A Source Level Empirical Study of Features and Their Interactions in Variable Software

Stefan Fischer, Lukas Linsbauer, Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria
Email: Stefan.Fischer,Lukas.Linsbauer,Alexander.Egyed@jku.at

Roberto E. Lopez-Herrejon
Dept. Software Engineering and IT
École de technologie supérieure, Canada
Email: roberto.lopez@etsmtl.ca

Abstract—**Robust and effective support for the detection and management of features and their interactions is crucial for many software development tasks but has proven to be an elusive goal despite the extensive research and practice on the subject. Providing the required support becomes even more challenging with variable software whereby multiple variants of a system and their features must be collectively considered. An important premise to provide better support for feature interactions in variable systems is the need of a deeper understanding on how features interact at different levels starting from the source level. In this context, recent work has looked at feature interactions from different angles and for different purposes, for instance for developing performance models, extracting interfaces for maintenance or describing feature evolution patterns. However, there is a gap in understanding how features interact in fact at the source level in contrast with how features ought to interact according to variability models that describe the valid combinations of features in variable software systems. In this paper we perform an empirical study to explore this gap. We use seven case studies, implemented in Java and C, totalling over nine million LoC, and analysed over seven thousand feature interactions. Our study revealed important inconsistencies between how feature interactions occur at source level and how they are modeled, and corroborated that the majority of source level interactions involve less than three features. We discuss the implications of our findings and avenues for further research.**

I. Introduction

Variable software stems primarily from the adoption of generator-based techniques (e.g. [26], [28]), full-blown *Software Product Line (SPL)* approaches (e.g. [14]), advanced modularization paradigms (e.g. [9]), or ad hoc reuse practices collectively called *clone-and-own* [20], [39]. Typical variable software systems are formed with a large number of distinct variants, sometimes in the order of myriads [12]. What distinguishes a variant is the set of *features* – increments in program functionality [50] – that each variant provides.

The effective development of variable software poses unique challenges to software engineers not only because the number of variants and their desired feature combinations should be collectively considered, but also because their feature interactions must be properly detected, analyzed, and managed. Broadly speaking, a feature interaction occurs when the behaviour of one feature changes depending on the presence or absence of another feature or set of features [6]. Feature interactions manifest in a wide range of levels; for example, as source level artifacts, as unexpected executions, or as changes in non-functional properties. Research in feature interactions has a long standing history, and the interest in the subject has been rekindled in light of recent research developments in the context of variable software [5]. Despite the extensive body of research attested by the large number of publications (e.g. [5], [8], [7], [45], [31]), there are many open and pressing challenges [5]. Salient among them is the limited understanding on the source level structure of features and how they interact in real-world systems [5], [6], [11]. A more in-depth understanding will have far-reaching implications for both researchers and practitioners. It could lead to better tooling support for the overall management of feature interactions in variable software from their detection and analysis all the way up to exploiting feature interaction knowledge, for instance, to flexibly and automatically generate test code with adequate coverage. Emerging results already corroborate the potential benefits of this endeavour (e.g. [45], [46], [44], [42], [13]).

In this paper we pick up on this challenge and perform an empirical study [40], [25], [49] that focuses on the structure of the source level artifacts that implement variable systems. The main goal of our study is exploring how features interact *in fact* at the *code level* in contrast with how features *ought to* interact according to *variability models* – that describe the valid combinations of features in variable software systems [16]. We are interested in answering questions such as: How many features actually interact at source level?, What level of granularity (e.g. entire classes or statements) do feature interactions have?, What proportion of the source code corresponds to feature interactions?, and How do features interact according to variability models?. To address these questions we employed seven case studies totalling over nine million LoC, implemented in Java and C, and analysed over seven thousand feature interactions using six software metrics. Our study revealed important inconsistencies between how feature interactions occur at source level and how they are modeled, and that the majority of source level interactions involve less than three features. We present the implications of our findings and describe open avenues for further research.

II. Background

In this section we provide the background and basic terminology required for our empirical study along with an illustrative running example.

Artifacts and Granularity. Our study considers source code artifacts written in languages Java and C. We consider artifacts as Abstract Syntax Tree (AST) nodes obtained with parsers from the Eclipse Java Development Tools (JDT) and C Development Tools (CDT). *Henceforth, every AST node is considered an artifact*. From the AST perspective, we regard artifact granularity as a depth measure from the root of the AST. The granularity levels and what artifacts they refer to are shown in Table I for Java and C respectively.

Depth	Java	C
0	files	files
1	classes, interfaces	functions, globals, ...
2	fields, methods, ...	statements
3+	statements	

TABLE I: Artifact granularity levels, coarse (top) and fine (bottom)

Features and Feature Interactions. Within the realm of variable software there are many definitions or interpretations for the concept of features [11]. Because the focus of our study is on source level artifacts, we use Zave's definition as our running definition and regard features as increments in program functionality [50]. Hence, for our study, features are source code artifacts that implement a given program functionality. Similarly, there are also multiple conceptions and interpretations of the concept of feature interactions [5]. We provide our working definition, based on Apel et al.'s classification [6], as follows:

Definition 1: A *structural feature interaction* manifests at source level whenever source level artifacts are included in a software product because of a combination of selected and unselected features of a software variant.

Henceforth and unless otherwise stated, whenever we say feature interaction we mean structural feature interaction.

Base and Derivative Modules. In order to describe and analyze the structure of source level artifacts and how they relate to features and their interactions we now introduce the notion of *modules* to label the different relationships that can be identified in the implementing artifacts. This notation and terminology is further explained in [30], [20], [21]

Definition 2: A *module* is a set of signed (positive or negative) features. Positive features denote features selected in the module whereas negative features represent unselected (i.e. not present) ones.

We distinguish two kinds of modules defined as follows.

Definition 3: A *base module* labels artifacts that implement a given feature without any feature interactions, that is, it consists of exactly one positive feature and no negative features. We refer to them with the feature's name written in lowercase.

Definition 4: A *derivative module* $\delta^{\mathrm{n}}(\mathtt{c}_0, \mathtt{c}_1, ..., \mathtt{c}_{\mathrm{n}}) = \{c_0, c_1, ..., c_n\}$ labels artifacts that implement feature interactions, where c_i is `F` (if feature `F` is positive) or $\neg$`F` (if negative), and n is the order of the derivative. A derivative module contains at least one positive feature and any number of negative features.

A derivative module of order n represents the interaction of $n+1$ features. The order of a module denotes how many features interact in that module. Base modules have order 0, because they label a single feature and hence do no interact with any other features. *Henceforth, when we say that a module has order 0 we are speaking of a base module*.

Running Example. We now illustrate these definitions on the example of a drawing application. This application has several features that can be enabled or disabled to form variants: feature `BASE` represents the basic GUI framework that all variants of the drawing application have in common, features `LINE` and `RECT` are responsible for the functionality to draw lines and rectangles respectively, and feature `COLOR` represents the ability to use colors to draw.

Figure 1 shows a preprocessor annotated code snippet for generating the variants. The preprocessor annotations (highlighted in blue) mark which code parts are responsible for implementing which features. For example, the field definition `List<Line> lines` shown in Lines 2-4 will be included in class `Canvas` of *all* the variants that include feature `LINE`, independent of any other features being present. Hence this field definition must be part of the base module `line`.

Consider now the code in Lines 16 and 17. For this piece of code to be included in class `Line` of a variant the conditions in Lines 13 and 15 must hold. This means that both features `LINE` and `COLOR` must be selected by such variants, hence these two lines of code belong to derivative module $\delta^1(\mathtt{line}, \mathtt{color})$ because they represent feature interaction between features `LINE` and `COLOR`. Similarly, if in a variant the feature `LINE` is selected but the feature `COLOR` is not, then the Lines 19 and 20 will be included in class `Line`. Hence, these two lines belong to derivative module $\delta^1(\mathtt{line}, \neg\mathtt{color})$ because they represent the feature interaction of feature `LINE` being selected and feature `COLOR` not being selected. For a detailed explanation of the rationale behind negative features please refer to [30].

Variability Models (VMs). An important part of variable software is to specify the set of different feature combinations or variants that it comprises, that is the role of *variability models* for which there are several alternatives (e.g. [16], [12]). One of the most commonly used are feature models that are hierarchical tree-like structures where nodes represent features and edges denote different types of variability relations [26]. Variability models can be formally described, for instance using propositional logic, which enables formal reasoning about their properties. For instance, counting the number of feature combinations or verifying if a variant with a certain partial feature combination exists. For further details please refer to [10].

III. Study Setup

This section describes how our empirical study was set up. We followed the *Goal-Question-Metric (GQM)* approach to describe the underlying goal of our study, the research questions that we considered and the metrics used for addressing them [49]. We present the variable systems that constitute

```
class Canvas {
  #ifdef $LINE
  List<Line> lines;  //  line
  #end
  #ifdef $RECT
  List<Rect> rects;   // rect
  #end
  #ifdef $COLOR
  void setColor(String c) {...} // color
  #end
  ...
}
#ifdef $LINE
class Line {
  #ifdef $COLOR
  // derivative δ^1(line,color)
  Line(Color c, Point start) {...}
  #else
  // derivative δ^1(line,¬color)
  Line(Point start) {...}
  #end
  ...
}
#end
#ifdef $RECT
class Rect {
  #ifdef $COLOR
  // derivative δ^1(rect,color)
  Rect(Color c, int x, int y) {...}
  #else
  // derivative δ^1(rect,¬color)
  Rect(int x, int y) {...}
  #end
  ...
}
#end
```

Fig. 1: Code example of preprocessor annotations

the corpus of our study and describe the process employed to collect the data. The results and analysis are presented in Section IV.

A. Goal and Research Questions

The driving goal of our work is defined as follows:

Goal: Characterize how features and feature interactions are realized at the source level and contrast their realization with how they are denoted by the respective variability model.

Our work focuses on answering three basic and fundamental questions regarding features and their interactions at the source level.

RQ1. How many features interact at source code level? *Rationale:* The number of features that participate in an interaction is the first reflection of its complexity. We argue that as the number of interacting features increases so would the complexity of detecting and managing the interactions.

RQ2. How are features and their interactions realized by source level artifacts? *Rationale:* Studying how features and their interactions are implemented in the source code by different language elements (e.g. complete classes, methods, blocks or statements) can provide insights on the complexity that, for example, analysis tools would have to cope with and the challenges for program comprehension that users would have to face.

RQ3. How many features interact according to the variability models? *Rationale:* Recall that variability models denote the feature combinations, and hence feature interactions, that system variants can have. In contrast to RQ1 where we look at the *actual* interactions based on the source code, here we are concerned with the *modeled* feature interactions as denoted by variability models. The information gathered for addressing this question could shed light on the adequacy of techniques or approaches that blindly rely on variability models without considering how features are actually realized in the source code.

B. Metrics

Next we concisely describe the metrics we employed for our study grouped according to their focus. In stark difference with other studies summarized in Section V, our set of metrics aims at providing a more in depth view of feature interactions, how they are actually realized in the source code and contrast how they are modeled by their corresponding variability models.

System Size Metrics. These two metrics focus on basic properties of variable systems.

- **Number of Features** F. The number of features in a variable system.
- **Number of Lines of Code** LoC. The number of lines of source code in a variable system, not including comments and empty lines.

Artifact Metrics. The following two metrics focus on the source code artifacts that implement a variable system. Recall that we represent artifacts as AST nodes, so these metrics consider the AST nodes that implement modules.

- **Number of Artifacts per Order** A_o. The number of AST nodes in a system that implement modules with order o. We denote the aggregated artifacts of all orders as A.
- **Number of Artifacts per Granularity Level** $AG_{d,o}$. The number of AST nodes in a system at granularity level d, i.e. depth in the AST, that implement modules with order o.

Module Metrics. The following metrics focus on modules, base and derivatives, that are actually found in the source code and those described by variability models.

- **Number of Modules per Order** M_o. The number of modules of order o identified in the source code of variable systems. We denote the aggregated modules of all orders as M.
- **Number of Modules per Order in Variability Model** MVM_o. The number of modules with order o denoted by the variability model of a system. We denote the aggregated modules of all orders as MVM.

Following the GQM approach, Table II summarizes the alignment of our three research questions with our metrics. RQ1 focuses on how many features interact which relies on the order of modules measured by M_o. RQ2 focuses on the realization of features and their interactions by source level artifacts which is measured by metrics A_o and $AG_{d,o}$. Finally, the focus of RQ3 is features and their interactions from the variability model perspective which is measured by MVM_o. Note that the systems size metrics, F and LoC, are used for analysis of correlation with the other metrics.

Research Question	**Metrics**
RQ1	M_o
RQ2	A_o, $AG_{d,o}$
RQ3	MVM_o

TABLE II: GQM alignment of questions and metrics

C. Study Corpus

Table III lists the systems that form the corpus of our empirical study and the values for some of our metrics. Our focus was to study variable systems implemented in different languages. Hence, our corpus includes systems written in Java and C, the most common languages in variable systems.

We looked for case studies that have been used, by us and others, for variable software research and practice. For our selection we had two basic requirements: *i)* publicly available and complete source code, and *ii)* publicly available variability model. Despite the extensive research in variable software and feature interactions, summarized in the related work (see Section V), we found only few systems that met both requirements. This was so because the great majority of existing research either focuses on the extraction of variability models from different sources (e.g. configuration files) or focuses on specialized analysis of variable source code artifacts. This split is in part due to the fact that the research on these topics is fundamentally carried out by independent groups. A particularly thorny issue was that the version number of available source code was not always the same version number as the variability model. In such cases we fetched from the corresponding repository the source code with the same version number of the available variability model.

Let us briefly describe the selected systems. ArgoUML is an open source project that has been made into a product line of UML modeling tools [15]. VOD is a product line for video-on-demand streaming applications [2], [33]. The axTLS embedded SSL project is a highly configurable client/server TLSv1 SSL library designed for platforms with small memory requirements. BusyBox is an open source C project that combines tiny versions of many common UNIX utilities into a single small executable providing an environment for any small or embedded system. Linux is a version of the operating system kernel source code. OpenSSL is an open source project for providing commercial-grade SSL and TLS support as well as a full-strength general purpose cryptography library. uClibc is an open source C library for developing embedded Linux systems. The data of our study corpus is available at http://www.jku.at/isse/content/e139529/e126342/e219248/e280716.

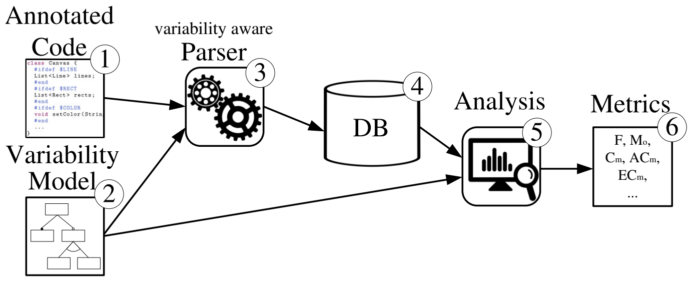


Fig. 2: Process for analyzing variable software systems

D. Process Overview

In this section we describe the process followed to gather the metrics' data for our study which is depicted in Figure 2. For every variable system we collected its: *i)* list of features, *ii)* entire source code, and *iii)* variability model.

The input to our process is the annotated code (1), similar to that shown in Figure 1, and a variability model (2). Both inputs are received by the variability aware parser (3) that we developed which first harvests the modules and the implementation artifacts, and stores them in a database (4). This variability aware parser creates an AST from the source code and assigns the artifacts to modules according to the annotations. There were two special cases to consider in this parser. The first special case was the source code that was not annotated at all. We lumped this source code artifacts into an additional feature which we called `CORE` and labeled them with corresponding base module `core`. Hence this feature represents the artifacts that are selected for every possible system variant. Furthermore, we added module `core` to the modules we found that had negative features only, which were hence violating our Definition 4. The second special case was for parsing the BusyBox, Linux and uClibc systems. The process for these systems was complicated by the fact that the selection of which files to process was handled by complex make and configuration files based on the selection of features. To address this issue we relied on the information provided by Kästner et al. who tag source code files with *presence conditions* which are propositional formulas that determine when a file is processed or not [35]. We considered the presence conditions when computing the modules based on the annotations.

The database (4) that we used is based on our previous framework for variable software reuse called ECCO [30], [20], [21]. The database contains the following output information for each system: *i)* the set of modules M, *ii)* the implementation artifacts A, *iii)* a mapping between modules and implementation artifacts establishing how each module is actually implemented in the source code, and *iv)* a graph with the structural dependencies among artifacts (e.g. field references). The analysis (5) uses the database and the variability model to compute all the metrics (6) for each system. In this regard we should remark a few points. For computing the lines of

System	Lang	F	V	C	LoC	A	M	MVM^*	Source
ArgoUML	Java	11	11	13	180.4K	1456.9K	25	441	http://argouml-spl.tigris.org/
VOD	Java	11	11	16	4.6K	35.2K	32	408	Based on [2], [33]
axTLS 1.2.7	C	90	684	2155	28K	141.4K	65	6404378	http://axtls.sourceforge.net//
BusyBox 1.18.5	C	651	6796	17836	333.3K	1417.8K	395	634738	http://www.busybox.net/
Linux 2.6.33.3	C	11004	31713	293826	8176.3K	5.6E7	5608	≈2.13E19	https://www.kernel.org/
OpenSSL 1.0.1c	C	589	589	17	381.9K	1851.3K	962	≈1.32E24	https://www.openssl.org/
uClibc 0.9.33.2	C	137	6408	35632	315.9K	2123.8K	259	≈1.18E10	http://www.uclibc.org/

Lang: Implementation Language, F: Number of Features, V: Number of Variables in the CNF Variability Model, C: Number of Clauses in the CNF Variability Model, LoC: Number of Lines of Code, A: Number of Distinct Artifacts, M: Number of Modules, MVM^*: Number of Modules in the Variability Model up to the highest order found in the implementation

TABLE III: Variable Systems Overview

code (LoC) we used the *Count Lines of Code* tool [1], which distinguishes between blank lines, comments and actual code. In Table III we report the number of actual lines of code. For the computation of the metric MVM_o we used the SAT solver *Sat4J* [3]. Checking if a module is denoted by a variability model is a straightforward process that uses the standard mapping of variability models to CNF (please refer to [10]) to which we add the features (positive and negative) labeled by the module as additional CNF clauses. In Table III the column MVM shows the number of modules in the variability model up to the order for which we actually found source level artifacts.

For OpenSSL, uClibc and Linux it was not feasible to compute all theoretically possible modules MVM in a reasonable amount of time due to the large number of features and the complexity of the constraints in their variability models [12]. For these systems we used an estimation similar to the one used by Henard et al. [22]. For this estimation we computed 1000 random modules for each order o and checked how many of these random modules are valid in the variability model. The actual number of modules in the variability model can then be estimated from the upper bound of theoretically possible modules of a variability model with F features and no constraints, which is at most $\binom{F}{o+1} * (2^{o+1} - 1)$, that is, the number of combinations of $o + 1$ features (since the order is the number of features minus one) in all F features times the possible combinations of features (negative and positive) minus one (to exclude the one module with all negative features which does not adhere to our definition of derivative modules). For instance, if 800 of the 1000 samples exist in the variability model, the estimated number of modules of order o is equal to $\frac{800}{1000} * \binom{F}{o+1} * (2^{o+1} - 1)$.

IV. Results and Analysis

In this section we concisely present the results obtained for each research question, their collective analyses, and the threats to validity we identified in our study.

A. Research Questions Results

RQ1. How many features interact at source code level?: To answer this question Figure 3a shows the number of modules per order M_o and Figure 3b shows the percentage of modules per order in relation to the total number of modules in each system. The modules over all orders add up to M and 100% respectively. The graphs show a similar pattern for all systems. Most of the modules that exist are of order 1, i.e. modules that implement interactions of two features, followed by modules of order 0, i.e. base modules. The only exception is the system BusyBox where there exist almost solely base modules because almost all of the presence conditions consist of single features and are placed in the GNU-build system at a coarse level (i.e. entire files) and almost no annotations within files exist.

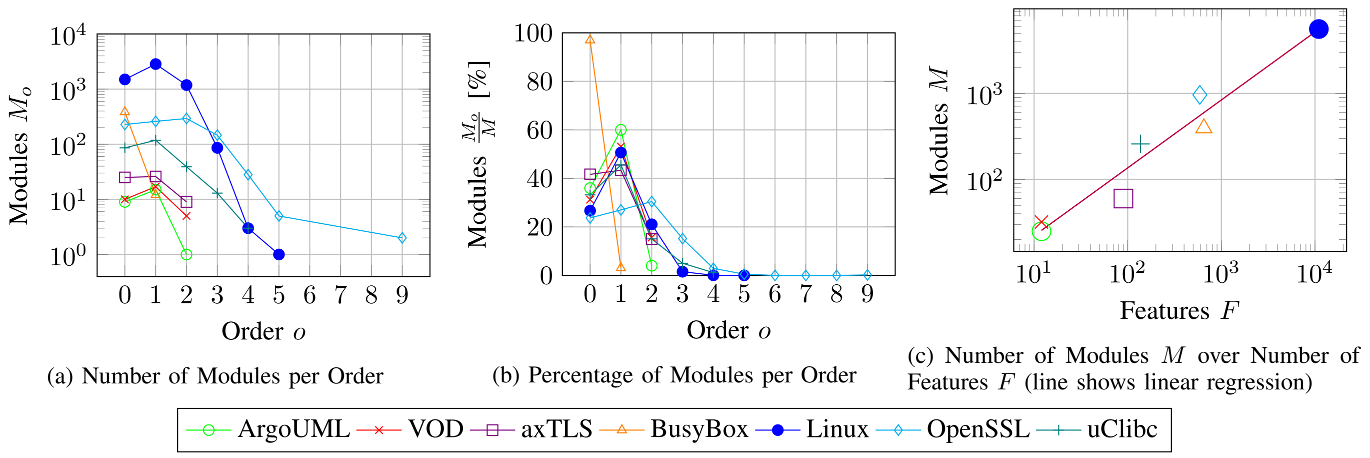
We analyzed the relations between number of modules M with number of features F and LoC. We found, as expected, strong correlations respectively with values 0.993 and 0.992 using the *Pearson Product-Moment Correlation*. Furthermore, as shown in Figure 3c, our results indicate that the number of modules appears to only increase linearly with the number of features even though every additional feature could potentially interact with every combination of other features. This finding is an indication that the number of interacting features is bounded.

RQ2. How are features and their interactions realized by source code artifacts?: To address this question we first considered how artifacts distribute per order. We investigate this issue because even if higher order modules are rare they could still involve a large part of the source code artifacts. Figure 4a shows the results, which uses the metric of number of artifacts per order A_o plotted as percentage of the total number of artifacts A. This figure shows that the majority of artifacts implement modules of order 0 and only very few implement modules of an order higher than 3.

Secondly, we analyze the relation between the number of features F and the number of artifacts as shown in Figure 4b. While the number of artifacts implementing an individual feature is very specific to each feature, the total number of artifacts tends to increase with the number of features of the system.

Thirdly, since LoC is a more common metric than number of artifacts, we also studied their correlation. As could have been expected, these two metrics have a strong correlation (see Figure 4c), with a *Pearson Product-Moment Correlation Coefficient* of 0.9998.

Fourthly, we analyzed the number of artifacts per module order at different granularity levels. We were interested in knowing whether modules of different orders are implemented by artifacts of different granularities. Figure 5 shows one



(a) Number of Modules per Order

(b) Percentage of Modules per Order

(c) Number of Modules M over Number of Features F (line shows linear regression)

Fig. 3: RQ1. How many features interact.

bubble plot per case study system. The bubble size corresponds to the number of artifacts per granularity and order $AG_{d,o}$ which we plot scaled by a factor f for sake of better visualization. In all systems most artifacts are around depth 5 (corresponding to statements) and implement modules of order 0. Only systems Linux and uClibc also have bigger portions of artifacts implementing modules of order 1 and even 2. While there is a difference in the number of artifacts for different orders, we did not find a relevant difference in their depth. Summarizing, higher order modules and lower level modules are similarly spread along the granularity levels. However, lower level modules have a larger number.

RQ3. How many features interact according to the variability models?: To address this question we relate per order the existing modules M_o, those extracted from the systems' artifacts, with the number of modules as denoted by the variability models MVM_o. The result is shown in Figure 6a. We found that this percentage decreases with order and that modules of order 0 (i.e. base modules) are the ones with the highest percentage of existing modules across all systems. The highest value being 83% for VOD and the lowest being Linux with 18%. Notice that we would expect every base module to exist; however, that is not the case. This is because some features are only implemented in first order (or generally higher order) modules, i.e. they are not implemented by any base module. Such features require the presence of other features and are encapsulated within those features, i.e. the annotations of those features in the source code are always wrapped inside annotations of the required features (i.e. the annotations are nested). For example, the annotation of feature `COLOR` of our running example in Figure 1 at Line 15 is wrapped inside the annotation of feature `LINE` at Line 13.

B. Analysis

In this section we concisely analyze and summarize the most salient findings of our study.

Pre-eminence of Low Order Modules. Our study found that across all systems most modules are of orders 0 to 2, not only in number but also considering the proportion of implementation artifacts where modules of order 0, i.e. base modules, contain the bulk of the code.

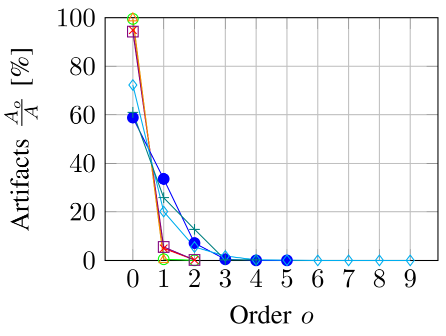
Pre-eminence of Fine Granularity Artifacts. Our study revealed that the majority of source code artifacts are at the level of statements or below. Furthermore, the distribution of granularity is independent of module order, in other words, high order modules as well as low order modules favour artifacts of fine granularity.

Larger Systems Do Not Have More Complex Feature Interactions. We found that systems with more features tend to have more modules and artifacts, but not necessarily have interactions of higher order. Interesting is also that the number of modules seems to only increase linearly with the number of features even though every feature could potentially interact with every combination of all other features.

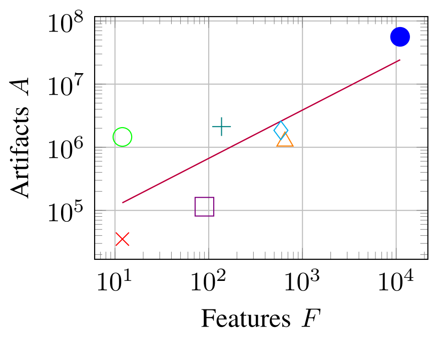
Modules Denoted by Variability Models Outnumber Those with Implementation Artifacts. Our study found that for orders higher than 0 only a very small subset of modules denoted by the variability model actually have implementation artifacts. This is in part due to the extremely large number of possible modules denoted in the variability models, as estimated for our larger systems, but it is also an observation made for our small systems.

Existence of Dead Modules. While analyzing the systems in more detail we found that in Linux and OpenSSL, some modules that exist in the source code should not exist according to the variability model. In other words, the implementations of these two systems contain modules that are not included in any of the variants that can be configured according to their variability models. We therefore refer to these modules as *dead modules*. The number of dead modules for these two systems is shown in Figure 6b and the percentage of dead modules over the total number of modules is depicted along module order in Figure 6c.

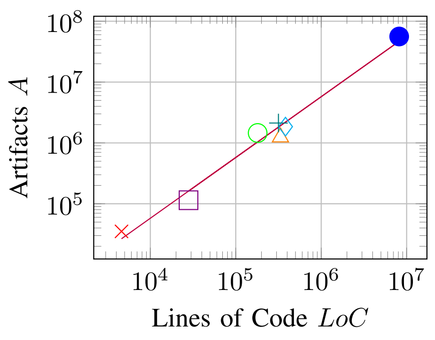
The last two findings are clear indications of inconsistencies between how a variable system is modeled (i.e. the variability model) and how it is actually implemented. Whether these inconsistencies are intentional by the developers or an unwanted



(a) Percentage of Artifacts per Order



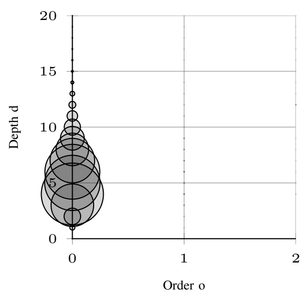
(b) Number of Artifacts A over Number of Features F (line shows linear regression)



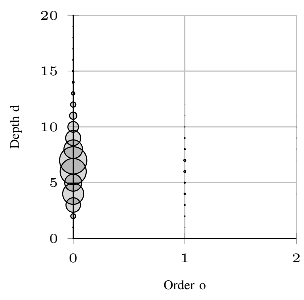
(c) Number of Artifacts A over LoC (line shows linear regression)

ArgoUML VOD axTLS BusyBox Linux OpenSSL uClibc

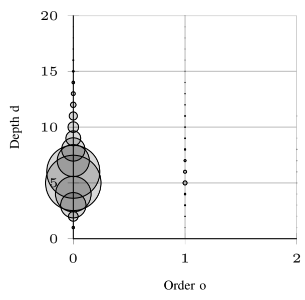
Fig. 4: RQ2. Artifacts, their order, and their relation with F and LoC



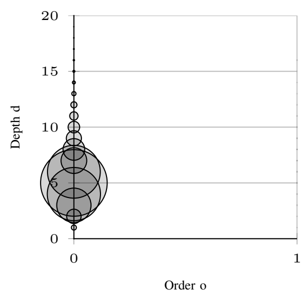
(a) ArgoUML ($f = 0.00005$)



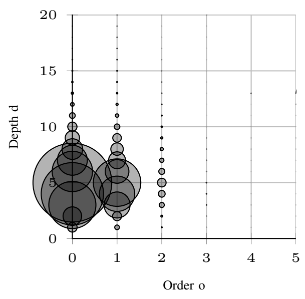
(b) VOD ($f = 0.001$)



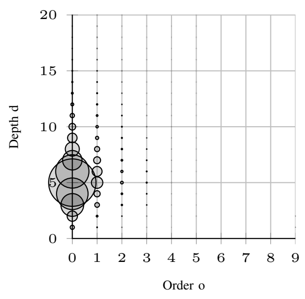
(c) axTLS ($f = 0.0005$)



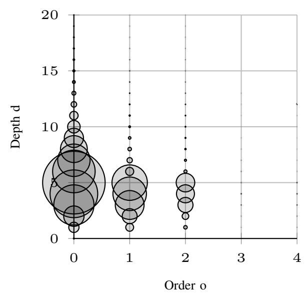
(d) BusyBox ($f = 0.00005$)



(e) Linux ($f = 0.000002$)



(f) OpenSSL ($f = 0.00003$)



(g) uClibc ($f = 0.00005$)

Fig. 5: RQ2. Number of Artifacts $AG_{d,o}$ (scaled by factor f) per Granularity (i.e. Depth in AST) and Module Order

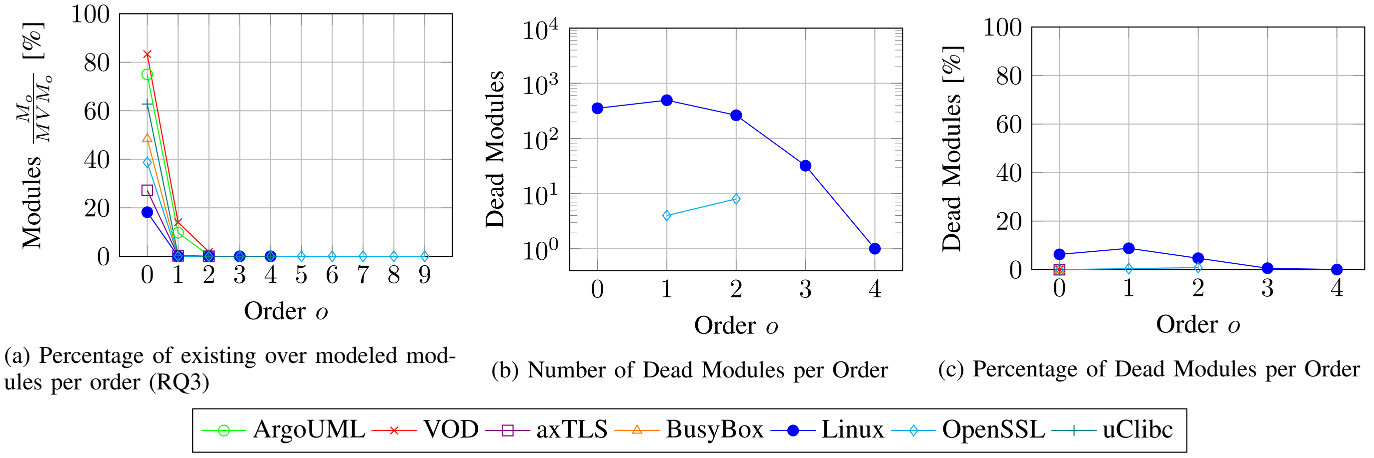
bug is an open question outside the scope of this paper.

C. Threats to Validity

In this subsection we describe the threats to validity that we identified in our work, based on the guidelines presented in [49], and how we addressed them.

The first threat was the selection of the variable systems considered in our study. We addressed this threat by including systems that have been extensively studied by ourselves and others, see Section V, and hence are deemed representative of variable software. We included all systems for which complete source code artifacts and corresponding variability models were publicly available and those for which we found ways to achieve internal consistency, as we described in Subsection III-D. These systems happened to be implemented with pre-processor annotations. Certainly, other variable systems might have yielded different results. As part of our future work we plan to expand our study to include more variable systems that use other variability management approaches (e.g. feature orientation) and are implemented in other programming languages.

The second threat can stem from the elaborated process we used to gather and analyze the metrics data as described in Subsection III-D. We addressed this threat by performing multiple cross-checks and validations both on our data and process as well as those coming from third parties.



(a) Percentage of existing over modeled modules per order (RQ3)

(b) Number of Dead Modules per Order

(c) Percentage of Dead Modules per Order

Fig. 6: RQ3. How many features interact in variability model, and Dead Modules

The third threat is the choice of metrics to answer our research questions. We do acknowledge that there is an extensive body of research on software system metrics for different types of systems and with different purposes and goals. For our study, we selected the metrics that we believe best reflect the fundamental aspects of our research questions. We informed our decisions based on the work of Meneely et al. [34] and Wholin et al. [49].

The fourth threat concerns the generalization of our findings. Our study focused on structural features and structural feature interactions, the most fundamental type of interactions. As it is summarized in the related work, Section V, there is extensive research on other types of feature interactions that involve, for instance, control-flow analysis. Hence, studying whether our findings also hold for those types of interactions is outside the scope of this work, but nonetheless an avenue worth of future research.

V. Related Work

There is an extensive body of research related to our work. In this section we briefly describe those pieces of work closest to ours categorized by main topics.

Relationship between variability models and implementation artifacts. Tartler et al. describe for Linux four different types of inconsistencies between implementation artifacts and their configuration [47]. Subsequent work by Nadi et al. focuses on the inconsistencies where pre-processor conditions contain variables that are not defined in a variability model (i.e. inferred from KConfig files in their case) and how they were fixed [36]. Our findings complement their work because we looked in further detail at dead modules, i.e. code fragments whose annotations are not valid feature combinations according to their variability model, not only for Linux but also in the OpenSSL case study. Passos et al. developed a catalogue of coevolution patterns for the Linux kernel that captures the changes that occur in the variability model and how they impact the implementation artifacts [37]. Hence, their focus in on evolution patterns not in characterizing how features interact.

Variable Software Metrics. Kästner et al. raise the issue of granularity of features in SPLs [27], for which Liebig et al. propose five metrics to analyze its impact for program comprehension and refactoring [28]. Queiroz et al. analyze the distribution of three metrics on C-based preprocessor systems and propose thresholds for them [38]. Hunsen et al. performed a study that compared preprocessor-based variability in open source projects and how they relate to industrial software systems [23]. They found that the knowledge, insights, and tooling developed from open source projects is indeed transferable to industry-strength systems. In stark contrast with all these works, our work provides a focused, encompassing, and detailed perspective on the source level features and feature interactions, using a different set of metrics and taking in consideration the role and impact of the variability model.

Variable Software Analysis. In recent years, there has been an increasing interest in devising strategies to analyze SPLs. A survey and classification for *static strategies* has been proposed by Thüm et al. that considers over a hundred of research articles [48]. The static strategies their work considers are: type checking, conventional static analyses (e.g. control flow), model checking, and program verification. Salient among the surveyed works is Liebig et al.'s who propose an approach for variability-aware type checking and liveness analysis that enhances traditional static analyses and AST representations with variability knowledge stemming from annotations in C code and variability models, for example, derived from configuration files [29]. Along the same lines, Cafeo et al. propose an approach that employs a clustering algorithm to segregate members of feature interfaces that are relevant for maintenance tasks from those that are not [13]. In stark contrast with all these works our focus is on measuring source level features and their interactions. Related to this topic is the work of Dit et al. that provide a survey and taxonomy of feature location [18]. However, their work is not focused on variable software, where feature location entails considering that features can be present in different combinations to form a typically large number of different variants.

Variable Software Testing. There is an extensive and recent interest in the area of SPL testing as attested by several systematic mapping studies (e.g. [19], [17]). Salient among the identified techniques was *Combinatorial Interaction Testing (CIT)*, that when applied to SPLs advocates selecting sets of system variants, called *covering arrays*, whose combinations of features contain all possible combinations according to a variability model of a given number t of selected and unselected features. This number t is called the strength of the covering array and within the context of our study corresponds to the order of feature interactions minus one. For example, a covering array of $t = 2$ (also called pairwise) is a set of system variants whose feature combinations consider all the interactions of order 1. We performed a systematic mapping study to delve into more detail in the subject, and identified over forty different CIT approaches for SPL testing that rely on different techniques (e.g. genetic or greedy algorithms) evaluated on multiple problem domains of different characteristics [32]. Among other findings, our mapping study revealed that the large majority of approaches focuses only on computing the samples of products to test based purely on variability models (e.g. feature models) without considering how the distinct feature combinations are actually implemented, in other words, they are oblivious to how features interact when they are realized. Another related common thread in the surveyed approaches is that they do not empirically show that higher coverage strengths (i.e. $t > 3$) are more effective for fault detection to actually pay off for their typically more expensive computation. Exploiting the information obtained in our study can lead computing of more accurate covering arrays that do not test combinations of features that do not interact hence significantly reducing the overall testing effort.

VI. Implications of Findings

In this section we put forward the main implications of our empirical study based on our analysis, Subsection IV-B, and the broader context provided by the related work.

Sampling Analysis Techniques for Variable Systems. There exist several techniques that rely on sampling to select the variants subject to the analysis [48]. To the best of our knowledge, none of such strategies study or exploit any knowledge or insights similar to those derived from our empirical study. We argue that tool developers and users of those and similar approaches can benefit from our work for instance to select or adapt sampling heuristics depending on the characteristics of the variable systems, that is, by reducing their sampling space based on how feature interactions are *in fact* realized and not on how they ought to interact according to some variability model. This, we believe, is a promising avenue for further research.

CIT for SPLs. CIT approaches that focus on selecting variants based on their artifact structure can benefit from our findings. Our study empirically showed that feature interactions of more than two features are, for the most part, rare. Hence, software engineers could decide to opt for computing pairwise covering arrays when higher strengths are infeasible or computationally expensive for their concrete contexts. Early evidence seems to suggest that covering arrays of flexible strength that consider the actual structural feature interactions could prove more effective for fault detection than covering arrays of high and fixed strengths [46]. Our findings strongly suggest such possibility, which we argue is worth of further research.

Adequacy of Variability Models. Our study found a large gap between interactions as denoted by the variability models and the interactions actually observed at the code level. In the best of scenarios this gap could cause unnecessary computational effort but at worst it could render reasoning techniques infeasible or unreliable. We propose that analysis techniques that heavily rely on variability models should question and consider the adequacy of these models for their concrete domains.

VII. Conclusions and Future Work

In this paper we presented the results of our empirical study that focused on structural feature interactions. We formulated and addressed three fundamental research questions that shed light on the characteristics of such interactions. Our work revealed the existence of a significant percentage of dead modules (those that should not exist according to the respective variability model) present in two of the analyzed systems, and that the interactions at the code level are greatly outnumbered by the interactions computed from the variability model. Among other findings were that most of the interactions involved less than three features and happen with fine grain source code elements (i.e. statement level). We place our study into a comprehensive review of the related research, and highlight the implications of our findings and avenues for further research.

We now sketch the most important items of our future work. First, we plan to expand our empirical study to consider other programming languages, other variability mechanisms, and consequently more examples of publicly available variable systems. Our purpose is to further explore the impact of these two factors. Second, we plan to go beyond structural interactions to include data flow and control flow analyses. Our goal is to analyze how the additional knowledge changes, or not, the modules' boundaries at structural level. For instance, we want to find out if more features interact when considering control flow information, in other words, does control flow affect module orders? Third, we want to assess how beneficial it is to consider the knowledge of structural feature interactions when employing CIT for SPL testing. We are currently evaluating two systems in this regard [4], [41].

VIII. Acknowledgements

We thank Sven Apel, Christian Kästner and their colleagues for their technical help and advice. Stefan Fischer is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute for Software Systems Engineering. This research was partially funded by the Austrian Science Fund (FWF) projects P25289-N15, P25513-N15.

References

[1] CLOC: Count lines of code. http://cloc.sourceforge.net/. Accessed: 2015-08-20.

[2] Java mpeg player. http://peace.snu.ac.kr/dhkim/java/MPEG/. Accessed: 2015-08-20.

[3] Sat4j: the boolean satisfaction and optimization library in java. http://www.sat4j.org/. Accessed: 2015-08-20.

[4] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *ASE '14*, pages 421–432, 2014.

[5] S. Apel, J. M. Atlee, L. Baresi, and P. Zave. Feature interactions: The next generation. Technical Report 14281, Dagstuhl Seminar, July 2014.

[6] S. Apel, S. S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *FOSD '13*, pages 1–8, 2013.

[7] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 2013.

[8] D. S. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *GPCE 2011*, pages 13–22, 2011.

[9] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[10] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[11] T. Berger, D. Lettner, J. Rubin, P. Grnbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *SPLC*, pages 16–25, 2015.

[12] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE TSE*, 39(12):1611–1640, Dec 2013.

[13] B. B. P. Cafeo, C. Hunsen, A. Garcia, S. Apel, and J. Lee. Segregating feature interfaces to support software product line maintenance. In *MODULARITY*, volume TBD, pages 1–34, 2016.

[14] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns von*. Addison Wesley, 2007.

[15] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, pages 191–200, 2011.

[16] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *VaMoS*, pages 173–182, 2012.

[17] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011.

[18] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[19] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Inform. & Software Tech.*, 53(1):2–13, 2011.

[20] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *ICSME*, pages 391–400, 2014.

[21] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ecco tool: Extraction and composition for clone-and-own. In *International Conference on Software Engineering (ICSE)*, volume 2, pages 665–668, 2015. tool demonstrations track.

[22] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE TSE*, 40(7):650–670, 2014.

[23] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *EmSE*, 2015.

[24] P. Jalote, L. C. Briand, and A. van der Hoek, editors. *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014.

[25] N. J. Juzgado and A. M. Moreno. *Basics of software engineering experimentation*. Kluwer, 2001.

[26] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[27] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In Schäfer et al. [43], pages 311–320.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE*, pages 105–114, 2010.

[29] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE'13*, pages 81–91, 2013.

[30] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In T. Kishi, S. Jarzabek, and S. Gnesi, editors, *SPLC*, pages 131–140. ACM, 2013.

[31] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE-28*, pages 112–121. ACM, 2006.

[32] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In *ICST Workshops*, pages 1–10, 2015.

[33] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed. From requirements to features: An exploratory study of feature-oriented refactoring. In *SPLC*, pages 181–190, 2011.

[34] A. Meneely, B. H. Smith, and L. Williams. Validating software metrics: A spectrum of philosophies. *ToSEM*, 2012.

[35] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: static analyses and empirical results. In Jalote et al. [24], pages 140–151.

[36] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux variability anomalies: what causes them and how do they get fixed? In *MSR*, pages 111–120, 2013.

[37] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, pages 1–50, 2015.

[38] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. The shape of feature code: an analysis of twenty c-preprocessor-based systems. *Software & Systems Modeling*, pages 1–20, 2015.

[39] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: a framework and experience. In *SPLC*, pages 101–110, 2013.

[40] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012.

[41] A. B. Sánchez, S. Segura, J. A. Parejo, and A. Ruiz-Cortés. Variability testing in the wild: The drupal case study. *Software and Systems Modeling Journal*, pages 1–22, Apr 2015.

[42] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (T). In *ASE*, pages 342–352, 2015.

[43] W. Schäfer, M. B. Dwyer, and V. Gruhn, editors. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008.

[44] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *ESEC/FSE*, pages 284–294, 2015.

[45] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, pages 167–177, 2012.

[46] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. *Operating Systems Review*, 45(3):10–14, 2011.

[47] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10, 000 feature problem. In *EuroSys*, pages 47–60, 2011.

[48] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput.ing Surveys*, 47(1):6:1–6:45, 2014.

[49] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[50] P. Zave. Faq sheet on feature interaction.