.1 Typed Features and Attributes

Some variability mechanisms are not pure Boolean, already the original FODA report admitted non-Boolean attributes associated with features [KCH+90]. Some authors pick this concept up and collect individual results for different types of features, for instance Strings and Numbers [PNX+11, BG14]. Further, features may be differentiated whether they are

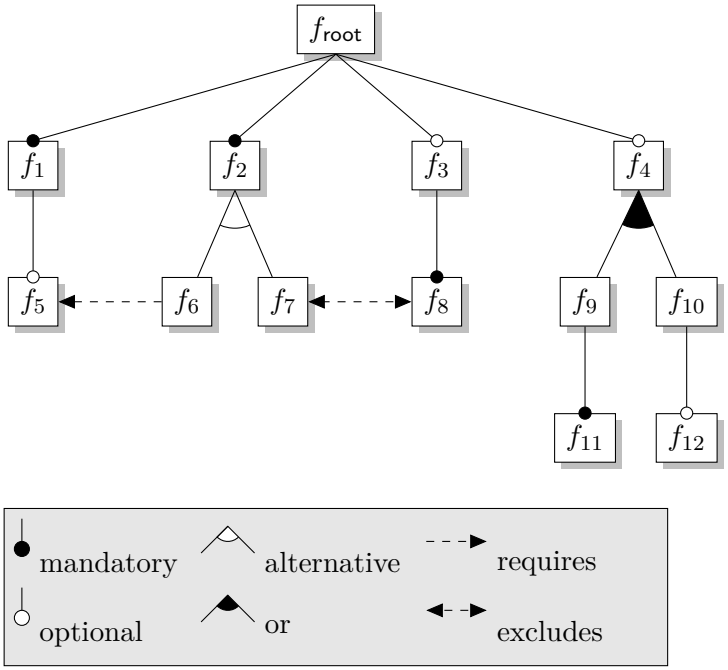

Figure 3.1: Example feature model to demonstrate metrics.

configurable (e.g., optional features) or constants (e.g., mandatory features with no configurable features in between the feature itself and the root feature) [PNX+11].

3.1.2 Atomic Sets (AS)

Zhang et al. [ZZM04] combine connected mandatory features to *atomic sets*, to reduce the complexity during automated verification. For a given feature f, they add all its mandatory child features, f itself, and its parent feature, if f is a mandatory feature, to the same atomic set. The result is a simplified model, which contains the same variability as the original feature model. An example is given in Figure 3.2.

3.1.3 Type of Influence

Mann et al. distinguish between different kind of dependencies between features or atomic sets [MR11]:

Positive influence (PI) A feature implies another feature.

Negative influence (NI) A feature implies the deselection of another feature, i.e., it excludes that feature.

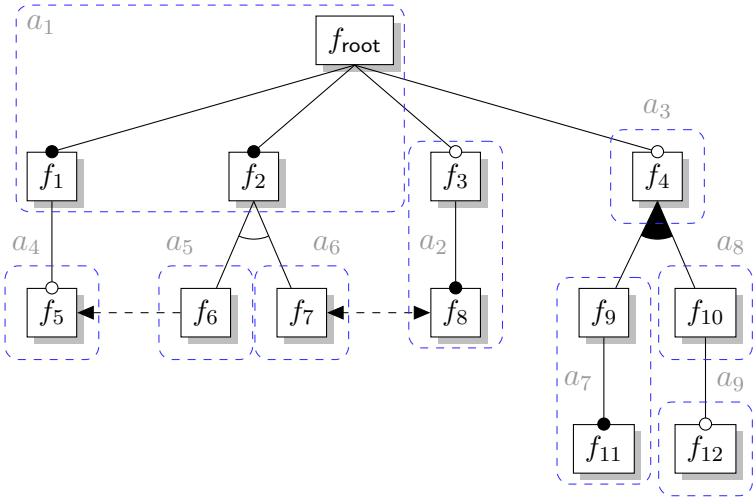

Figure 3.2: 13 features (f_{root}, f_1 – f_{12}) are mapped to 9 atomic sets (a_1 – a_9) retaining the same variability.

Preconditional influence (PreI) The feature is needed by another feature. This is the inverse relation to positive influence.

Positive or negative influence (P/NI) There is a dependency between two features, which does not apply to one of the aforementioned categories. Features of OR-feature groups and features involved in cross tree constraints, which cannot be expressed by requires/exclude-constraints belong to this category.

The authors further distinguish between direct and indirect relations. A direct relation only considers the direct connect features / atomic sets, while the calculation of the indirect relations considers only transitive relations. The indirect relations are labeled with a plus symbol (+). An example is given in Table 3.1.

3.1.4 Variability Points (VP)

Variability points serve as a counterpart to atomic sets. They define the number of decisions, which have do be made to configure a feature diagram. This comprises the number of optional features and feature alternatives [MR11]. However, the authors do not provide a precise definition and leave out how to treat OR feature groups. To close this gap, we describe here how we believe this should be done. This leads to results consistent to existent examples: Analogous to feature alternatives, we count the complete OR-group as one

AS	PI	PI+	NI	PreI	PreI+	P/NI
as_1	$\emptyset$	$\emptyset$	$\emptyset$	$\{as_2 - as_6\}$	$\{as_7 - as_9\}$	$\emptyset$
as_2	$\{as_1\}$	$\emptyset$	$\{as_6\}$	$\emptyset$	$\emptyset$	$\emptyset$
as_3	$\{as_1\}$	$\emptyset$	$\emptyset$	$\{as_7, as_8\}$	$\{as_9\}$	$\emptyset$
as_4	$\{as_1\}$	$\emptyset$	$\emptyset$	$\{as_5\}$	$\emptyset$	$\emptyset$
as_5	$\{as_1, as_4\}$	$\emptyset$	$\{as_6\}$	$\emptyset$	$\emptyset$	$\emptyset$
as_6	$\{as_1\}$	$\emptyset$	$\{as_5, as_2\}$	$\emptyset$	$\emptyset$	$\emptyset$
as_7	$\{as_3\}$	$\{as_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{as_8\}$
as_8	$\{as_3\}$	$\{as_1\}$	$\emptyset$	$\{as_9\}$	$\emptyset$	$\{as_7\}$
as_9	$\{as_8\}$	$\{as_1, as_3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.1: Example for type of influence based on the diagram from Figure 3.2. NI+ and P/NI+ are empty for all atomic sets in this example and for this reason omitted.

element. Further, this treatment is also similar to the definition of the NOr and the NXor metrics of Section 3.2.2.1 by Berger et al. [BG14].

3.1.5 Type of Constraints

Passos et al. [PNX+11] filter constraints with respect to their complexity, e.g., pure Boolean constraints, pure non-Boolean constraints and mixed constraints. The indention behind this filter is to provide a more precise view of the hardness of the models. They also differentiate between the purpose of the constraints, like visibility/existence conditions, value/range restrictions, and default values.

3.2 Size Metrics

In this section, we present size metrics defined for variability models. The most used size metric is the "Number of Features" presented in Section 3.2.1. For this metric various derivations are defined. Other size metrics are defined for the number of constraints, the maximal depth of the tree, and the size of the configuration space, i.e., the number of valid configurations.

3.2.1 Number of Features (NoF)

Number of Features (NoF) measures the total number of features which are present in a feature model, in other words, this metric counts the number of

all nodes in a feature model [BG11, BG14, NH14, MBBA15, AB11, MR11, MBBA16, PNX+11, SSRC14]:

$$NoF(fm) = \#features$$

This metric is often used to express the complexity of the modeled variability. There also exist a lot of variations of this metric:

N_{Top} This metric counts the number of features that are direct descendants of the feature model root [BG11, BG14, MBBA15, MBBA16].

N_{Leaf} This metric counts the number of features with no children or further specializations [BG11, BG14, MBBA15, MBBA16].

NGF This metric counts the number of features with at least one child (Number of Grouping Features) [BG14].

localMandatory/Optional Mann et al. [MR11] differentiate between features, which are mandatory or optional with respect to their direct parent features. These two metrics count the number of local mandatory and optional features. Also Leitner et al. measure the localOptional-metric, but they call this metric "element complexity" [LWK12].

pathMandatory/Optional Mann et al. [MR11] differentiate further between features, which are globally mandatory or optional, i.e., with respect to root feature. This differentiation is a subset of the modifiers of Section 3.1.1. These two metrics count the number of absolute mandatory and optional features.

Typed Some restrict the NoF-metric to specific types to collect individual results for different types of features, for instance Strings and Numbers (cf. Section 3.1.1) [PNX+11, BG14]. This metric was only applied for textual configuration languages, i.e., Component Definition Language (CDL) the configuration language of eCos and Kconfig the configuration language of Linux and other open source systems.

#constraintsfeatures This metric measures the number of features involved in the cross-tree constraints [SSRC14].

#atomicSets This metric count the number of atomic sets (cf. Section 3.1.2) [MR11]. The indention is to provide an overview about variable parts in a variable model. Atomic sets group features together that can be treated as one variable unit.

#variability points This metric count the number of variability points (cf. Section 3.1.4) [MR11]. The indention is to provide an overview about the decisions, which have to be made to derive a product. This metric is conceptually similar to the "variability coverage" metric used by Sánchez et al. [SSRC14][1].

dead / false optional features Mann et al. measure the number of optional features, which cannot be selected without violating constraints ("dead features"), and optional features, required by mandatory features ("false optional") [MR11].

Table 3.2 gives an overview of the presented NoF-metrics based on the example of Figure 3.1. For each metric, except for "Features by type", we present the calculated values as well as the counted elements, which lead to the result. The metric "Features by type" which is defined for typed, textual variability modeling approaches, could not be applied to the feature model due to the missing datatype definitions.

Metric	**Value**	**Counted Elements**
Number of features	13	$\{f_{\text{root}}, f_1 - f_{12}\}$
N_{Top}	4	$\{f_1 - f_4\}$
N_{Leaf}	6	$\{f_5 - f_8, f_{11}, f_{12}\}$
localMandatory	4	$\{f_1, f_2, f_8, f_{11}\}$
localOptional	8	$\{f_3 - f_7, f_9, f_{10}, f_{12}\}$
pathMandatory	2	$\{f_1, f_2\}$
pathOptional	10	$\{f_3 - f_{12}\}$
Number of grouping features	7	$\{f_{\text{root}}, f_1 - f_4, f_9, f_{10}\}$
#constraintsfeatures	4	$\{f_{\text{root}}, f_5 - f_8\}$
#atomicSets	9	$\{a_1 - a_9\}$
#variability points	6	$\{v_1 - v_6\}$

Table 3.2: Comparison of NoF-based metrics based on the example of Figure 3.1.

3.2.2 Constraint Measures

In this section, we list metrics that count different kind of constraints in a variability model.

[1]The authors of [SSRC14] count also the root feature, which leads to a different result.

3.2.2.1 Number of Constraints (NoC)

The number of constraints should indicate the complexity of the variability model as well as its hardness from the analysis perspective [PNX+11]:

$$NoC(fm) = \#constraints$$

However, many authors treat different elements as constraints. This ranges from requires/excludes constraints [BG11], via all cross-tree constraints, to constraints together with feature alternatives and OR-feature groups [LWK12]. We treat all cross-tree constraints (requires, excludes, and propositional logic) to be counted by this metric, to provide a clear definition. Based on our reading of the available literature this seems to be the most common understanding of this metric.

3.2.2.2 Interface Complexity (C_{If})

Leitner et al. treat also feature alternatives and OR-feature groups as constraints as they define "one of many" and "at least one of many" constraints [LWK12]. They count the number of cross-tree constraints with these groups to measure the complexity of relationships in a variability model:

$$C_{If}(fm) = \#constraints + \#OR\text{-}Groups + \#Alternatives$$

3.2.2.3 Derivations

The following metrics have been defined to count only a specific subset of the modeled constraints:

NOr The number of OR-feature groups, i.e. only "at least one of many" constraints [BG14].

NXor The number of XOR-feature groups, i.e. only "one of many" constraints [BG14].

NMutex The number of mutual exclusive constraints [BG14].

Operators Passos et al. count constraints using a specific type of operator (`==`, `!=`, `>`, ...) and combines this with the type modifier of Section 3.1.5 to show the importance of the operators for the different types of constraints [PNX+11].

3.2.2.4 Examples

Table 3.3 gives an overview of the presented NoC-metrics based on the example of Figure 3.1. For each metric except for "(typed) operators", we present the calculated values as well as the counted elements, which lead to the result. The metric "(typed) operators" which is defined for typed, textual variability modeling approaches, could not be applied to the feature model, since it does not contain appropriate constraints.

Metric	**Value**	**Counted Elements**
Number of Constraints	2	$\{<f_6, \overset{rq}{\rightarrow}, f_5>, <f_7, \overset{ex}{\leftrightarrow}, f_8>\}$
Interface Complexity	4	$\{<f_6, \overset{rq}{\rightarrow}, f_5>, <f_7, \overset{ex}{\leftrightarrow}, f_8>, f_2, f_4\}$
NOr	1	$\{f_4\}$
NXor	1	$\{f_2\}$
NMutex	1	$\{<f_7, \overset{ex}{\leftrightarrow}, f_8>\}$

Table 3.3: Overview of NoC-based metrics based on the example of Figure 3.1.

3.2.3 Other Size Metrics

Here we present further size metrics, which count other aspects than features or constraints. Also for these metrics, we present examples in Table 3.4.

3.2.3.1 Depth of Tree (DT)

The metric measures the length of the longest path from the root feature to leaf features [PNX+11, MBBA15].

3.2.3.2 Number of Valid Configurations (NVC)

This metric measures the size of the configuration space as it measures the number of all possible and valid configurations that can be derived from the feature model, while considering its structure and cross-tree constraints [BG11]. Also Mann et al. use the *"number of products of an input feature model"* as part of their metrics [MR11]. It is unclear whether this is practically computable for large-scale variability models. For instance, the variability model of Linux has more than 15.500 variables [ESKS17].

Metric	Value
Depth of Tree	4
Number of Valid Configurations	24

Table 3.4: Overview of other size metrics based on the example of Figure 3.1.

3.3 Ratio and Complexity Metrics

In this section, we present metrics which shall depict the complexity of the analyzed variability model. Most of these metrics are a combination of multiple size metrics.

3.3.1 Cross Tree-Constraints Ratio (CTCR)

This metric counts the number of distinct features involved in cross-tree constraints an divides them through the total number of features in the feature model [BG11, SSRC14]:

$$CTCR(fm) = \frac{\#constraintsfeatures(fm)}{NoF(fm)}$$

3.3.2 Ratio of Connectivity (RCon) & Density of the Graph (RDen)

Berger et al. define two derivations of the aforementioned metric [BG14]. In the context of Kconfig, "Ratio of Connectivity (RCon)" measures the percentage of features, which define at least one dependency to other variables. Please note that in Kconfig constraints are modeled as attributes of features. RDen measures the density of the graph, i.e., the average number of features referenced in constraints per feature.

3.3.3 Cyclomatic Complexity of Feature Models (CC)

This metric should count the number of distinct cycles in a feature model. The authors make use of the ability that feature models are in the form of trees and cycles can only be caused by cross tree constraints[2]. They count the number of these constraints [BG11]. As a consequence, this metric is conceptually identical to the "Number of Constraints" metric from

[2]The authors treat feature diagrams as undirected graphs and consider the binary constraints *requires* and *excludes* only.

Section 3.2.2.1. Meanwhile, this metric was also frequently used by other authors [SSRC14, MBBA15, MBBA16].

3.3.4 #variability points & Cyclomatic Complexity (VP&CC)

Sánchez et al. combine #variability points of Section 3.2.1 and cyclomatic complexity of the previous section for test case prioritization [SSRC14]. They use this metric for test case prioritization when testing with sample products:

$$VP\&CC(fm) = \sqrt{\#\text{variability points}(fm)^2 + CC(fm)^2}$$

It must be noted that the authors use this metric only for features and constraints, which are relevant for a specific product, i.e., they exclude constraints from the calculation of the cyclomatic complexity, if the involved features are not part of the measured product. We list this metric here, because the original sub-metrics are defined on product lines level and, thus, the combined metric may also be applied without considering individual products.

3.3.5 Ratio of Variability (RoV)

This metric measures the average branching factor of parent features in the feature model, i.e., how the average number of children per parent node [BG11]. While the authors do not provide the full details of the metrics definition, based on circumstantial evidence we reconstruct the metric as:

$$RoV(fm) = \frac{|parents(fm)|}{|childs(fm)|}$$

with:

$$parents(fm) = \{f_p \mid f_p \in fm \ \wedge f_p \text{ is not a leaf feature}\}$$
$$childs(fm) = \{f_c \mid f_c \in fm \ \wedge f_c \text{ is not the root feature}\}$$

This metric was also frequently used in other work [BG14, MBBA15, MBBA16].

3.3.6 Coefficient of Connectivity-Density (CoC)

This metric calculates the ratio of the number of edges over the total number of features (nodes). This includes connections between parent features and

their children, but also all cross-tree constraints independently if they are represented as graphical connections as in our example or if they are in textual form. In graph theory, this metric expresses how well the graph components are connected [BG11]. If $H = \{< p, c > \mid p \text{ is parent feature of } c\}$ are the hierarchical connections and CTC is the set of all cross-tree constraints[3], than this metric is measured as follows:

$$CoC(fm) = \frac{|H| + |CTC|}{NoF(fm)}$$

This metric was also used in the following publications [SSRC14, MBBA15, MBBA16].

3.3.7 Flexibility of Configuration (FoC)

This metric calculates the ratio of the number of optional features over all of the available features. The rationale behind this is that the more optional features exist in the feature model, the more choices are available for the designers to choose from, leading to a higher complexity of the configuration process [BG11].

3.3.8 Commonality and Variability Degrees (CR, VR)

This set of metrics measures the ratio of local/path variable (VR_{loc} / VR_{path}) and local/path common (CR_{loc} / CR_{path}) features over all features (cf. Section 3.2.1) by reusing the "localMandatory/Optional" and "pathMandatory/Optional" metrics of Section 3.2.1. Further, these metrics may be applied to atomic sets instead of features (cf. Section 3.1.2) [MR11]:

$$CR_{\text{path}}(fm) = \frac{pathMandatory(fm)}{NoF(fm)}$$
$$CR_{\text{local}}(fm) = \frac{localMandatory(fm)}{NoF(fm)}$$
$$VR_{\text{path}}(fm) = \frac{pathOptional(fm)}{NoF(fm)}$$
$$VR_{\text{local}}(fm) = \frac{localOptional(fm)}{NoF(fm)}$$

Please note: That the authors of [MR11] provide only the first formula. The other formulas are re-engineered based from their description.

[3]The authors consider the binary constraints *requires* and *excludes* only.

3.3.9 Actual Commonality and Variability Degrees (CR_{actual}, VR_{actual}, DR_{actual})

This set of metrics measures the ratio of *actual* variable (VR_{actual}), common/mandatory (CR_{actual}), and dead features (DR_{actual}) over all features (cf. Section 3.2.1). Contrary to the metrics of the previous section, these metrics consider the cross-tree constraints to calculate the *actual* state of each feature independently of its definition. Also these metrics may be applied to atomic sets instead of features (cf. Section 3.1.2) [MR11]:

$$CR_{\text{act}}(fm) = \frac{|reallyMandatory(fm) \cup actualMandatory(fm)|}{NoF(fm)}$$
$$VR_{\text{act}}(fm) = \frac{|reallyOptional(fm) \backslash actualMandatory(fm)|}{NoF(fm)}$$
$$DR_{\text{act}}(fm) = \frac{|actualDead(fm)|}{NoF(fm)}$$

reallyMandatory and *reallyOptional* contain features, which are modeled as mandatory or optional features. *actualMandatory* and *actualDead* contain false optional and dead features.

Please note that the authors of [MR11] provide only the first formula. The other formulas are re-engineered from their description.

3.3.10 Ratio of Typed Variables

This metric calculates the relative amount of typed variables (cf. Section 3.1.1) in contrast to the total number of all variables (cf. NoF-metric in Section 3.2.1). This metric was only applied for textual configuration languages, i.e., CDL and Kconfig [PNX+11, BG14].

3.3.11 Ratio of Attributed Variables

Some textual configuration languages like CDL and Kconfig provide the possibility to augment variables with further attributes. Among others, these are cross tree constraints, visibility conditions, and value restrictions. Berger at al. define a set of metrics to measure the ratio of variables with a specific attribute in contrast to the overall amount of defined variables [BG14]:

RConstr measures the ratio of variables that explicitly declare any kind of value restriction constraint.

RPurelyBoolConstr measures proportion of purely Boolean constraints among all declared constraints, which indicates to what extent propositional-logic-based reasoners are applicable on a model.

RDerived measures the ratio of variables with a constraint to their range. This can be further distinguished into two sub categories:

RDerivedExpr measures the ratio of values derived with an expression.

RDerivedLit measures the ratio of values set by a constant literal.

RVisibility measures the ratio of features with at least one visibility condition (expression).

RDefault measures the ratio of features with an explicitly modeled default value. These default values may be overwritten by the user. This can be further distinguished into two sub-categories:

RDefaultExpr measures the ratio of expressions.

RDefaultLit measures the ratio of literals used for such defaults.

Metric	**Value**	**Most Relevant Elements**
CC	2	$\{<f_6, \overset{rq}{\rightarrow}, f_5>, <f_7, \overset{ex}{\leftrightarrow}, f_8>\}$
VP&CC	$\sqrt{6^2+2^2} \approx 6.32$	
CTCR	$\frac{4}{13} \approx 0.31$	$\{f_5 - f_8\}$
RoV	$\frac{12}{7} \approx 1.71$	$\{f_{\text{root}}, f_1 - f_4, f_9, f_{10}\}$
CoC	$\frac{14}{13} \approx 1.08$	$\{f_{\text{root}}, f_1 - f_{12}\}$
FoC	$\frac{4}{13} \approx 0.31$	$\{f_3, f_4, f_5, f_{12}\}$
CR / $\text{CR}_{\text{actual}}$	$\frac{3}{13} \approx 0.23$	$\{f_{\text{root}}, f_1, f_2\}$

Table 3.5: Comparison of ratio and complexity metrics based on the example of Figure 3.1.

3.4 Element Metrics

The metrics of the previous subsections are applied on the whole variability model to measure its size or complexity. In this subsection, we present measures for single elements of a variability model. For instance to measure the complexity of a single feature.

3.4.1 Commonality of Feature (Comm)

The metric indicates the reuse ratio of a feature f in an SPL [SSRC14]. This is done by the division of the number of products in which feature f appears through the number of all products:

$$Comm(f, fm) = \frac{|filter(fm, f)|}{|products(fm)|}$$

with:

$$filter(fm, f) = \{p \mid p \text{ is product based on } fm \wedge f \text{ is selected in } p\}$$
$$products(fm) = \{p \mid p \text{ is product based on } fm\}$$

3.4.2 Variability Influence (VI)

Mann et al. measure with their *"variability influence (VI)"* metric how strong a feature influences the number of theoretically possible configurations (cf. Section 3.2.3.2) [MR11]:

$$VI(fm, f) = \begin{cases} 0 & fm \text{ is inconsitent } \vee f \text{ is dead or mandatory} \\ 1 - \frac{|configs(fm,f)|-1}{NVC(fm)} & \text{else} \end{cases}$$

with:

$$configs(fm, f) = \{c \mid c \text{ is legal configuration of } fm \wedge f \text{ is selected in } c\}$$

3.4.3 Theoretical (De-)Selection Ratio of Features

Mann et al. measure for each feature in how many percent of all configurations (cf. Section 3.2.3.2) it is selected or deselected [MR11]. They call this two metrics theoretical selection ratio (TSR) and theoretical deselection ratio (TDR) or features. They mention that these metrics could also be applied only on explicitly defined configurations to measure the *actual* selection/deselection ratios.

3.4.4 Coherence Between Atomic Sets ($\mathbf{Co_{imp}}$ / $\mathbf{Co_{pre}}$)

Mann et al. define two metrics to measure the coherence between atomic sets [MR11]. For these measures, they analyze the influence of connected atomic sets as described in Section 3.1.3. These two metrics are defined as follows:

$$\mathrm{Co}_{\mathrm{imp}}(as) = \frac{|PI(as)| + |PI + (as)| + |NI(as)| + NI + (as)}{\#atomicSets}$$

$$\mathrm{Co}_{\mathrm{pre}}(as) = \frac{|PreI(as)| + |PreI + (as)|}{\#atomicSets}$$

The authors claim, that these metrics are similar to coherence metrics between objects or components.

3.5 Overview

Table 3.6 provides an overview of the variability model metrics we could identify in literature. Two types of metrics are very prominent: Metrics that count features in a feature model and ratio-based metrics. There are only few metrics for the measurement of single elements of the feature model or quantifying aspects of constraints. For instance, while we found a definition for counting `mutual exclusive` constraints (NMutex, Section 3.2.2.1), we missed an analogous definition to measure `requires` constraints.

		[AB11]	[BG11]	[BG14]	[LWK12]	[MBBA16]	[MR11]	[MBBA15]	[NH14]	[PNX+11]	[SSRC14]
No. of Features	NoF	✓	✓	✓		✓	✓	✓	✓	✓	✓
	N_{Top}		✓	✓		✓		✓			
	N_{Leaf}		✓	✓		✓		✓			
	localMandatory						✓				
	localOptional				✓		✓				
	pathMandatory/Optional						✓				
	Typed			✓						✓	
	#constraintsfeatures										✓
	#atomicSets						✓				
	#variability points						✓				✓
	dead/false optional features						✓				
Constraints	NoC									✓	
	C_{If}				✓						
	NOr			✓							
	NXor			✓							
	NMutex			✓							
	Operators									✓	
	Typed									✓	
Depth of Tree			✓					✓			
No. of Valid Configurations			✓				✓				
Ratio & Complexity	Cross Tree-Constraints Ratio		✓								✓
	RCon & RDen			✓							
	Cyclomatic Complexity		✓			✓		✓			✓
	#variability points & CC										✓
	Ratio of Variability		✓	✓		✓		✓			
	CoC-Density		✓			✓		✓			✓
	Flexibility of Configuration		✓			✓	✓	✓			
	$CR_{(actual)}$, $VR_{(actual)}$, DR_{actual}						✓				
	Ratio of Typed Variables			✓						✓	
	Ratio of Attributed Variables			✓							
Elems.	Commonality of Feature										✓
	Variability Influence						✓				
	TSR/TDR of Features						✓				
	Co_{imp} / Co_{pre}						✓				

Table 3.6: Use of variability model metrics.

Chapter 4

Code Metrics

In this chapter, we present code metrics for software product lines for annotation-based implementation techniques. In the next chapter, we present code metrics, which are explicitly designed for composition-based implementation techniques, i.e., feature-oriented programming and aspectization. Both chapters answer research question **RQ1.2**. Contrary to Chapter 6, we list here pure code metrics only. These are metrics do not take any information of the variability model into account, e.g., complexity of constraints. However, these metrics usually need some mechanism to identify variability-related information, i.e., references to features. Usually the complete variability model is not needed for the identification, often a name pattern is sufficient to identify these references. For instance in Linux these references can be identified via the prefix `CONFIG_` [Kco17]. In the remainder, we call these references *feature constant*s as done in earlier publications [LAL+10, BG14].

4.1 Modifiers

In this section, we describe identified modifiers specific to code metrics. These modifiers can be regarded as different perspectives on the code and, thus, serve as a kind of filter which may be applied on multiple metrics.

4.1.1 Granularity of Variation Point

This modifier analyzes the granularity of a variation point, e.g., a C preprocessor directive. Thus, this modifier may be used to filter for code changes of a specific granularity.
Examples for C preprocessor granularity are [LAL+10, BG14]:

```c
#include <stdio.h>

#ifdef CALCULATION
int calc(int operand1, int operand2) {
    #if defined(CALCULATION) && defined(ADDITION)
        return operand1 + operand2;
    #elif defined(CALCULATION) && defined(SUBTRACTION)
        return operand1 - operand2;
    #endif
}
#endif

int main() {
    #ifdef GREETINGS
        printf("This is a demo.\n");
    #endif

    #ifdef CALCULATION
        int result = calc(3, 2);
    #endif

    #if defined(CALCULATION) && defined(ADDITION)
        printf("The result of 3 + 2 = %i.\n", result);
    #elif defined(CALCULATION) && defined(SUBTRACTION)
        printf("The result of 3 - 2 = %i.\n", result);
    #endif

    return 0;
}
```

Listing 4.1: Example C programm to demonstrate code metrics.

GL Changes on global level, e.g., adding a structure or function.

FL Changes on function level, e.g., adding an if-block or statement inside a function or a field to a structure.

BL Changes on block level, e.g., adding a block.

SL Changes on statement level, e.g., varying the type of a local variable.

EL Changes on expression level, e.g., changing an operator within an expression.

ML Changes on function signature level, e.g., adding a parameter to a function.

Couto et al. analyze the granularity of artifacts realizing one feature, which extends the aforementioned types for Java-based SPLs. Elements are only counted once, in the most coarse-grained category, e.g., classes of a package are not treated as "Class"-grained, if the whole package was measured as "Package"-grained [CVF11]:

Package The complete package, including its classes and interfaces, belongs to one feature.

Class A complete class or interface belongs to one feature.

ClassSignature A class signature was changed, e.g., `extends` and `implements` clauses.

InterfaceMethod Method signatures in a Java interface annotated to implement a feature.

Method Counts methods entirely annotated to implement a feature, i.e., signature and body.

MethodBody Counts methods whose body (but not the signature) has been annotated to implement a feature.

Attribute A class or instance attribute is changed by a feature.

Statement Filters for statements of a method, which implement a feature. This includes method calls, assignments, conditional statements, loop-statements, etc.

Expression Expressions annotated to implement a particular feature, e.g., the expression of an if- or loop-statement

Examples for make file granularity are [NH14]:

Dir Conditional compilation of complete (sub-)directories.

File Conditional compilation of files.

4.1.2 Variation Point Groups

Zhang et al. introduce the concept of variation point groups [ZB12, ZBP+13]. They group variation points by their expressions, i.e., treat variation points with the same expressions as identical. However, expressions with too many variables may become to hard to compare. For this reason, the authors compare only whether the same feature constants are used within the variation points and not whether the expressions are equivalent. Couto et al. count for each pair of features the occurrences in variation points, independently of whether the expressions are equivalent [CVF11].

4.1.3 Localization Within Methods

Some authors differentiate among the locations for changes within a method. These are [CVF11, RB09]:

StartMethod Statements annotated for a particular feature that appear at the beginning of a method.

NestedStatememt Statements annotated for a particular feature that appear nested in the scope of a non-annotated and more external statement

BeforeReturn Statements annotated for a particular feature that appear immediately before a `return` statement.

EndMethod Statements annotated for a particular feature that appear nested in the scope of a non-annotated and more external statement.

This differentiation is useful to analyze the usefulness of certain implementation techniques. For instance, changes of type "NestedStatememt" are more challenging to realize with aspectization [CVF11].

4.2 Size Metrics

In this section we present size metrics, which are applied to the complete product line platform. These metrics are used to compare the size of multiple product lines or sometimes to compare alternative implementation techniques.

4.2.1 Lines of Feature Code (LoF)

This metric is an adaptation of the classical lines of code metrics. This metric counts the number of lines that are linked to feature expressions, e.g., surrounded by `#ifdef`-blocks [LAL+10, CVF11, ZB12, GFFdAM12, GFFdAM14, BG14, HZS+16]. This metric may be combined with the granularity modifier of Section 4.1.1.

4.2.2 Number of Feature Constants (NoFC)

This metric is the counterpart of the metric of Section 3.2.1. This metric counts the number of feature constants used inside of code artifacts, e.g., distinct feature constants in `#ifdef`-blocks [LAL+10, ZBP+13, BG14, APFC15, HZS+16].

4.2.3 Number of Variation Points (NoVP) & Number of Variation Point Groups (NoVPG)

Zhang et al. define a metric to measure the number of variation points in code, i.e., variable parts of code elements like the total number of `#ifdef`-blocks [ZBP+13, HZS+16]. Instead of counting individual variation points, it is possible to group variation points by their expressions by (cf. Section 4.1.2) [CVF11, ZB12, ZBP+13].

4.2.4 Configuration Knowledge Scattering

Torres et al. count all files that have some sort of variation points (in their words "configuration knowledge") in them. For instance, source code files containing `#ifdef`-blocks [TKS+10]. The authors used this to measure the number of artifacts containing variable code parts, independently of any feature specification. For this reason, we classified this metric as size metric instead of a scattering metric (cf. Section 4.3).

4.2.5 Examples

Table 4.1 gives an overview of the presented size metrics for code based on the example of Listing 4.1. For each metric, we present the calculated values as well as the counted elements, which lead to the result. The metric "(typed) operators" which is defined for typed, textual variability modeling approaches, could not be applied to the feature model, since it does not contain appropriate constraints.

Metric	**Value**	**Counted Elements**
NLoF	9	Lines: 4, 6, 8, 10, 13, 15, 19, 23, 25
NoFC	4	CALCULATION, ADDITION, SUBTRACTION, GREETINGS
NoVP	7	VPs: ⟨3,11⟩, ⟨5,7⟩, ⟨7,9⟩, ⟨14,16⟩, ⟨18,20⟩, ⟨22,24⟩, ⟨24,26⟩
NoVPG	4	⟨CALCULATION⟩, ⟨CALCULATION ∧ ADDITION⟩, ⟨CALCULATION ∧ SUBTRACTION⟩, ⟨GREETINGS⟩
Configuration Knowledge Scattering	1	Example contains only 1 file.

Table 4.1: Overview of code size metrics based on the example of Listing 4.1.

4.3 Scattering Degree Metrics (SD)

These metrics are intended to analyze the scattering degree of code in relation to a feature. This can be done per feature constant to retrieve individual values for each feature. The surveyed work published aggregated values (maximum and average values) to compare different product lines.

4.3.1 Scattering of Variation Points ($\mathbf{SD_{VP}}$)

This metric measures how many variation points, e.g., `#ifdef`-blocks are affected by a feature constant [LAL+10, CVF11, ZB12, ZBP+13, BG14, NH14, HZS+16]. An example is given in Table 4.2.

4.3.2 Scattering of Variation Point Groups ($\mathbf{SD_{VPG}}$)

The SD_{VP}-metric may be combined with the modifier of variation point groups (cf. Section 4.1.2) to count the occurrences of similar variation point

VPs	Value	Counted VPs
CALCULATION	6	⟨3, 11⟩, ⟨5,7⟩, ⟨7,9⟩, ⟨18,20⟩, ⟨22,24⟩, ⟨24, 26⟩
ADDITION	2	⟨5,7⟩, ⟨22,24⟩
SUBTRACTION	2	⟨7,9⟩, ⟨24,26⟩
GREETINGS	1	⟨14,16⟩

Table 4.2: Example of the SD_{VP}-metric based on the example of Listing 4.1.

groups, e.g., how often is a given expression used in `#ifdef`-blocks [ZB12]. However, the authors treat expressions based on the same feature constants as identical independently of the logical equality. This is done for reasons of computability. An example is given in Table 4.3.

Variation Point Groups	Value	Counted Groups
CALCULATION ∧ ADDITION	2	⟨5,7⟩, ⟨22,24⟩
CALCULATION ∧ SUBTRACTION	2	⟨7,9⟩, ⟨24,26⟩

Table 4.3: Example of the SD_{VPG}-metric based on the example of Listing 4.1.

4.3.3 Scattering of Files (SD_{File})

This metric measures in how many files a feature constant is used in [ZBP+13, HZS+16].

4.3.4 Concern Diffusion over Components (CDC)

This metric measures the number of components, which belong to a concern, i.e., a feature [RB09, ARG+11, GFFdAM12].

4.3.5 Concern Diffusion over Objects (CDO)

Additionally to the aforementioned metric, this metric measures the number of methods and constructors, which belong to a feature [GFFdAM12].

4.3.6 Scattering of Code Elements

Maazoun et al. define several metrics to measure the influence of a feature constant to different elements on the conceptual level of programming

[MBBA16]. They measure the number of packages, classes, methods, attributes, and associations which belong to a feature. The metrics for measuring the number of classes and methods are conceptual similar to the CDC and the CDO metric.

4.3.7 Pairs of Cloned Code (PCC)

Andrade et al. measure pairs of cloned code as it could harm maintenance [ARG+11, ARRB13]. For this, the authors consider code to be cloned if two pairs have at least 40 continuous tokens in common. They use this metric to compare different implementation techniques in terms of maintainability.

4.4 Tangling Degree Metrics (TD)

These metrics serve as a counter part to the scattering degree metrics of Section 4.3. Instead of measuring how many elements are associated to a specific feature constant, they measure the number of feature constants involved in a specific element. Also these metrics can be measured per element to retrieve individual values or be aggregated to maximum/average values to compare the tangling degree over different product lines.

4.4.1 Tangling of Variation Points ($\mathbf{TD_{VP}}$)

Some authors measure how many feature constants are involved in expressions of variation points, e.g., `#ifdef`-blocks [LAL+10, ZBP+13, BG14, NH14, HZS+16]. Liebig et al. argue that a high number of feature constants in expressions may impair program comprehension [LAL+10].

4.4.2 Tangling of Files ($\mathbf{TD_{File}}$)

Zhang et al. measure the number of different feature constants used inside a file. While the authors originally named this metric *"Var fan-in on file"* [ZBP+13] they changed its name later to TD_{File} [HZS+16]. This should indicate the complexity in terms of variability realization by measuring the number of involved feature constants per file.

4.5 Variability Density Metrics

These metrics shall provide insights of variable code parts of the overall implementation of the product line.

4.5.1 Fraction of Annotated Lines of Code (PLoF)

This metric calculates the ratio of variable code lines (LoF, cf. Section 4.2.1) over all lines of code (classical LoC-metric) [HZS+16].

4.5.2 Internal/External-ratio Feature Dependency (IFD/EFD)

Apel and Beyer define two metrics to measure the cohesion of code elements, which belong to the same feature. This is done based on the following four definitions [AB11]:

elems(F) is a set of all program elements, e.g., methods, fields, classes, which belong to the feature F. The authors assume that each feature introduces at least one element and that every element is introduced by at least one feature, possibly by multiple features.

intdep(F) are the connections between two program elements <v, w> of the feature F (including self-references).

dep(v) is the set of all program elements w, which are connected to program element v. These elements do not necessarily belong to the same feature.

dep(F) $dep(F) = \bigcup_{v \in elems(F)} dep(v)$[1] are all dependencies of program elements, which belong to Feature F. Again, these connections are not necessarily only between elements of the feature.

The authors define the following to feature dependency metrics:

Internal-ratio Feature Dependency (IFD) measures the number of internal dependencies in relation to the total number of potentially possible internal dependencies ($|elems(F)|^2$) of a feature:

$$IFD(F) = \frac{|intdep(F)|}{|elems(F)|^2}$$

External-ratio Feature Dependency (EFD) measures the number of internal dependencies in relation to the total number of actual depen-

[1] dep(F) was originally defined as a sum. Based on the use in the original publication, we believe that this was a mistake and present a corrected definition based on an union.

dencies of a feature. $EFD(F) = 0$ indicates that F is not cohesive, while $EFD(F) = 1$ indicates that F is very cohesive:

$$EFD(F) = \frac{|intdep(F)|}{|dep(F)|}$$

4.5.3 Distance-based Internal/External-ratio Feature Dependency ($\mathbf{IFD_W/EFD_W}$)

Apel et al. also define two metrics based on the distance of program elements belonging to the same feature. For this further definitions are needed [AB11]:

||p(v) - p(w)|| measures the Euclidean distance between two program elements.

diag(p) is the maximal length of of two program elements in program p.

The authors define the following two distance-based feature dependency metrics:

Distance-based Internal-ratio Feature Dependency ($\mathbf{IFD_W}$)
measures whether elements of a feature are located at the same place:

$$IFD_W(F) = \frac{\sum\limits_{<v,w>\in intdep(F)} \Big(diag(p) - ||p(v) - p(w)||\Big)}{diag(p) \cdot |elems(F)|^2}$$

Distance-based External-ratio Feature Dependency ($\mathbf{EFD_W}$)
measures the ratio of the distance between program elements belonging to the same feature in relation to all connected program elements:

$$EFD_W(F) = \frac{\sum\limits_{<v,w>\in intdep(F)} \Big(diag(p) - ||p(v) - p(w)||\Big)}{\sum\limits_{<v,w>\in dep(F)} \Big(diag(p) - ||p(v) - p(w)||\Big)}$$

4.5.4 Normalized Average/Maximum Radius (NAR/NMR)

Complementary to the distance-based metrics of the previous section, the authors also define two metrics that are based relative distances. For this, a further definition is needed [AB11]:

bc(F) The barycenter is the arithmetic mean over all positions of the elements of F.

The authors define the *dependency radius* of a feature. For this, they define the following two metrics:

Normalized Average Radius (NAR) measures the normalized average radius of the distances between elements of a feature f as follows:

$$NAR(F) = 1 - \frac{\underset{v \in elems(F)}{\text{mean}} ||p(v) - bc(f)||}{diag(p)}$$

Normalized Maximal Radius (NMR) measures the normalized maximal radius of the distances between elements of a feature f as follows:

$$NMR(F) = 1 - \frac{\underset{v \in elems(F)}{\max} ||p(v) - bc(f)||}{diag(p)}$$

4.6 Complexity Metrics

Here, we present metrics intended to measure the complexity of code based on variation points.

4.6.1 (Number of) Internal #ifdefs

Ferreira et al. assume that the number of nested `#ifdef`-blocks influence the understandability of the code as the developer has to reason about each block while trying to understand or to modify a piece of variable code. They measure for each function the number of `#ifdef`-blocks independently whether they are nested or in sequence as an approximation of the complexity. This is done independently of the number of involved feature constants [FMK+16].

4.6.2 Nesting Depth (ND)

This metric is an adaptation of the classical metric of counting the nesting depth of if-statements to express complexity [CDS86]. Here, the authors measure the nesting depth of `#ifdef`-blocks, i.e., variation points, to indicate the complexity of the variability. This was done as:

ND_{avg} This metric measures the *average* value for the complete system [LAL+10, ZB12, BG14, HZS+16].

ND_{max} This metric measures the *maximum* value for the complete system [BG14, HZS+16].

ND_{VP} Zhang et al. measure the nesting depth for each *variation point*, while also considering `#ifdef`-blocks not related to feature constants. They list the number of variation points for the discovered levels [ZBP+13].

4.6.3 (Number of) Internal Configuration Options

This metric counts the number of distinct feature constants (configuration options in Linux) used inside a function [FMK+16]. The intuition is that the higher the number of features affecting code inside a function, regardless of how many `#ifdef`-blocks are in the function, the harder the code is to maintain, due to the increased number of feature constants a developer has to consider. Contrary to the metric of Section 4.6.1 this metric captures complexity by accounting for the amount of variability inside a function .

4.6.4 (Number of) External Configuration Options

This metric counts the number of distinct feature constants (configuration options in Linux), which control the conditional inclusion of the complete function [FMK+16]. This includes feature constants used in build scripts to include a C-file as well as all feature constants inside a file, which are used to control the inclusion of the analyzed function. The authors expect that functions with more feature constants involved in their inclusion criteria are less frequently used and tested. For this reason, they expect these functions to be more likely to contain vulnerabilities.

4.6.5 Degree Centrality

Degree Centrality [New10] measures the immediate importance of a node in the (weighted) graph by counting how many edges connect that node to other nodes. The authors of [FMK+16] adapt this metric and measure the incoming and outgoing calls for a function and add weights based on the number of external configuration options (cf. Section 4.6.4). They add 1 to the value of the aforementioned metric to consider also functions, which are independent of any feature constants. The authors expect that functions with a high complexity value are called (or calling other functions) often and are under complicated conditions more difficult to understand and more likely to be vulnerable.

4.6.6 Eigenvector Centrality

This metric is an extension of the aforementioned metric as it calculates the eigenvector centrality [New10], which is effectively a recursive version of the degree centrality, assigning higher values to nodes in neighborhoods of other nodes with high values. The intuition of the authors is that this metric should be higher for functions with complicated conditional call relationships to other functions, especially in neighborhoods where many such complicated call relationships exist [FMK+16].

4.6.7 Betweenness Centrality

Also this metric is based on a classical network metric [New10]. It measures how many times a node acts as a bridge along the shortest path between two other nodes. For this metric, the authors incorporate the number of external configuration options (cf. Section 4.6.4) that constrain the edge, and consequently modify the strength of alternative shortest paths (chain of function calls) between two other functions [FMK+16].

4.6.8 Variation Point Fan-in on File

Zhang et al. define this metric to measure the number of variation points, e.g., `#ifdef`-blocks, in a file. They consider this metric as a fan-in measure for files. The authors use this metric together with the TD_{File}-metric of Section 4.4.2 to express the complexity of a file [ZBP+13].

4.6.9 Variation Point Cyclomatic Complexity

Lopez-Herrejon and Trujillo adapted the metric of cyclomatic complexity developed by McCabe [McC76]. This metric was originally defined based on control structures only for the development of single systems, e.g., operating on `if-else` statements and loops. However, Lopez-Herrejon and Trujillo transfer this concept to code of software product lines and use variation points as basis for measurement as these use similar control structures, e.g., `#ifdef`-blocks [LT08].

4.7 Overview

Table 4.4 provides an overview of variability-aware code metrics. Most metrics are designed from scratch and are not based on established code metrics. Exceptions are the LoF-metric (cf. Section 4.2.1), based on lines of code

metrics for single systems and the *"Variation Point Cyclomatic Complexity"* metric (cf. Section 4.6.9), which is based on McCabe's metric of cyclomatic complexity.

		[APFC15]	[ARRB13]	[ARG+11]	[AB11]	[BG14]	[CVF11]	[FMK+16]	[GFFdAM12]	[GFFdAM14]	[HZS+16]	[LAL+10]	[MBBA16]	[NH14]	[RB09]	[TKS+10]	[ZB12]	[ZBP+13]	[LT08]
Size	LoF					✓	✓		✓	✓	✓	✓					✓		
	NoFC	✓				✓					✓	✓						✓	
	NoVP										✓							✓	
	NoVPG						✓										✓	✓	
	CK Scattering															✓			
Scattering	SD_{VP}					✓	✓				✓	✓		✓			✓	✓	
	SD_{VPG}																✓		
	SD_{File}										✓							✓	
	CDC			✓					✓				✓		✓				
	CDO								✓				✓						
	Code Elements												✓						
	PCC		✓	✓															
TD	TD_{VP}					✓					✓	✓		✓				✓	
	TD_{File}										✓							✓	
Variability Density	PLoF										✓								
	IFD/EFD				✓														
	IFD_W/EFD_W				✓														
	NAR/NMR				✓														
Complexity	No. of Int. #ifdefs							✓											
	ND_{avg}					✓					✓	✓					✓		
	ND_{max}					✓					✓								
	ND_{VP}																	✓	
	Int. Config. Options							✓											
	Ext. Config. Options							✓											
	Degree Centrality							✓											
	Eigenvector Centrality							✓											
	Betweenness Centrality							✓											
	VP Fan-in on File																	✓	
	CC of VPs																		✓

Table 4.4: Use of variability-aware code metrics.

Chapter 5

Metrics for Feature-Oriented Programming

In this chapter, we present code metrics for feature-oriented programming. This approach allows an independent development of a basic implementation and feature-specific extensions. Feature-oriented programming serves as a modularization technique of feature related code [ABKS13]. We do not differentiate between different feature-oriented approaches. Here, we also list metrics developed for aspect-oriented programming. Even if aspectization was originally developed from a different programming perspective (reduction of scattering while realizing crosscutting concerns), it may also be used to develop feature-oriented product lines [ABKS13].

5.1 Modifiers

In this section we present modifiers of metrics for feature-oriented code.

5.1.1 Measures per Feature / System

Feature-oriented programming separates the basic implementation from feature-specific extensions. This simplifies the isolated inspection of code belonging to the same feature. Abilio et al. make use of this characteristic and collect different values for single features in isolation as well as for all elements of the whole product line platform [AVFC16].

5.2 Size Metrics

In this section we present size metrics. Most, but not all, are defined on product line platform level.

5.2.1 Number of Constants (NoCt)

This metric measures the number of constants (classes and interfaces) belonging to the basic implementation, i.e., elements which may be refined by feature-specific extensions. The authors combine this metric with the modifier of Section 5.1.1 to define two metrics:

$\mathbf{NoCt_s}$ measures the total number of all constants for the complete system s [AVFC16]. In an earlier publication, this metric was also called "Total Number of Constants (TNCt)" [APFC15].

$\mathbf{NoCt_f}$ measures the number of constants, which belong to a single feature f [AVFC16].

5.2.2 Number of Refinements (NoR)

This metric is the counterpart of the NoCt-metric of Section 5.2.1 as it measures the number of refinements defined to extend constants of the basic implementation. Again, the authors combine this metric with the modifier of Section 5.1.1 to define two metrics:

$\mathbf{NoR_s}$ measures the total number of all refinements for the complete system s [AVFC16]. In an earlier publication, this metric was also called "Total Number of Refinements (TNR)" [APFC15].

$\mathbf{NoR_f}$ measures the number of refinements, which belong to a single feature f [AVFC16].

5.2.3 Number of Components (NoC)

This metric measures the number of components (constants and their refinements) belonging to the same feature. The authors combine this metric with the modifier of Section 5.1.1 to define two metrics:

$\mathbf{NoC_s}$ measures the total number of all components for the complete system s [AVFC16].

$\mathbf{NoC_f}$ measures the number of components, which belong to a single feature f [AVFC16].

5.2.4 Number of Refined Constants (NRC)

This metric measures the number of constants that have refinements [AVFC16]. In [APFC15], this metric was named as "Total Number of Refined Constants (TNRC)".

5.2.5 Number of Constant Refinements (NCR)

This metric measures the number of refinements that a constant has and shall indicate the complexity of the relationship between a constant and its features [VF15, VAFG15, APFC15, AVFC16].

5.2.6 Number of Method Refinements (NMR)

This metric measures the number of refinements that a method has and shall indicate the complexity of the relationship between a method and its feature-specific refinements [APFC15, AVFC16]. [AVFC16] defines also a metric to count the total number of method refinements (TNMR).

5.2.7 Number of Refined Methods (NRM)

This metric is number of methods that were refined [AVFC16]. In [APFC15], this metric was named as "Total Number of Refined Methods (TNRM)".

5.2.8 Aspect Size (AS)

The aspect size counts the number of elements belonging to the definition of an aspect (attributes, methods, pointcuts, advice, inter-type declarations) [MGvS10].

5.2.9 Number of Access Attributes (NAA)

This metric counts the number of attributes, which are used in the implementation of an join point [MGvS10].

5.2.10 Number Of Parameters (NOP)

This metric counts the number of parameters in the pointcut signature [MGvS10].

5.2.11 Number of Aspects referring to shared Join Point (NAsJP)

This metric counts the number of aspects, which a defined for a defined join point [MGvS10].

5.2.12 Set of Primitive Pointcuts (SPP)

This metric counts the number of primitive pointcuts (e.g. call, execution) used in pointcut expressions [MGvS10].

5.2.13 Number of Pointcuts with Join Points in common (NPJP)

This metric counts the number of pointcuts that have 3 or more primitive expressions in common. This metric is intended to identify duplicated / cloned pointcut expressions [MGvS10].

5.2.14 Set of the corresponding Join points of a given Pointcut (SJP)

This metric counts the number of join points picked out by a given pointcut. SJP=0 indicates a pointcut that does not match any join points [MGvS10].

5.2.15 Number of Advice referring to a Pointcut (NAdP)

This metric counts the number of advices referring to the same pointcut. NAdP=0 indicates a superfluous pointcut that is not used by any advice [MGvS10].

5.3 Classical Metrics

Much work uses classical metrics to measure feature-oriented code. However, the authors usually miss to explain if and how these metrics are adapted for feature-orientation. As a consequence, it remains unclear if for example only the relations of the basic implementation, relations of different feature implementations of the same component, or both are measured.

5.3.1 LoC

Much work uses Lines of Code to measure the size of a class, feature, or the total lines of the whole implementation [ARG+11, DRC12, GFFdAM12, ARRB13, GFFdAM14, APFC15, VAFG15, VF15]. However, the authors do not specify whether they measure only the basis implementation, feature implementations, or all feature extensions for a class together with the basis implementation.

Contrary to the aforementioned examples, [MGvS10] gives are more precise definition. The authors define for aspect-oriented code the two metrics $\mathrm{LoC}_{\mathrm{jp}}$ and $\mathrm{LoC}_{\mathrm{adv}}$, which measure the lines of code of a joint point and an advice, respectively.

5.3.2 CBO

Coupling Between Objects, sometimes also referred to as Coupling Between Components (CBC), measures the number of non-inheritance related couplings between classes [CK91]. This metric was used in several studies to measure feature-oriented code [APFC15, DRC12, VAFG15, VF15]. However, the authors do not specify whether and how relations of feature-related extensions are considered.

Macia et al. [MGvS10] use a derivation of the classical CBO-metric for aspect-oriented code and measure for aspects the number of coupled classes.

5.3.3 LCOM

Lack of COhesion in Methods measures the number of disjoint attribute sets, which are used by methods of the same class. A higher value indicates that a class probably be split into two or more sub-classes [CK91]. This metric was used by Dyer et al. [DRC12] without specifying how to integrate feature-related extensions.

Macia et al. use an adapted form, which is called "Lack of COhesion in Operations (LCOO)". This metric treats methods and advices in the same way and computes the number of disjoint attribute sets, which are used by methods and advices [MGvS10].

5.3.4 WMC

Weighted Methods per Class counts the number of methods of a class and may be combined with a complexity function [CK91]. This metric is intended to estimate the complexity of a class. Also this metric was used in several

studies to measure feature-oriented code [APFC15, VAFG15, VF15]. Again, it remains unclear if and how methods are measured, which are introduced by feature-related extensions.

5.3.5 No. of Components/Methods (VS)

This metric measures the number of components, e.g., classes, refinements, and aspects [GFFdAM12, GFFdAM14, DRC12]. This metric was also called "Vocabulary Size (VS)" [RB09, ARG+11, ARRB13]. Dyer at al. define a separate metric to measure also the number of methods [DRC12]. However, the presented work measures complete systems only and does not provide a clear definition of how to treat components and methods, which belong to a specific feature only.

5.3.6 Cyclomatic Complexity (CC)

[APFC15] apply McCabe's cyclomatic complexity metric to detect code smells in feature-oriented programs. However, the authors to not specify whether and how relations of feature-related extensions are considered.

[MGvS10] measure the cyclomatic complexity only for advices of an aspect-oriented system, i.e., they measure the complexity of feature-specific extensions only.

5.4 Tangling Metrics

5.4.1 Concern Diffusion over Lines of Code (CDLoC)

This metric measures the number of transition points for each concern through the lines of code. It measures the number of concern switches between code specific to a feature and other features or the base implementation. The higher the CDLOC, the more intermingled is the concern code within the implementation of the components; the lower the CDLOC, the more localized is the concern code [SGC+03, RB09, TKS+10, GFFdAM12, GFFdAM14].

5.5 Overview

Table 5.1 provides an overview of code metrics for composition based approaches like feature-oriented programming and aspectization. In our study

of implementation metrics for software product lines, we found only few metrics specially designed for composition based approaches. Instead, many authors use classical code metrics without a precise definition how these metrics are adapted for composition-based approaches.

		[AVFC16]	[APFC15]	[ARRB13]	[ARG+11]	[DRC12]	[GFFdAM12]	[GFFdAM14]	[MGvS10]	[RB09]	[TKS+10]	[VAFG15]	[VF15]
Size Metrics on Feature-Oriented Code	$NoCt_s$	✓	✓										
	$NoCt_f$	✓											
	NoR_s	✓	✓										
	NoR_f	✓											
	NoC_s	✓											
	NoC_f	✓											
	NRC	✓	✓										
	NCR	✓	✓									✓	✓
	NMR	✓	✓										
	NRM	✓	✓										
	AS								✓				
	NAA								✓				
	NOP								✓				
	NAsJP								✓				
	SPP								✓				
	NPJP								✓				
	SJP								✓				
	NAdP								✓				
Classical	LoC		✓	✓	✓	✓	✓	✓				✓	✓
	LoC_{jp}								✓				
	LoC_{adv}								✓				
	CBO		✓			✓			✓	✓		✓	✓
	LCOM					✓			✓				
	WMC		✓									✓	✓
	VS			✓	✓	✓	✓	✓		✓			
	CC		✓						✓				
TD	CDLoC						✓	✓		✓	✓		

Table 5.1: Use of code metrics for feature-oriented programming.

Chapter 6

Mixed Metrics

Here, we present metrics, which integrate information from variability model and code artifacts.

6.1 Features per Space

Two Papers define metrics based on the NoF-metric from Section 3.2.1 to measure the usage of features per space (variability model (Kconfig), Code artifacts, and build space (Make files)).

$\mathbf{NoF_{K\text{-}C\text{-}M}}$ Is the number of features, which are only used in the variability model [NH14].

$\mathbf{NoF_{C\text{-}M}}$ The number of features, which are also used in code artifacts [NH14]. This definition corresponds to the definition of "Number of Features with Code (NFC)" from Abilio et al. [AVFC16].

$\mathbf{NoF_{C \cap M}}$ The number of features, which are used in code artifacts and in build scripts [NH14].

$\mathbf{NoF_{M\text{-}C}}$ The number of features, which are only used in build scripts, i.e., to control the product derivation [NH14].

6.2 Percentage of Features (PoF)

Additional to the definitions of the aforementioned section, Nadi and Holt also calculate relative numbers of used variables per space [NH14]:

$\mathbf{PoF_{K\text{-}C\text{-}M}}$ Is the relative number of features, which are only used in the variability model.

PoF$_{\text{C-M}}$ The relative number or features, which are also used in code artifacts.

PoF$_{\text{C}\cap\text{M}}$ The relative number or features, which are used in code artifacts and in build scripts.

PoF$_{\text{M-C}}$ The relative number or features, which are only used in build scripts.

Chapter 7

Observations

While studying the literature, we made the following key observations regarding the definitions of metrics for software product lines:

- Many metrics have been defined from scratch instead of adapting established metrics. This applies mainly to the code metrics of Chapter 4. Here, we miss metric definitions based on classical code metrics. In particular, we did not find any measures for coupling and cohesion, fan-in and fan-out metrics, or metrics for the identification of code smells.

- Unfortenutaly, many publications do not provide a clear definition of the used metrics. This applies mainly, but not exclusively to the metrics of Section 5.3, which use classical metrics for feature-oriented programming. But also many of the other metric definitions do not provide a clear definition. For instance, we did not find a clear definition whether `#ifdef`s, `#elif`s, and `#else`s are treated as one or multiple variation points.

- For some metrics different definitions by different groups exist. For instance, the *"Cyclomatic Complexity of Feature Models"* (cf. Section 3.3.3) and the *"Number of Constraints "* (cf. Section 3.2.2.1) metrics are conceptually same, but are developed by different groups for different purposes. Another example are the multiple definitions of the *"Number of Feature Constants"* (cf. Section 4.2.2) metric.

- There are few metrics, which combine knowledge of the variability model with code artifacts. In fact, we found only metrics calculating relative values of how variability model elements are used in the different spaces. However, we miss metrics combining complexity information of variability model elements and the complexity of code artifacts.

Part III

Evaluation

Chapter 8

RQ2: Studied Correlations

In this part, we answer **RQ2** through gathering the evaluation results of the discovered papers. We observed that most metrics were applied to measure quantitative aspects, without providing any proof whether these metrics are sufficient to draw conclusions about qualitative characteristics of the studied object. Only 10 out of the 28 identified papers conducted an evaluation of the presented metrics. These papers are presented in Table 8.1. Some of these papers define new metrics and present evaluations for them, while others present an evaluation for already existing metrics. We listed metric definitions of the papers in the third column. Further, we observed that not all evaluations measure correlations between metrics and qualitative characteristics of the software. For instance, some papers analyze whether there are correlations between metrics or analyze how thresholds can be determined efficiently. We highlighted measured correlations to qualitative aspects of implementation artifacts in the second column of the table. Figure 8.1 shows the distribution of the studied correlations. Papers, which do not present an evaluation between metrics and qualitative aspects of implementation artifacts, are shown in red. Papers, which investigate at least one correlation between metrics and qualitative aspects of implementation artifacts, are shown in the lower part of the diagram and are discussed in the remainder of this section in more detail.

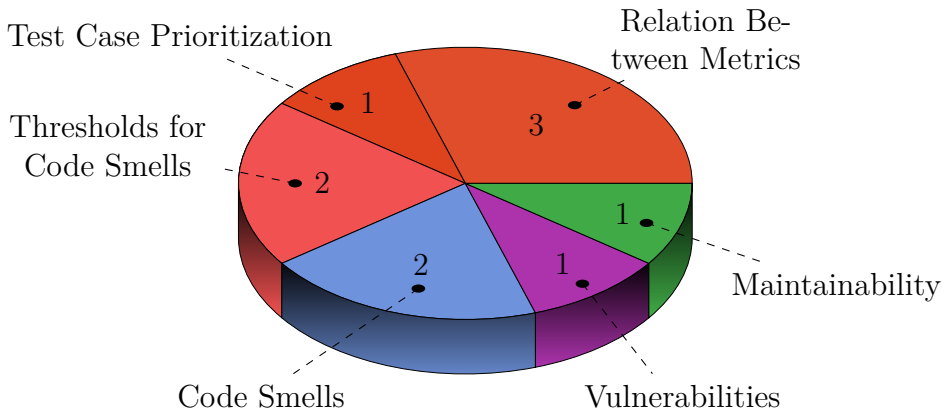

Figure 8.1: Distribution of analyzed correlations.

Paper	Analyzed Correlations	Metrics Defined by Paper	QA1	QA2	Method
[APFC15]	**Code Smells**	—	II	IV	Experiments with students
[AB11]	Correlation between metrics (self-defined, LoC, NoFC, LoF), influence of the development process (developed from scratch vs. refactored projects)	IFD, EFD, IFD_W, EFD_W, NAR, NMR	III	IV	Pearson correlation, Mann-Whitney-U test
[BG11]	**Maintainability (analyzability, changeability, understandability)**, between metrics, between sub elements of maintainability	NoF, N_{Top}, N_{Leaf}, DT, NVC, CTCR, CC, RoV, CoC, FoC	III	IV	Experiments with students
[BG14]	Between metrics	NOr, NXor, NMutex, ratio of attributed / typed variables, RCon & RDen	III	V	Pearson correlation
[FMK+16]	**Vulnerabilities**	No. int. `#ifdef`-blocks, No. int. conf. options, No. ext. conf. options, centrality, eigenvector centrality, betweenness centrality	III	V	Welch two sample t-test between vulnerable and non-vulnerable function samples
[LAL+10]	Between metrics	NoFC, LoF, SD_{VP}, TD_{VP}, ND_{avg}	III	V	Correlation
[MGvS10]	**Code Smells**	AS, NAA, NOP, SPP, LoC_{jp}, LoC_{adv}, CC	II	V	Recall & precision on 3 use cases
[SSRC14]	Test case prioritization strategies (for testing with sample products)	NoF, #constraintsfeatures, #variability points, Cross Tree-Constraints Ratio, Cyclomatic Complexity, #variability points & CC, CoC-Density, Commonality of Feature	III	IV	Average Percentage of Faults Detected (APFD)
[VAFG15]	Threshold derivation for detecting code smells	—	II	IV	Recall & precision on 33 use cases
[VF15]	Threshold derivation for detecting code smells	—	II	IV	Recall & precision on one use case

Table 8.1: Papers, which provided an evaluation (**QA1** $\geq$ II) of variability-aware implementation metrics. In column 2 shows analyzed correlations between metrics and qualitative characteristics of software artifacts in bold.

In Table 8.2 and 8.3, we present studied correlations of variability-aware metrics and qualitative aspects of implementation artifacts. The first table provides an overview of evaluations, which studied a direct correlation between a single metric and a qualitative aspect. We collected from the papers whether there exist a positive (↗) or negative (↘) correlation between the analyzed metric and the studied aspect. We use the **o** symbol to mark analysis results, which did not show any correlation or were not statistically significant.

In Table 8.3, we show evaluations, which do not analyze single metrics in isolation. This applies to work of the identification of code smells, which typically combines several metrics for the identification of smells. The presented work can be distinguished into two categories:

- Papers that analyze whether the combination of metrics can be used to identify code smells. These are the papers of the first two rows. We use the symbol $\bar{\gamma}$ if at least one of the studied combinations in which the metric was used, has shown in at least 50% of the cases to be useful to identify code smells.

- Papers that use a combination of metrics to analyze different possibilities to derive metric thresholds. This work assumes that a specific combination of metrics is useful to identify code smells and study different approaches to derive thresholds to improve their accuracy. We use the symbol • if the metric was used in at least one metric combination to evaluate a threshold derivation approach.

In summary, we can conclude that only very few contributions analyze the correlation of metrics with qualitative properties in the context of software product lines.

	Metric \ Studied Correlation	Maintainability [BG11]			Vulnerabilities
		Analyzability	Changeability	Understandability	[FMK+16]
Variability Model	N_{Top}	o	o	o	
	N_{Leaf}	↘	↘	↘	
	DT	o	o	o	
	CTCR	o	o	↘	
	RoV	o	o	↘	
	CoC	o	o	↘	
	FoC	o	↗	↗	
	NVC	↘	o	o	
	Cyclomatic Complexity of Feature Model	o	o	o	
Code Metrics	Number of Internal #ifdefs				↗
	Internal Configuration Options				↗
	External Configuration Options				↘
	Degree Centrality (outgoing)				↗
	Degree Centrality (ingoing)				o
	Eigenvector Centrality				↗
	Betweenness Centrality				o

Table 8.2: Usefulness of metrics (direct measures, ↗ = positive correlation, ↘ negative correlation, **o** = no (significant) correlation/usefulness shown).

Metric		Code Smells			
		[APFC15]	[MGvS10]	[VAFG15]	[VF15]
Metrics for FoP	AS		⍛		
	CBO	⍛	⍛	•	•
	CC	⍛	⍛		
	LCOM		⍛		
	LoC	⍛		•	•
	LoC_{jp}		⍛		
	LoC_{adv}		⍛		
	NAA		⍛		
	NAdP		**o**		
	NAsJP		⍛		
	NCR	⍛		•	•
	NMR	⍛			
	SJP		**o**		
	SPP		⍛		
	WMC	⍛		•	•

Table 8.3: Usefulness of Metrics (indirect measures, ⍛ = useful combined with other metrics , • = metric combination was successfully used to evaluate threshold derivation approach, **o** = no (significant) correlation/usefulness shown).

Chapter 9

Concluding Remarks

In this report, we presented the results of our systematic literature review to identify variability-aware implementation metrics, designed for the needs of SPLs. This literature study covered peer-reviewed work from the last decade in product line engineering. We identified variability model metrics as well as code metrics considering variation points. With our study we were able to identify existing metrics and to assess their ability to serve as a basis to draw qualitative conclusions.

Part II presents a catalog of variability-aware implementation metrics for software product lines based on the results of our systematic literature review. To the best of our knowledge this is the first catalog of this kind in scientific literature. This catalog provides a basis for future studies as other researchers do not need to re-invent the wheel. Further, we also identified unaddressed research areas in the topic of variability-aware implementation metrics for software product lines.

In Part III, we analyzed the evaluation status of the state of the art of variability-aware implementation metrics for software product lines. Comparing the collected results with the large number of metrics described in the previous chapters shows that there exist only very few contributions analyzing the qualitative power of metrics in the context of software product lines.

Acknowledgments

This work is partially supported by the ITEA3 project REVaMP^2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the BMBF.

Bibliography

[AB11] Sven Apel and Dirk Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430. ACM, 2011.

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[APFC15] Ramon Abílio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting code smells in software product lines – an exploratory study. In *Proceedings of the 2015 12th International Conference on Information Technology - New Generations*, ITNG '15, pages 433–438. IEEE Computer Society, 2015.

[ARG+11] Rodrigo Andrade, Marcio Ribeiro, Vaidas Gasiunas, Lucas Satabin, Henrique Rebelo, and Paulo Borba. Assessing idioms for implementing features with flexible binding times. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 231–240. IEEE Computer Society, 2011.

[ARRB13] Rodrigo Andrade, Henrique Rebêlo, Márcio Ribeiro, and Paulo Borba. Aspectj-based idioms for flexible feature binding. In *Proceedings of the 2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS '13, pages 59–68. IEEE Computer Society, 2013.

[AVFC16] Ramon Abilio, Gustavo Vale, Eduardo Figueiredo, and Heitor Costa. Metrics for feature-oriented programming. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, WETSoM '16, pages 36–42. ACM, 2016.

[BG11] Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19:579–612, Sep 2011.

[BG14] Thorsten Berger and Jianmei Guo. Towards system analysis with variability model metrics. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 23:1–23:8. ACM, 2014.

[BSL+13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.

[CDS86] Samuel D. Conte, Herbert E. Dunsmore, and Vincent Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 197–211. ACM, 1991.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley, 2002.

[CVF11] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 191–200. IEEE Computer Society, 2011.

[DRC12] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 143–154. ACM, 2012.

[ESKS17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. An empirical study of configuration mismatches in linux. In *21st International Systems and Software Product Lines Conference*, New York, NY, USA, 2017. ACM. Accepted.

[FMK+16] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 65–73. ACM, 2016.

[GFFdAM12] Felipe Nunes Gaia, Gabriel Coutinho Sousa Ferreira, Eduardo Figueiredo, and Marcelo de Almeida Maia. A quantitative assessment of aspectual feature modules for evolving software product lines. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 134–149. Springer-Verlag, 2012.

[GFFdAM14] Felipe Nunes Gaia, Gabriel Coutinho Sousa Ferreira, Eduardo Figueiredo, and Marcelo de Almeida Maia. A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Science of Computer Programming*, 96:230–253, Dec 2014.

[HZS+16] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21:449–482, Apr 2016.

[KBB16] Barbara Kitchenham, David Budgen, and Pearl Brereton. *Evidence-Based Software engineering and systematic reviews*. CRC Press, 2016.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A.S̃pencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, Software Engineering Institute, Carnegie-Mellon University, 1990.

[Kco17] 2017. Last visited 08.08.2017.

[Kit04] Barbara Kitchenham. Procedures for performing systematic reviews. Joint Technical Report TR/SE-0401, Software Engineering Group, Keele University, 2004.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty

preprocessor-based software product lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114. ACM, 2010.

[LT08] Roberto E. Lopez-Herrejon and Salvador Trujillo. How complex is my product line? the case for variation point metrics. In *Second International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS'08, pages 97–100, 2008.

[LWK12] Andrea Leitner, Reinhold Weiß, and Christian Kreiner. Analyzing the complexity of domain model representations. In *Proceedings of the 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, ECBS '12, pages 242–248. IEEE Computer Society, 2012.

[MBBA15] Mariem Mefteh, Nadia Bouassida, and Hanêne Ben-Abdallah. Implementation and evaluation of an approach for extracting feature models from documented uml use case diagrams. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1602–1609. ACM, 2015.

[MBBA16] Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-Abdallah. Change impact analysis for software product lines. *J. King Saud Univ. Comput. Inf. Sci.*, 28:364–380, Oct 2016.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762. ACM, 2009.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[MGvS10] Isela Macia, Alessandro Garcia, and Arndt von Staa. Defining and applying detection strategies for aspect-oriented code smells. In *Proceedings of the 2010 Brazilian Symposium on Software Engineering*, SBES '10, pages 60–69. IEEE Computer Society, 2010.

[MR11] Stefan Mann and Georg Rock. Control variant-rich models by variability measures. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 29–38. ACM, 2011.

[New10] Mark Newman. *Networks: an introduction*. Oxford University Press, 2010.

[NH14] Sarah Nadi and Ric Holt. The linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26:730–746, Aug 2014.

[PNX+11] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. A study of non-boolean constraints in variability models of an embedded operating system. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 2:1–2:8. ACM, 2011.

[RB09] Márcio Ribeiro and Paulo Borba. Improving guidance when restructuring variabilities in software product lines. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 79–88. IEEE Computer Society, 2009.

[SGC+03] Cláudio Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt Von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of Brazilian symposium on software engineering*, pages 19–34, 2003.

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. *VaMoS '10*, 10:45–51, 2010.

[SSRC14] Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 41–50. IEEE Computer Society, 2014.

[TKS+10] Mário Torres, Uirá Kulesza, Matheus Sousa, Thais Batista, Leopoldo Teixeira, Paulo Borba, Elder Cirilo, Carlos Lucena, Rosana Braga, and Paulo Masiero. Assessment of product derivation tools in the evolution of software product lines: An empirical study. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 10–17. ACM, 2010.

[VAFG15] Gustavo Vale, Danyllo Albuquerque, Eduardo Figueiredo, and Alessandro Garcia. Defining metric thresholds for software product lines: A comparative study. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 176–185. ACM, 2015.

[VF15] Gustavo Andrade Do Vale and Eduardo Magno Lages Figueiredo. A method to derive metric thresholds for software product lines. In *Proceedings of the 2015 29th Brazilian Symposium on Software Engineering*, SBES '15, pages 110–119. IEEE Computer Society, 2015.

[VFAC14] Gustavo Vale, Eduardo Figueiredo, Ramon Abílio, and Heitor Costa. Bad smells in software product lines: A systematic review. In *Proceedings of the 2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS '14, pages 84–94. IEEE Computer Society, 2014.

[ZB12] Bo Zhang and Martin Becker. Code-based variability model extraction for software product line improvement. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 91–98. ACM, 2012.

[ZBP+13] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability evolution and erosion in industrial product lines: A case study. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 168–177. ACM, 2013.

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *International Conference on Formal Engineering Methods*, pages 115–130. Springer, 2004.

Appendix A

Search Parameter

Here, we present the detailed search parameters of our survey, which we applied to ScienceDirect (Section A.1) and ACM (Section A.2). Due to the syntactic differences in defining advanced search strings for the different search engines, we adapted these strings accordingly.

A.1 ScienceDirect

Below we show the detailed search parameters for ScienceDirect:

- Expert search
- ✓ at `Journals`
- ✓ at `Books`
- ✓ at `All`
- `Selected sciences:` Computer Science

```
tak(("software product line" OR "software product-line"
   OR "software product family" OR "software product-
   family" OR "software family based" OR "software family
   -based" OR "software variability" OR "software mass
   customization" OR "software mass customization
   production lines" OR "software-intensive system")
AND ("metric" OR "metrics" OR "quantitative analysis" OR
   "quantitative characteristics" OR "complexity of
   variability" OR "variability complexity"))
```

Listing A.1: Search string for ScienceDirect

A.2 ACM

Below we show the detailed search parameters for ACM. A preliminary search revealed a bug in the search engine of ACM: Searching the product line keywords in *"Any field"* (default, if nothing is specified) also finds papers including the categories and subject descriptor *"D.2.8 [Software Engineering]"*, even if the paper does not mention software product lines at all. For this reason, we limited the search of the product line terms for the title, abstract, and the author defined keywords and excluded the categories and subject descriptor. This bug did not occur for the metrics terms, for this reason we kept for them the any field search parameter.

Parameters:

- Advanced Search
- `Select items from`: "The ACM Guide to Computing Literature"

```
(acmdlTitle:("software product line" "software product-
    line" "software product family" "software product-
    family" "software family based" "software family-based
    " "software variability" "software mass customization"
     "software mass customization production lines" "
    software-intensive system")
OR keywords.author.keyword:("software product line" "
    software product-line" "software product family" "
    software product-family" "software family based" "
    software family-based" "software variability" "
    software mass customization" "software mass
    customization production lines" "software-intensive
    system")
OR recordAbstract:("software product line" "software
    product-line" "software product family" "software
    product-family" "software family based" "software
    family-based" "software variability" "software mass
    customization" "software mass customization production
     lines" "software-intensive system")
) AND ("metric" "metrics" "quantitative analysis" "
    quantitative characteristics" "complexity of
    variability" "variability complexity")
```

Listing A.2: Search string for ACM