Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?

Kai Ludwig
Harz University of Applied Sciences
Wernigerode, Germany
kludwig@hs-harz.de

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

ABSTRACT

The annotation-based variability of the C preprocessor (CPP) has a bad reputation regarding comprehensibility and maintainability of software systems, but is widely adopted in practice. To assess the complexity of such systems' variability, several analysis techniques and metrics have been proposed in scientific communities. While most metrics seem reasonable at first glance, they do not generalize over all possible usages of C preprocessor variability that appear in practice. Consequently, some analyses may neglect the actual complexity of variability in these systems and may not properly reflect the real situation. In this paper, we investigate two types of variation points, namely *negating* and `#else` directives, to which we refer to as *corner cases*, as they are seldom explicitly considered in research. To investigate these directives, we rely on three commonly used metrics: lines of feature code, scattering degree, and tangling degree. We (1) describe how the considered directives impact these metrics, (2) unveil the resulting differences within 19 systems, and (3) propose how to address the arising issues. The results show that the corner cases appear regularly in variable feature code and can heavily change the results obtained with established metrics. We argue that we need to refine metrics and improve variability analysis techniques to provide more precise results, but we also need to reason about the meaning of corner cases and metrics.

CCS CONCEPTS

• **Software and its engineering → Preprocessors**; **Software product lines**; *Feature interaction*; *Maintaining software.*

KEYWORDS

Software product lines; preprocessor; variability analysis; empirical study; software metrics

ACM Reference Format:
Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3336294.3336296

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336296

1 INTRODUCTION

The C preprocessor [14, 20], for which we show an example from Linux in Listing 1, is a widely used tool to implement variability in software [3, 13, 32]. To this end, conditional directives define *variation points* in the code (e.g., `#ifndef` in Listing 1 Line 58), whereby their *feature constants* [6] (i.e., `CONFIG_PM`) are used to include or exclude code for a specific feature configuration. This *conditional compilation* [11, 20] allows to implement internal (i.e., visible to developers) and external (i.e., visible to users) variability in a configurable software system [24, 32, 39].

The pros and cons of the C preprocessor are extensively discussed in academia [9, 28, 30, 32, 43, 47]. In particular, the software-product-line [3, 39] community is concerned with understanding, analyzing, and refactoring such variability. These efforts have led to a variety of tools [22, 34] and metrics [6] that allow to perform different analyses of preprocessor-based variability. These analyses are usually based on metrics of variation point characteristics, for instance, size, scattering, and tangling.

In a recent literature review, El-Sharkawy et al. [6] summarize such metrics and aim to define them more precisely. However, the results indicate that researchers do not always measure metrics in the same way throughout all studies (e.g., compare Liebig et al. [29] and Queiroz et al. [40]). We see three issues in using current metrics to analyze software variability: (1) varying definitions, (2) different ways of measurement, and (3) limited applicability. These issues threaten the results of studies on variable source code, prevent comparisons, and may mislead discussions on variability.

The first two issues are a concern of clearly defining metrics and their application. In this paper, we are concerned with the third issue and the conceptual terms of (i) *covert* and (ii) *phantom features* (cf. Section 2) that are not at all or wrongly captured by existing metric definitions (cf. Section 3.2): To what extent do variation points that are either (i) anonymous (i.e., `#else` directives, cf. Listing 1 Line 78) or (ii) not depending on a feature presence (i.e., negated feature expressions, cf. Listing 1 Line 58) affect variability metrics? Throughout this paper, we refer to such variability as *corner cases*, as we rarely found research that directly mentioned how to address such cases. For example, Sincero et al. [46] consider all types of conditional directives (including `#else`) as propositional formulas and focus on automated constraint solving. Liebig et al. [30] focus on specific usage patterns of variable code compared to feature expressions' semantics. However, neither study elaborates on the meaning of variation points for metrics and the consequent implications for software maintenance and evolution.

To answer the posed question, we first describe and motivate corner cases and their implications in terms of variability (cf. Section 2). Furthermore, we report an empirical study of 19 open-source

Listing 1: Covert (`#else`) & phantom (`#ifndef`) variability in `linux-4.10.4/arch/powerpc/platforms/pseries/power.c`.

```
58 #ifndef CONFIG_PM
59 struct kobject *power_kobj;
60
61 static struct attribute *g[] = {
62         &auto_poweron_attr.attr,
63         NULL,
64 };
65
66 static struct attribute_group attr_group = {
67         .attrs = g,
68 };
69
70 static int __init pm_init(void)
71 {
72   power_kobj = kobject_create_and_add("power", NULL);
73   if (!power_kobj)
74     return -ENOMEM;
75   return sysfs_create_group(power_kobj, &attr_group);
76 }
77 machine_core_initcall(pseries, pm_init);
78 #else
79 static int __init apo_pm_init(void)
80 {
81   return (sysfs_create_file(power_kobj, &auto_poweron_attr.attr));
82 }
83 machine_device_initcall(pseries, apo_pm_init);
84 #endif
```

systems (cf. Section 3). In this study, we measured how often our corner cases appear in practice, how many lines of feature code they comprise, and how they affect the commonly used metrics scattering degree and tangling degree (cf. Section 4). Overall, we derived two research questions:

RQ$_1$ To what extent do corner cases exist in real-world systems?
RQ$_2$ To what extent do corner cases affect variability metrics?

The results show that the investigated corner cases appear regularly in several systems, for instance, influencing over one million lines of feature code in Linux. Moreover, we found that the impact on metrics can be heavy, for instance, changing the scattering degree in MySQL by 25%. We provide our tooling and all measurements of our study in an open-access repository.[1] Finally, we propose ways to address such corner cases and how to better understand their meaning as well as their importance for variability analysis.

2 MOTIVATION

In this section, we first describe and motivate the problem of covert and phantom features (cf. Listing 1). We then describe the individual corner cases we are concerned with in more detail.

2.1 Problem Statement

It is a general problem if metrics mislead developers in their expectations of what is measured [10, 41], as we experienced ourselves. While performing our own analyses of C preprocessor variation points and investigating related studies [29, 30, 40], we searched for the most wide-spread features using scattering degree and the most complex feature interactions using tangling degree (we define these metrics for our study in Section 3.2).

However, we found that the corresponding metric definitions [6] do not generalize over all possible variation points. Basically, scattering degree and tangling degree count feature constants [6, 29, 40]. Guided by these metrics, developers may then assign code locations

[1] https://bitbucket.org/ldwxlnx/splc2019data.git

```
1 #ifdef A
2 // ...
3 #else
4 // ...
5 #endif
```

(a) `#else`.

```
1 #ifndef A
2 // ...
3 #else
4 // ...
5 #endif
```

(b) `#ifndef`.

```
1 #if ! defined(A)
2 // ...
3 #elif ! defined(B)
4 // ...
5 #endif
```

(c) Negations.

Figure 1: Examples of general corner cases we investigated.

to respective features. For instance, regarding the feature constant `CONFIG_PM` in Listing 1, the question arises, whether—although enclosed by an `#ifndef`-block—lines 58 to 78 really do belong to this feature? They are only affected if the feature `CONFIG_PM` is deselected, which still aligns to most metric definitions, but seems unreasonable due to the negating directive.

A different problem arises, as it is unclear whether other features affect this code region, for example, because of an alternative dependency that is not represented in the code [21]. Similarly, the code may be intended to be mandatory, for instance, because the feature `CONFIG_PM` is optional, but requires overwriting of base code. In other words, it is undecidable at first glance, whether such variation points are just *phantom features*, meaning that their feature constant suggests an erroneous affiliation to the respective feature.

Even worse, Lines 78 to 84 are not handled by scattering degree and tangling degree at all, due to missing feature constants in the `#else` directive. For that reason, a potential relation to a specific feature is *covert* and requires manual or automatic inspection of its context (i.e., by analyzing all preceding directives in that particular group). In our example in Listing 1, the actual feature code is present only in these lines, demanding the selection (presence) of the feature `CONFIG_PM`. This results in a blind spot for variability analyses and developers that rely on the corresponding metrics.

2.2 Corner Cases

Several analyses rely on metrics to infer implications from variability in code [6]. However, vague, varying, and incomplete metric definitions can result in deviations. For instance, the tangling degree of variation points simply counts the number of feature constants in a directive, arguing that the variable code is affected by the defined feature. This is reasonable for the usual `#ifdef`, `#if`, and `#elif` directives, but not for the corner cases that we consider in the following. We display corresponding examples in Figure 1.

`#else` *Directives.* Considering already simple `#else` directives, as in Figure 1a, the question arises, whether feature `A` impacts the directly following code (i.e., Line 4)? We could argue that the code depends on the absence of feature `A`, and thus is closely related to it. However, from a maintenance perspective, this argumentation poses problems, for instance, supposing the task: "*Perform a walkthrough of all code sections contributing to feature* `A`." Would we inspect only the code following the `#ifdef` directive, only the code following the `#else` directive, or both?

Because `#else` directives induce *covert* variability, they are obstacles for scattering degree, tangling degree, or any other metric solely relying on feature constants. As aforementioned, `#else` directives may represent various kinds of variability, such as:

- Code that is solely related to the absence of feature A.
- Base code, if feature A needs to override it.
- Code that is related to other features, for example, if feature B is in an alternative dependency to feature A.

Considering the third case, metrics may provide biased results not only for feature A, but also for feature B that is not accounted for.

Negating Directives. In Figure 1b and Figure 1c, we display cases to which we refer to as negations. For example, an `#ifndef` directive (cf. Figure 1b) is the inverse of an `#ifdef` directive (cf. Figure 1a). Many metrics still account its feature constant and thereby the code it encloses to feature A. Moreover, the code in the `#else` directive is only included if feature A is selected, which renders the `#ifndef` directive a *phantom* feature location.

In more complex situations, such as in Figure 1c, the problems become more challenging. The feature constants in each directive are negations, meaning that the corresponding code is only selected in more specific situations than some metrics would indicate. For example, simply counting the number of feature constants results in a tangling degree of one for each directive. However, when developers use such metrics as indicators to infer the complexity of feature interactions—which was probably intended by the inventors [29]—an observation of one directive alone is misleading. Since the `#elif` directive actually requires feature A to be present and B to be absent, there is a relation between two features, which may imply a tangling degree of two. Further considering `#elif` directives, the dependencies can become complex, while the preprocessor needs to check only a single feature in each directive. Again, the issue arises how different metrics may lead to irritating measurements depending on the structure of the code. Eventually, scattering degree and tangling degree count object macro names in source code—no more, no less. This does only partly allow for semantic conclusions based on the metrics' respective values.

Addressing Corner Cases. Properly interpreting all corner cases requires domain knowledge or technical solutions, such as SAT or CSP solvers [46, 48]. Still, if our cases are really corner cases that rarely appear, they may be negligible. In the remaining paper, we investigate this issue to improve the awareness for such cases and propose some initial solutions to mitigate them. We do not claim that our solution is ideal or that existing metrics, on which we also rely, are unsuited for variability analysis. Our goal is to raise awareness for such problems, aiming to initiate further research on their importance and solutions.

3 STUDY DESIGN

In this section, we report the details of our study design, namely our *subject systems*, *metrics*, *methodology*, and *tooling*, which other researchers can reuse to reproduce our study.

3.1 Subject Systems

To answer our research questions, we aimed to investigate a set of real-world and differently sized software systems that comprise preprocessor variability. For this purpose, we selected an initial set of 20 popular and still maintained open-source systems from previous works [29, 30, 33, 40]. Afterwards, we tested our analysis tooling [27] in a pilot study to identify and fix potential errors. During this phase, we found that our tool could not parse 69 files from the test suite of the GNU Compiler Collection (GCC), which comprises syntactically malformed input files for testing the fault and recovery behavior of the C preprocessor and compiler. Consequently, we omitted the GCC in this study and analyzed 19 systems.

Within Table 1, we show our subject systems, which cover a variety of domains (e.g., web servers, operating systems), development periods, and sizes. In particular, we analyzed the Linux Kernel, which is one of the most common subject systems for variability analysis, due to its size of almost 15 million source lines of code and its practical importance. The feature prefixes represent constants that are commonly used and established for external, customer-visible features. For Linux, this is the well-known `CONFIG_` prefix, while Vim uses an abbreviation of feature (`FEAT_`). Furthermore, most systems rely either on prefixes induced by the GNU Autotools suite (`HAVE_`) or make use of prefixes based on established naming conventions (`USE_`). In this study, we focus on such external features [39], while omitting internal variability that may only be used during development. We are aware that these features represent only a subset of each system's full variability and that the selection is not perfect and may distort our results. Nevertheless, we still cover a large set of variability and the extent of corner cases in the features we inspect can be seen as a lower boundary: Only more corner cases are possible, not less. While the ratio of corner cases in the remaining code may be smaller, we still found large differences for some systems. Moreover, we argue that the situation is comparable throughout the selected systems' variability and also for other systems—considering that we relied on established and still maintained open-source projects, which are similar to industrial systems in terms of C preprocessor usage [13].

3.2 Metrics

We aimed to understand the impact of `#else` (covert features) and negating (phantom features) directives on variability analysis. To this end, we discuss respective implications qualitatively and provide a quantitative analysis based on the following three metrics:

LoF *Lines of Feature Code* counts the number of lines that are enclosed by a conditional directive, without excluding whitespaces or comments, as defined by Liebig et al. [29]. Regarding the feature `CONFIG_PM` in our Linux example (cf. Listing 1), the area between the `#ifndef` (line 58) and the `#else` directives (line 78) comprises 19 lines of feature code.

SD *Scattering Degree of Variation Points* measures how many variation points are affected by a specific feature constant. For instance, if `CONFIG_PM` in Listing 1 would not appear in any other part of the system's code, the scattering degree for this feature would be one.

TD *Tangling Degree of Variation Points* represents the counterpart to the scattering degree. It measures the number of different feature constants in a single variation point (directive), for example, for the `#ifndef` in Line 58 of Listing 1 the tangling degree is also one.

We strictly follow these revised definitions of scattering degree and tangling degree of Queiroz et al. [40], for which feature constants of enclosing conditional directives are **not** added to currently investigated expressions, as, for instance, Liebig et al. [29] do.

Table 1: Overview of the subject systems that we considered for this study: The version we used, each version's release year, the domain, development start, and size of C code. We further show the feature prefixes we investigated with the corresponding number of analyzed (N_{Feat}) and the total number of feature expressions (N_{Total}).

System	Version	Year	Domain	Since	#SLOC (C)	Feature Prefixes	N_{Feat}	N_{Total}
APACHE	8.1	2017	Web server	1995	153,357	`(HAVE|USE)_`	86	1,000
CPYTHON	3.7.1rc1	2018	Program interpreter	1989	426,942	`(HAVE|USE)_`	686	4,295
EMACS	26.1	2018	Text editor	1985	330,196	`(HAVE|USE)_`	680	2,327
GIMP	2.9.8	2018	Image editor	1996	761,314	`(HAVE|USE)_`	90	1,996
GIT	2.19.0	2018	Version control system	2005	206,239	`(HAVE|USE)_`	65	821
GLIBC	2.9	2018	Programming library	1987	818,176	`(HAVE|USE)_`	409	5,217
IMAGEMAGICK	7.0.8-12	2018	Programming library	1987	342,797	`(HAVE|USE)_`	5	993
LIBXML2	2.7.2	2018	Programming library	1999	169,761	`(HAVE|USE)_`	117	2,360
LIGHTTPD	1.4.50	2018	Web server	2003	49,693	`(HAVE|USE)_`	173	450
LINUX KERNEL	4.10.4	2017	Operating system	1991	14,746,931	`CONFIG_`	11,011	36,082
MYSQL	8.0.12	2018	Database system	1995	153,157	`(HAVE|USE)_`	355	4,901
OPENLDAP	2.4.46	2018	Network service	1998	287,066	`(HAVE|USE)_`	347	1,377
PHP	7.3.0rc2	2018	Program interpreter	1985	894,426	`(HAVE|USE)_`	1,162	5,977
POSTGRESQL	10.1	2017	Database system	1995	790,282	`(HAVE|USE)_`	387	2,585
SENDMAIL	8.12.11	2018	E-mail server	1983	85,639	`(HAVE|USE)_`	24	1,223
SUBVERSION	1.10.2	2018	Version control system	2000	967,225	`(HAVE|USE)_`	39	1,008
SYLPHEED	3.6.0	2018	E-mail client	2000	117,980	`(HAVE|USE)_`	75	417
VIM	8.1	2018	Text editor	2000	343,228	`(HAVE|USE|FEAT)_`	1,378	2,570
XFIG	3.2.7a	2018	Graphics editor	1985	109,341	`(HAVE|USE)_`	29	193

3.3 Methodology

We addressed our research questions as follows: For $\mathbf{RQ_1}$, we analyzed to what extent corner cases exist within our subject systems based on quantitative data. To address $\mathbf{RQ_2}$ and to demonstrate the impact of corner cases on metrics, we used our tooling to infer all feature constants logically involved in `#else` directives (cf. Section 4.4). This allows us to compare scattering degree and tangling degree measurements with and without inspecting `#else` directives. We use the differences as impact indicator and discuss the measurements qualitatively.

***RQ_1**: Existence of Corner Cases.* To investigate to what extent our subject systems are affected by corner cases, we analyzed the cases' impact on variability based on two aspects:

- *Frequency:* We counted all occurrences of conditional directives that comprise the feature constants under inspection (cf. feature prefixes in Table 1). Furthermore, we grouped these occurrences according to the respective keyword (i.e., `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`) to obtain a detailed overview of the types of variability.
- *Size:* For each occurrence that we found, we additionally computed the respective lines of feature code. Again, we subdivided the results into the respective groups of keywords. By doing so, we investigated what textual volume corner cases contribute to each system.

The results demonstrate the spreading and volume of the investigated variation points within the subject systems, allowing us to reason about their proportions and impact. We examined all conditional directives that applied to the same filtering conditions, namely that at least one feature constant matches the feature prefixes that we show in Table 1 (i.e., `CONFIG_`, `FEAT_`, `HAVE_`, `USE_`). We then analyzed the identified variability as follows:

Totality. We counted the occurrences and lines of feature code grouped by the directives' keyword. Thus, we summarized the overall amount and size of variability under inspection.

Absence. We counted the occurrences and lines of feature code of simple negations for the keywords `#if` and `#elif`. Such negations consist of one unary logical negation (`!`), an optional `defined` operator, and one feature constant (e.g., `CONFIG_PM`). We define this subset of directives as *simple negations*, because their meaning can be easily interpreted by manual inspection. In contrast, we omitted complex expressions (e.g., `!(A && B) || C`). We did this, because the categorization into absence or presence conditions for the feature constants in such expressions requires to solve complex constraints in propositional or higher-order logic.

Presence. We separately counted the occurrences and lines of feature code for `#else` directives representing expressions that are neither simple negations (cf. Figure 1a) nor a complex expression, but imply requested feature constants. This means, that we inferred whether a presence condition is enforced by an `#else` directive and assigned this condition as feature expression (cf. Figure 1b). Logically, such directives' expressions consist of a `defined` operator and exactly one feature constant.

`#if(n)def` Directives. As defined in the C language standard [14], the directives `#ifdef A` and `#ifndef A` are semantically enriched keyword equivalents of `#if defined(A)` and `#if ! defined(A)`, respectively. Since the language standard describes that a `defined` operator's argument is exclusively a single macro name, this group of conditional directives always represents simple presence or

Listing 2: Revealing hidden SD and TD values.

```
1 #if defined(A) && defined(B)
2 // interaction (presence) of A and B
3 #elif defined(C)
4 // simple presence of C (and absence of A and B)
5 #else
6 // interaction (absence) of A, B, and C
7 #endif
```

absence conditions. We counted these classes of directives unrestricted, aside from the precondition that the feature constants matches a defined prefix.

***RQ$_2$**: Corner Cases' Impact on SD and TD.* The scattering degree and tangling degree relate to feature constants and are applied at the level of individual conditional directives. So, their conventional application [6, 29, 30, 40] disregards `#else` directives, due to non-existing feature expressions and constants. We demonstrate the impact of `#else` directives on scattering degree and tangling degree by pointing out the amount of ignored variation points.

To this end, we measured scattering degree and tangling degree with and without the inspection of `#else` directives. As both metrics measure at the level of variation points, differences here are problematic to visualize. For this reason, we use a condensed overview by summing up scattering degree and tangling degree values with and without inspecting `#else` directives for each of our subject systems. With respect to Listing 2, this results in a scattering degree of three for the feature constants `A`, `B`, and `C`, as each exists only at one location. The directive in Line 1 has a tangling degree of two and the directive in Line 3 of one, summing up to three, too. In this example, we completely ignored the meaning of `#else` directives, strictly following the aforementioned metric definitions.

Then, we applied the metrics to all `#else` directives, namely on their inferred feature expressions. For our example in Listing 2, the `#else` directive in Line 5 represents the expression:

```
!((defined(A) && defined(B)) || defined(C))
```

Now, every feature constant exists twice, increasing the scattering degree to two for each feature constant and summing up to six. The tangling degree remains the same as before for Lines 1 and 3, but the `#else` directive has a tangling degree of three, due to the three different feature constants within the expression it represents, resulting in an overall value of six, too.

By computing the differences for each system, we investigated the impact of `#else` directives for both variability metrics. Regarding our examples, the differences for scattering degree and tangling degree are three, for each. Thus, `#else` directives affect both metrics by 50%, considering our example in Listing 2.

3.4 Tooling

In order to analyze our subject systems, we continued to implement our Java application FeatureCoPP [26].[2] FeatureCoPP searches a specified folder for all C header (`.h`) and implementation (`.c`) files of a software system. In parallel, it analyzes the usage of conditional directives within the found files. FeatureCoPP is a purely text-based analysis tool, similar to the C preprocessor. Still, compared to TypeChef [18, 19] and SuperC [12], which actually parse the code, FeatureCoPP performs equally well in identifying presence conditions [27].

[2]https://github.com/ldwxlnx/FeatureCoPP

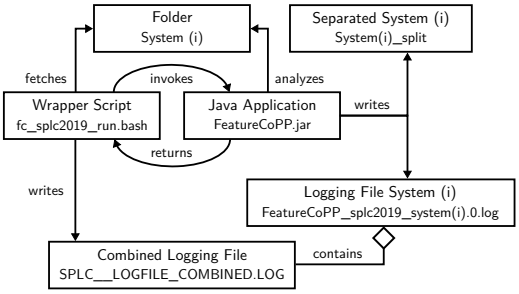

Figure 2: Conceptual behavior of our tool-chain.

We integrated FeatureCoPP into a bash-script tool-chain to automate the analysis and data collection, facilitating the replication of our study. In Figure 2, we depict the general workflow of our tool-chain. First, our *wrapper script* fetches specified systems from their repositories. Then, the script invokes *FeatureCoPP*, which analyzes the systems according to its configuration. During the analysis, FeatureCoPP creates two outputs for each system:

(1) A new system with physically separated features and a report on these features. This is the original purpose of FeatureCoPP [26] and facilitates manual inspection of features, which we used to test our tooling and verify our data.
(2) A *logging file* that tracks the analysis and summarizes the data we need as a statistical overview.

To facilitate reviewing of all systems (i.e., 19 logging files for this study), our wrapper script extracts and combines the statistics of all logging files into a single file. This *combined logging file* provides a condensed and more comprehensible overview of all systems and the corresponding data.

Reproducing this Study. We published our tooling and data in an open-access git repository[1] to enable other researchers to use our setup, for example, to replicate our study, to employ it on other systems, and extend it. In the repository, we provide an extended documentation on how to use and adapt our tool-chain. Moreover, we describe the details of our study and tagged the revision we used, ensuring reusability of this study's setup, despite future developments of FeatureCoPP.

4 RESULTS & DISCUSSION

In this section, we first discuss the suitability of our analysis for answering our research questions. Then, we present our results, discuss their implications, and propose how to address corner cases.

4.1 Appropriateness of Investigated Variability

Results. Before presenting our findings, we evaluate the appropriateness of our investigated subset of features. The question arises, whether we analyzed a representative portion of variability in our subject systems? In Table 2, we show the corresponding statistics. For example, we analyzed 278 variation points (i.e., conditional directives) in the Apache web-server that are induced by the Autotools (`HAVE_`) or by coding conventions (`USE_`). In contrast, we ignored 1,995 variation points, consisting of include guards (directives

Table 2: Variation points in our subject systems.

System	Total	Analyzed	%	Ignored
APACHE	2,273	278	12.23	1,995
CPYTHON	7,997	1,320	16.51	6,677
EMACS	6,524	2,701	41.40	3,823
GIMP	3,763	256	6.80	3,507
GIT	1,506	121	8.03	1,385
GLIBC	17,766	1,167	6.57	16,599
IMAGEMAGICK	3,250	6	0.18	3,244
LIBXML	9,859	427	4.33	9,432
LIGHTTPD	1,101	473	42.96	628
LINUX	102,939	49,771	48.35	53,168
MYSQL	10,328	1,237	11.98	9,091
OPENLDAP	3,992	1,008	25.25	2,984
PHP	17,837	3,474	19.48	14,363
POSTGRESQL	7,796	1,371	17.59	6,425
SENDMAIL	3,422	49	1.43	3,373
SUBVERSION	7,607	381	5.01	7,226
SYLPHEED	1,670	559	33.47	1,111
VIM	15,489	10,713	69.17	4,776
XFIG	523	66	12.62	457

assuring the singular parsing of header files), directives controlling internal variability (e.g., `WIN32`), but also unidentified external variability. However, to identify the missing external variability, we would need specific domain knowledge about the features in each of our subject systems. In total, we examined 12.23% of all variation points in Apache.

Discussion. We examined a noticeable low number of variation points for ImageMagick (six out of 3,250; 0.18%) and Sendmail (49 out of 3,422; 1.43%). Arguably, these systems are outliers that require a more appropriate selection of feature prefixes (cf. Table 1) based on domain knowledge, which we aim to address in future research. Although the results for other systems also indicate relatively low percentages of analyzed variability (e.g., libxml2), we argue that we examined a reasonable amount of directives to assess the impact of our corner cases. In 12 out of 19 subject systems, we investigated more than 10 percent of the overall variability, ranging from MySQL (1,237 out of 10,328 variation points; 11.98%) to Vim (10,713 out of 15,489 variation points; 69,17%). Especially for larger systems, such as Vim, Linux, and Emacs, we analyzed high ratios of variability.

Insight:
Overall, we argue that our subject systems are a sound foundation for our analysis. We deliberately did not exclude outlier systems, in which we analyzed less variability, to show the complete picture and see whether the results are comparable.

4.2 RQ_1: Existence of Corner Cases

Results. In Table 3, we show the number of matched directives we analyzed (N) and their total lines of feature code (LoF). We display these values grouped by directive and for each of our subject systems. Furthermore, we highlight corner-case directives with gray columns. Considering `#else` directives, we analogously show the corner cases in which these enclose code that is related to a feature presence. That is, its preceding `#if` or `#ifndef` forms a simple absence condition for a particular feature constant (cf. Listing 1 and Section 3.3). At the bottom of Table 3, we summarize statistical properties of each measurement based on the actual ratios in percent to show the extent of corner cases compared to the overall variability we analyzed.

In Table 4, we summarize the measurements from Table 3. To this end, we summarize the occurrences (N) and sizes (LoF) of examined directives (All) and their respective subset of corner cases (CC) in order to show how the latter affect each subject system. We also show the percentages of corner cases within each subject system for an easier interpretation of their impact.

Discussion: `#else` Directives. We show the total number of features related `#else` directives per system in the second last compound column (All) in Table 3. In every system, `#else` directives are used in conjunction with the feature constants that we investigated—sometimes rarely, as with two occurrences in ImageMagick comprising 13 LoF, or more frequently, as with 10,449 occurrences in Linux comprising 109,750 LoF. The median and mean values of around 17% demonstrate the quantitative impact of `#else` directives in all systems. We can explain the relatively high dispersion ($s = 7.17$) based on three factors:

(1) *Heterogeneous sizes:* The systems differ in their sizes, which can also impact the corner cases' occurrences and sizes.
(2) *Heterogeneous coding style:* Developers may avoid `#else` directives for project specific reasons (e.g., coding standards).
(3) *Inappropriately selected feature prefixes for subject systems:* As we examined only a subset of the variability in some of our subject systems (cf. Section 4.1), the measurements of these systems may bias the overall image.

In particular, `#else` directives that follow after a simple absence condition (i.e., `#if ! defined` and `#ifndef`) occur rarely (e.g., one in ImageMagick, Libxml2, Xfig, and Sendmail, each). Only seven subject systems comprise such variation points in more than ten cases. Nevertheless, we argue that 488 locations comprising 13,309 LoF in the Linux Kernel or the 15 variation points with 1,340 LoF in Glibc indicate the relevance of this group of corner cases.

Insight:
We argue that the omnipresence of `#else` directives in all subject systems, the comparatively high number of such variation points, and especially their extent in the Linux Kernel underpin their significance and importance for variability analysis and management.

Discussion: Negating Directives. The quantitative impact of simple absence conditions related to `#if` and `#elif` directives apparently converges towards zero, with a mean of 2.54% in terms of occurrences and 2.48% in lines of feature code for `#if` and a median of 0% in both metrics for `#elif` (cf. Section 4.1). The noticeable 33.3% (`#if`) for ImageMagick originate from the small number of analyzed directives and resemble an outlier system. In contrast, negations appear quite regularly as `#ifndef` directives (i.e., $5.91 \pm 5.23\%$ of the code). Again, ImageMagick induces a bias on the measurements as an outlier system. We have to mention that we cannot draw

Table 3: Overview of the analyzed preprocessor directives, including the number of variation points (N) and the corresponding lines of feature code (LoF). To investigate corner cases, we separately list the values for simple negations in `#if`, `#ifndef`, and `#elif` directives (Absence) as well as situations where an `#else` block is connected to such a negation. So, the `#else` directive indicates the presence of code if the previously checked feature expression is true. To make the results more comprehensible, all corner cases are highlighted in gray. Thereby, we distinguish between phantom (•) and covert (○) features.

	`#if`				`#ifdef`		`#ifndef`		`#elif`				`#else`			
	All		• Absence		Presence		• Absence		All		• Absence		All		○ Presence	
System	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF
APACHE	28	405	0	0	208	3,165	13	121	5	295	0	0	24	206	4	92
CPYTHON	275	5,357	15	73	727	14,372	53	753	37	369	0	0	228	1,977	10	80
EMACS	613	17,318	64	2,191	1,391	50,669	158	4,418	74	1,655	3	23	465	8,149	30	710
GIMP	60	1,242	0	0	130	2,100	22	300	14	233	0	0	30	286	6	137
GIT	21	236	1	3	65	974	8	101	2	32	0	0	25	306	3	144
GLIBC	393	5,261	14	108	416	4,196	71	905	6	57	0	0	281	3,368	15	1,340
IMAGEMAGICK	4	50	2	20	0	0	0	0	0	0	0	0	2	13	1	6
LIBXML2	47	724	3	43	263	2,238	22	115	45	465	0	0	50	577	1	7
LIGHTTPD	112	1,197	0	0	254	4,325	12	562	12	91	0	0	83	475	2	21
LINUX	7,221	161,649	118	1,454	29,805	849,807	1,409	21,055	887	6,247	11	141	10,449	109,750	488	13,309
MYSQL	97	2,248	9	761	728	7,641	74	784	43	298	0	0	295	1,727	5	24
OPENLDAP	133	2,954	1	3	571	16,304	38	827	71	857	2	10	195	1,725	5	76
PHP	1,208	98,827	20	781	1,570	25,951	125	1,013	129	1,142	2	14	442	4,329	17	1,370
POSTGRESQL	142	2,076	11	45	785	19,160	96	823	36	267	1	3	312	2,896	16	167
SENDMAIL	32	256	3	22	3	202	6	18	0	0	0	0	8	147	1	13
SUBVERSION	16	244	0	0	145	815	92	360	11	155	0	0	117	451	2	63
SYLPHEED	304	7,032	0	0	189	7,246	7	28	5	42	0	0	54	317	0	0
VIM	1,870	203,255	12	124	7,598	88,503	193	3,094	25	108	0	0	1,027	11,295	32	960
XFIG	3	11	0	0	53	459	4	49	0	0	0	0	6	185	1	151
Unweighted statistical summary of ratios in percent (%)																
MEAN	24.00	28.88	2.54	2.48	49.68	51.44	5.91	4.59	2.67	2.52	0.02	0.01	17.75	12.56	1.82	3.45
MEDIAN	17.46	18.00	0.58	0.18	53.72	59.07	5.15	3.30	2.54	1.37	0.00	0.00	17.27	9.91	0.98	1.04
STD. DEV.	19.07	23.19	7.59	7.26	20.40	21.02	5.23	4.08	2.67	3.17	0.05	0.01	7.17	7.82	3.67	5.42
MIN.	4.20	1.56	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	8.63	2.16	0.00	0.00
MAX.	66.67	79.37	33.33	31.75	80.30	75.97	24.15	17.78	10.54	11.29	0.20	0.04	33.33	26.28	16.67	21.45

conclusions on the influence of `#else` directives representing simple absence conditions. This relates to the fact that the difference of simple presence conditions from the totality of such directives also includes more complex expressions, for example, inferred from preceding `#if` and multiple `#elif` directives.

Insight:
While simple absence conditions appear seldom in our subject systems, we argue that the consequent use of `#ifndef` directives asks for analyzing and discussing these directives.

Summary. While single corner cases may appear rarely, we can see (cf. Table 4) that the total ratios of affected directives and feature code vary heavily. For instance, in Sylpheed, we can see that corner cases affect only 2.35% of feature code, while they represent 10.91% of directives. More extreme, for Linux, we analyzed 49,771 directives in total, of which 24.08% are connected to our corner cases, affecting over 100 thousand (11.53%) lines of feature code.

Answering RQ_1 (existence of corner cases):
While our analysis shows that we cover only a small part of some systems, we are still able to emphasize the relevance and impact of our corner cases. The results show that these corner cases appear regularly, and thus require more attention.

4.3 RQ_2: Corner Cases and Metrics

Results. In order to show the impact of our corner cases on the scattering degree and tangling degree, we show the summarized results (cf. Section 3.3) for `#else` directives in Table 5. For instance, within the Linux Kernel, we obtained a summarized scattering degree of 41,992 and 54,056 for omitting ($SD_{\neg else}$) and including (SD_{else}) `#else` directives, respectively. This represents a noticeable difference (Δ) of 12,064 (22.32%). Likewise, the tangling degree is remarkably influenced in Linux (a $TD_{\neg else}$ of 42,335 compared to a TD_{else} of 54,634) with a loss of 22.51%, caused by not taking `#else` directives into account. Each of our subject systems comprises a sufficient number of `#else` directives, which explains average misses for the scattering degree of $18.37 \pm 7.16\%$ and for the tangling degree of $20.03 \pm 9.64\%$.

Discussion. The dispersions for both metrics are caused by systems like Vim (9.39%, 8.97%) or Apache (8.56%, 9.27%) that comprise rather small differences. This situation may occur due to two reasons:

(1) A modest usage of `#else` directives.
(2) Less complex preceding conditional expressions, resulting in less complex expression equivalents for `#else` directives.

However, our results indicate that metrics that address only feature constants—and thus ignore `#else` directives—are not able to draw reliable conclusions on a systems' variability. We underpin

Table 4: Comparison of corner cases (CC) to all variation points analyzed in numbers (N), size (LoF), and percent (%).

	N			LoF		
System	**All**	**CC**	**%**	**All**	**CC**	**%**
APACHE	278	37	13.31	4,192	327	7.8
CPYTHON	1,320	296	22.42	22,828	2,803	12.28
EMACS	2,701	690	25.55	82,209	14,781	17.98
GIMP	256	52	20.31	4,161	586	14.08
GIT	121	34	28.1	1,649	410	24.86
GLIBC	1,167	366	31.36	13,787	4,381	31.78
IMAGEMAGICK	6	4	66.67	63	33	52.38
LIBXML2	427	75	17.56	4,119	735	17.84
LIGHTTPD	473	95	20.08	6,650	1,037	15.59
LINUX	49,771	11,987	24.08	1,148,508	132,400	11.53
MYSQL	1,237	378	30.56	12,698	3,272	25.77
OPENLDAP	1,008	236	23.41	22,667	2,565	11.32
PHP	3,474	589	16.95	131,262	6,137	4.68
POSTGRESQL	1,371	420	30.63	25,222	3,767	14.94
SENDMAIL	49	17	34.69	623	187	30.02
SUBVERSION	381	209	54.86	2,025	811	40.05
SYLPHEED	559	61	10.91	14,665	345	2.35
VIM	10,713	1,232	11.5	306,255	14,513	4.74
XFIG	66	10	15.15	704	234	33.24

our argumentation with Figure 3, in which we show an example from the Linux Kernel. In this example, we depict the five most scattered feature constants, which are ranked by their corresponding scattering degree. Furthermore, to derive this ranking, we measured with (□) and without (■) analyzing `#else` directives. Unsurprisingly, the scattering degree values increase for each feature constant that occurs in any `#if`, `#ifdef`, `#ifndef`, and `#elif` directive that precedes an `#else` directive, if we also analyzed that respective `#else` directive. For example, the scattering degree of feature `CONFIG_PM` increases from 1,597 to 2,018, which is caused by 421 occurrences in `#else` directives. Remarkable is the exchange of features `CONFIG_COMPAT` and `CONFIG_OF` between rank four and five. Without the analysis of `#else` directives, the feature `CONFIG_COMPAT` has a slightly higher scattering degree of 683. When we analyzed `#else` directives, the feature `CONFIG_OF` is suddenly scattered more often with a scattering degree of 894.

Similar situations appear for the tangling degree in `#else` directives. For instance, for the Linux Kernel, we found a maximum tangling degree of 12 for the `#if` directive in Line 1202 of the file `linux-4.10.4/drivers/tty/vt/keyboard.c` when we did not analyze the corresponding `#else` directive. In contrast, we identified the feature expression with the highest number of different feature constants for the `#else` in Line 141 of the file `linux-4.10.4/arch/mips/include/asm/module.h`, which has a tangling degree of 28. Admittedly, this particular variation point comprises only a single line of feature code with no functional impact (i.e., it is an `#error` directive).

Answering RQ_2 (corner cases' impact on metrics): The example and our results are good indicators for the impact of ignoring `#else` directives in practice: The measurements based on existing metrics may be wrong and can lead to faulty assumptions about a project's variability.

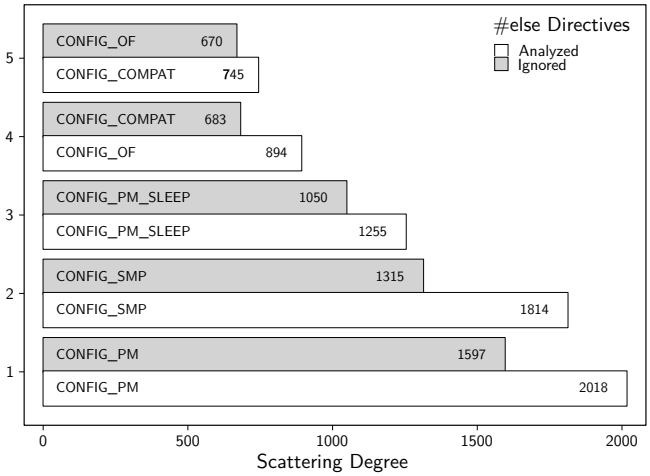

Figure 3: Linux' five most scattered features with and without analyzing `#else` directives.

4.4 Tackling Corner Cases

As we demonstrated the relevance of `#else` and negating directives for our subject systems, we now suggest an initial idea on how to cope with the corner cases discussed.

`#else` Directives. Variability analysis tools using scattering degree and tangling degree in a blackbox approach (cf. Section 3.2) would be more precise if `#else` directives are also analyzed. Yet, the expressiveness of the obtained values remains limited. In case the tooling follows a white box approach—meaning that it presents developers a detailed overview of all directives examined together with their respective file locations—the quality of the analysis increases alongside with developers' variability knowledge of the system under inspection. Still, the question remains: How to obtain involved feature constants especially for bare `#else` directives?

As a starting point, we propose our technique that we used in our tooling (cf. Section 3.4) to perform this study. We sketch our technique conceptually in Figure 4 and display a corresponding code example in Listing 3. Our technique parses contiguous conditional directives with a generated LALR(1) parser [5], which is based on the C language standard specification [14]. During the syntactical analysis, we create an abstract syntax tree (AST), consisting of each directive's operators and operands (e.g., feature constants). Furthermore, associativity and precedence of the respective conditional directive's feature expression are preserved by the AST structure. We need to handle the directives `#ifdef` and `#ifndef` separately, as the keywords (`#if`) are semantically enriched with operations (i.e., `!`, `defined`). Thus, we transform such directives' ASTs into an equivalent AST containing a `defined` operator and, if necessary, a leading logical unary negation (`!`), to separate the operations from the keyword. This is important to derive correct feature expressions for potentially following `#else` directives. In the last step, we preserve the ordering of a complete conditional directive: All ASTs from the opening `#if(def)` to the closing `#endif` are interconnected, preserving their order in the source code. If our technique recognizes an `#else` directive, it interconnects all previously built ASTs with logical disjunctions (`||`). Finally, the

Table 5: Comparison of unweighted and summarized scattering and tangling degrees with and without (¬) `#else` analysis.

System	$SD_{\neg else}$	SD_{else}	Σ %	Missed Δ	$TD_{\neg else}$	TD_{else}	Σ %	Missed Δ
APACHE	267	292	8.56	25	274	302	9.27	28
CPYTHON	1,189	1,443	17.60	254	1,308	1,598	18.15	290
EMACS-26.1	2,551	3,081	17.20	530	2,692	3,268	17.63	576
GIMP	228	263	13.31	35	236	277	14.80	41
GIT	98	124	20.97	26	116	147	21.09	31
GLIBC	935	1,229	23.92	294	1,192	1,584	24.75	392
IMAGEMAGICK	4	6	33.33	2	4	8	50.00	4
LIBXML2	384	447	14.09	63	410	496	17.34	86
LIGHTTPD	453	551	17.79	98	469	569	17.57	100
LINUX	41,992	54,056	22.32	12,064	42,335	54,634	22.51	12,299
MYSQL	969	1,295	25.17	326	1,016	1,391	26.96	375
OPENLDAP	884	1,138	22.32	254	908	1,194	23.95	286
PHP	3,286	3,816	13.89	530	3,567	4,283	16.72	716
POSTGRESQL	1,104	1,442	23.44	338	1,167	1,551	24.76	384
SENDMAIL	41	49	16.33	8	52	61	14.75	9
SUBVERSION	265	382	30.63	117	278	405	31.36	127
SYLPHEED	506	563	10.12	57	513	577	11.09	64
VIM	10,741	11,854	9.39	1,113	11,722	12,877	8.97	1,155
XFIG	63	69	8.70	6	61	67	8.96	6
MEAN			18.37	849.47			20.03	893.11
MEDIAN			17.60	117.00			17.63	127.00
STD.DEV.			7.16	2,729.46			9.64	2,778.26
MIN			8.56	2.00			8.96	4.00
MAX			33.33	12,064.00			50.00	12,299.00

created AST gets a logical unary negation as root node. This equivalence transformation follows De Morgan's laws. We illustrate this procedure in Figure 4, where we use the feature expressions of three conditional directives (i.e., one `#ifdef` and two `#elifs`), to construct a semantically equivalent feature expression for the trailing `#else` directive.

Our artificial creation of expressions for `#else` directives has the following advantages:

- We can create a textual representation of an `#else` directive by traversing the AST. Afterwards, each `#else` has a virtual name, which leverages developers to spot respective code locations more easily.
- The generated AST can be applied to a SAT or CSP solver in order to test satisfiability of the `#else` directive [46, 48]. This allows to analyze `#else` directives more easily, for example, to identify dead features.

Insight:
The preservation of feature constants allows us to apply scattering degree and tangling degree on every variation point, which results in more precise measurements.

Absence and Presence. The main issue with metrics, such as scattering degree and tangling degree, is the divergence from what is actually measured and what developers may expect to infer from values obtained with such metrics [10, 41]. Since scattering degree and tangling degree only allow to draw conclusions with regards to what feature name is somehow textually involved in what code location, they are limited to perform a sound reasoning about feature presence or absence conditions. Chances may be high that such text locations really enclose actual feature code, but, considering our results, this can barely be seen as a rule of thumb.

To improve metrics that build on counting feature constants (or other values) in preprocessor directives, the semantic evaluation of such variation points is necessary. Consequently, we need tooling that is able to perform constraint solving, since expressions in variation points do not only allow Boolean logic, but even arithmetic, bitwise, and relational operations [14]. With regards to our example in Listing 3, a metric for mapping text locations to actual features could roughly behave as follows:

For each conditional directive in a contiguous group, repeat until `#endif` *is reached:*

(1) *Emulate the C preprocessor's control flow.* In order to read the second `#elif`, the absence of feature `A` is necessary. Thus, a preceding expression needs to be negated and conjunctively connected to its successor expression (i.e., `!defined A && B && C`). If a directive has no predecessor and is not an `#else` directive, we can take it as is. If a directive is an `#else`, the conjunction of all negated preceding directives is exactly its feature expression.
(2) *Find all models.* After creating a respective expression, we can introduce it to a CSP solver. For each found solution,

Listing 3: Complex contiguous conditional directives.

```
1 #ifdef A
2 // presence of A
3 #elif B && C
4 // presence of B and C (absence of A)
5 #elif ! defined D
6 // absence of A, B, C, and D
7 #else
8 // absence of A, B, and C, presence of D
9 #endif
```

every feature constant is counted only once if its value expands to a value different from zero in a solution model. This counting happens only once, regardless whether the same constant has a value different from zero in multiple possible solutions or not. Admittedly, this step is expensive in terms of computation time [42].

(3) *Summarize occurrences.* After this step, for the tangling degree, we can summarize all counted feature constants, and for the scattering degree, we can increment a global counter for each constant. We arguably obtain more precise results for both metrics, as we analyze covert and phantom features.

This conceptual technique ignores the nesting of directives, though. However, it can be extended to consider surrounding directives in the same conjunctive fashion. To this end, a tool must be able to follow file inclusion directives (`#include`), since nesting structures may occur only after such inclusions.

This solution is only a rough scaffold for extending metrics that are based on feature constants towards semantic capabilities. Or simply put, this solution may be a way to let these metrics answer questions, such as: What directive does really enclose feature code?

Insight:
Metrics that rely on analyzing feature constants are only suitable to identify feature locations if the respective features' presence is guaranteed, and thus absence is ruled out.

5 THREATS TO VALIDITY

In this section, we provide an overview of threats to validity. To this end, we follow proposed classifications [38, 50] and discuss the *construct*, *internal*, and *external* validity of our study.

Construct Validity. Considering the construct validity, we strictly followed the metric definitions of other researchers. In particular, for lines of feature code, we used the definition of Liebig et al. [29], namely, we counted any line between consecutive conditional directives. Regarding scattering degree and tangling degree, we used the revised definitions of Queiroz et al. [40], meaning that we did not construct complex feature expressions from nested directives, as done by Liebig et al. [29]. Consequently, the results for these metrics provide insights into the impact of considering corner cases, but may not be representative about the systems themselves. They are still suitable to achieve our goal, and thus we argue that we mitigated this threat.

Moreover, we are fully aware of the shortcomings of so called "size metrics" [10, 41], such as lines of feature code. Since these only provide an absolute measure for actual **lines** of code (no matter how measured), they are never a reliable measure (and moreover language independent) to draw conclusion of the actual systems' code volume. However, as these metrics are heavily used in the scientific community, we still used them to allow for comparisons to other research in this area.

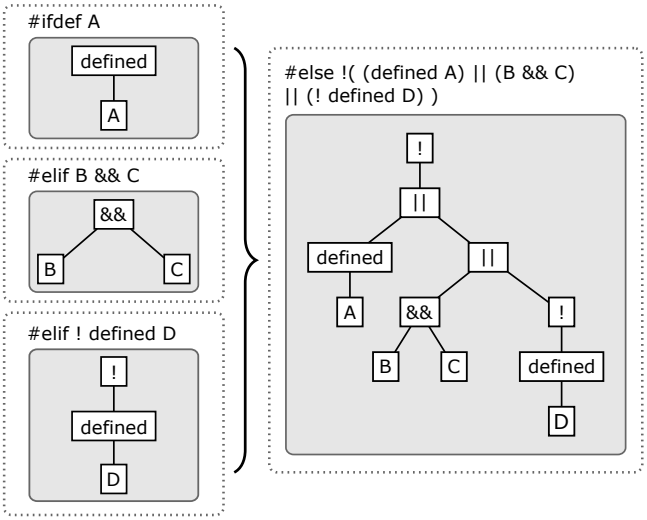

Figure 4: Automatic synthesis of `#else` directives, exemplified for Listing 3.

Internal Validity. A threat to the internal validity is the fact that we implemented our own tooling, which is more lightweight compared to existing, static variability-analysis tools. There may be bugs caused by unidentified and unintended usage patterns of our tooling. However, we have tested it extensively and applied it carefully. Moreover, we already compared it to existing tools, namely TypeChef [19] and SuperC [12], and found that it performs similar or even better for our purposes [27]. Although we addressed this threat properly, we cannot completely avoid it.

External Validity. For the external validity, we are aware that we only consider a limited set of features, corner cases, and metrics. All these points limit our ability to generalize our results. Despite these limitations, we could already show strong impacts of corner cases on variability analysis. Considering that we found over one million feature lines of code being affected in Linux, we argue that the unveiled problems are important to consider. Furthermore, conducting our study exclusively on open-source subject systems may prevent us from generalizing the results with regards to proprietary software systems. However, Hunsen et al. [13] have demonstrated that C preprocessor usage patterns are nearly identical between open-source and closed-source systems. For this reason, we consider this limitation as tolerable and argue that our insights are also relevant for industrial systems.

6 RELATED WORK

Open-source systems that incorporate C preprocessor variability are common subject systems for static variability analyses, due to their availability and usage in practice. Consequently, there are numerous studies on analyzing the conditional directives of the C preprocessor. For example, Liebig et al. [29, 30] are concerned with

the discipline of preprocessor usage. They argue that using such directives in undisciplined styles, for example, annotating code below the statement level, hampers the applicability of tools and the maintainability of code. Based on these findings, Medeiros et al. [33] have proposed a set of refactorings to improve the discipline of preprocessor directives and showed that most developers prefer this refactored code. In contrast, Schulze et al. [43] report a controlled experiment in which they found no differences between disciplined and undisciplined annotations on program comprehension.

The initial discussions on problems of the C preprocessor on software development emerged from personal, negative opinions and experiences [3, 47]. To underpin these experiences, several researchers have investigated the problems of C preprocessor usage based on user studies [24, 28, 32, 45]. They partly show that the raised problems exist and can be improved, but that a lot depends on the single developer and the preprocessor usage.

Fenske et al. [9] report an empirical study on the C preprocessor's change proneness, and thus tackle the impact of negating and `#else` directives from a different perspective. To this end, the authors inspect variation points only in implementation files (`.c`) at function level. As a result, they disregard variability in interface declarations and inline function definitions [14] induced by header files (`.h`).

Sincero et al. [46] present a linear growing algorithm to create propositional formulas from C preprocessor variation points in C source code. The authors focus primarily on aspects of satisfiability to detect, for example, dead feature code. To this end, they also take `#else` directives into account and transform them into equivalent expressions, comparable to our technique (cf. Section 4.4). While the authors assumed a potential impact of their technique on the understanding of variability and corresponding metrics, they did not address this topic, as we did with this work.

Some researchers have proposed new techniques to replace or improve C preprocessor variability. Most prominently may be the concept of virtually separating concerns [15, 16] and adding background colors to highlight feature code within an integrated development environment [7, 8]. While empirical studies show advantages for both techniques, they do not seem to be adopted in practice—where conventional development tools still dominate. Other researchers have proposed to integrate, combine, or replace preprocessor code with other variability mechanisms [17, 25, 26], for example, by using projectional editing to simply switch the representations of feature code to the user [4, 35, 49].

Considering different metrics, Queiroz et al. [40] investigate whether these have special statistic properties—namely power laws. Krüger et al. [23] compare them for mandatory and optional features in Marlin. Both studies show that the metrics align to certain patterns and may help developers to better understand the code or identify critical parts. Overall, numerous tools [22, 34] and metrics [6] have been proposed to perform such analyses.

In the context of reverse variability engineering, several researchers aim to extract variability information, for example, feature constraints from preprocessor directives [36, 37] or configuration options [31]. The goal of such techniques is to understand dependencies among configuration options (and thus their features) and their relation to the source code. To support the understanding of variability in a software system and define its configuration space, the reverse engineered constraints are used to derive a variability model of the system [1, 2, 44]. The results of our study may indicate limitations for such automated techniques, depending on how and what constraints the techniques extract and use—but can also guide new concepts to extend the currently existing techniques.

We are not aware of another study investigating corner cases of the C preprocessor and their impact on variability analysis and metrics. The aforementioned works all report and investigate other issues of the C preprocessor. Some of them also use different sets of metrics to perform their studies. Still, none of them seems to address the issue of negations or `#else` directives and their impact on metrics for variability analysis. Therefore, our work differs from existing works and can be seen as a complement that may ask for reshaping some metrics or investigating to what extent these are suitable to achieve the goals of previous studies.

7 CONCLUSION

In this paper, we analyzed how specific corner cases of C preprocessor directives, namely negating and `#else` directives, affect the variability analysis of such systems. To this end, we focused on determining the extent to which such cases appear in the source code of 19 open-source systems. We compared the results for scattering degree and tangling degree after applying them to our corner cases. Finally, we proposed first steps towards improving variability analysis and the conceptual understanding of corner cases. However, our most important results indicate that:

- Corner cases can hide variability information that only exist as domain knowledge.
- Corner cases appear frequently in several systems, whether as `#else` directives—inducing anonymous, *covert* variation points—or as negated feature expressions, which require the absence of a feature—yielding *phantom* variation points regarding their involved feature constants.
- Resolving corner cases can significantly change the results of variability metrics that focus on counting feature constants, such as scattering degree and tangling degree.
- Variability enclosed in corner cases has a relevant textual size, which highlights the importance of considering covert and phantom features in real-world systems.

We hope to motivate discussions and further analyses regarding evaluation, precision, and purpose of existing and upcoming variability metrics. Based on our findings, we argue that more research about such corner cases is necessary to better understand their characteristics and provide guidance for practitioners.

To this end, we plan to extend our analysis significantly, including more advanced parsers and metrics, as well as more subject systems. A particularly interesting factor that we want to focus on is the question how and why corner cases are applied: When and for what purpose are developers using them? For this purpose, we intend to conduct interview studies and surveys on various systems to collect real-world experiences and practices. This knowledge may help us to improve and scope variability analysis and to reason about the importance of considering corner cases.

ACKNOWLEDGMENTS

This research has been supported by the German Research Foundation (DFG) project EXPLANT grants LE 3382/2-1 and LE 3382/2-3.

REFERENCES

[1] Mathieu Acher, Benoit Baudry, Patrick Heymans, Anthony Cleve, and Jean-Luc Hainaut. 2013. Support for Reverse Engineering and Maintaining Feature Models. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 20:1–20:8. https://doi.org/10.1145/2430502.2430530

[2] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. 2012. Efficient Synthesis of Feature Models. In *International Software Product Line Conference (SPLC)*. ACM, 106–115. https://doi.org/10.1145/2362536.2362553

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[4] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 563–574. https://doi.org/10.1109/ICSE.2017.58

[5] Frank DeRemer. 1969. *Practical Translators for LR(k) Languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[6] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 106 (2019), 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

[7] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering* 18, 4 (2013), 699–745. https://doi.org/10.1007/s10664-012-9208-x

[8] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. 2011. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Annual Conference on Evaluation & Assessment in Software Engineering (EASE)*. IET, 66–75. https://doi.org/10.1049/ic.2011.0008

[9] Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 77–90. https://doi.org/10.1145/3136040.3136059

[10] Norman Fenton and James Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach*. CRC Press. https://doi.org/10.1201/b17461

[11] Cristina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 26, 3 (2001), 109–117. https://doi.org/10.1145/379377.375269

[12] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 323–334. https://doi.org/10.1145/2254064.2254103

[13] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[14] ISO/IEC. 2011. *Programming Languages - C*. Technical Report ISO/IEC 9899:201x. International Standards Organization.

[15] Christian Kästner. 2010. *Virtual Separation of Concerns*. Ph.D. Dissertation. Otto-von-Guericke University.

[16] Christian Kästner and Sven Apel. 2009. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

[17] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. *SIGPLAN Notices* 45, 2 (2009), 157–166. https://doi.org/10.1145/1837852.1621632

[18] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 805–824. https://doi.org/10.1145/2048066.2048128

[19] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 25–32. https://doi.org/10.1145/1868688.1868693

[20] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall.

[21] Sebastian Krieter, Jacob Krüger, and Thomas Leich. 2018. Don't Worry About It: Managing Variability On-The-Fly. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 19–26. https://doi.org/10.1145/3168365.3170426

[22] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Open Infrastructure for Product Line Analysis. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 5–10. https://doi.org/10.1145/3236405.3236410

[23] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 105–112. https://doi.org/10.1145/3168365.3168371

[24] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *Symposium On Applied Computing (SAC)*. ACM, 2044–2052. https://doi.org/10.1145/3167132.3167351

[25] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2018. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience* 48, 3 (2018), 402–427. https://doi.org/10.1002/spe.2525

[26] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 74–84. https://doi.org/10.1145/3001867.3001876

[27] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 284–288. https://doi.org/10.1145/3233027.3236399

[28] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150. https://doi.org/10.1109/VLHCC.2011.6070391

[29] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 105–114. https://doi.org/10.1145/1806799.1806819

[30] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202. https://doi.org/10.1145/1960275.1960299

[31] Max Lillack, Christian Kästner, and Eric Bodden. 2017. Tracking Load-Time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (2017), 1269–1291. https://doi.org/10.1109/TSE.2017.2756048

[32] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 495–518. https://doi.org/10.4230/LIPIcs.ECOOP.2015.495

[33] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[34] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 94–101. https://doi.org/10.1145/2647908.2655972

[35] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *International Conference on Software Engineering (ICSE)*. ACM, 81–84. https://doi.org/10.1145/3183440.3183499

[36] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *International Conference on Software Engineering (ICSE)*. ACM, 140–151. https://doi.org/10.1145/2568225.2568283

[37] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841. https://doi.org/10.1109/TSE.2015.2415793

[38] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *International Conference on on Software Engineering - Future of Software Engineering Track (ICSE)*. ACM, 345–355. https://doi.org/10.1145/336512.336586

[39] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer. https://doi.org/10.1007/3-540-28901-1

[40] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *Software & Systems Modeling* 16, 1 (2017), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[41] Jarrett Rosenberg. 1997. Some Misconceptions About Lines of Code. In *International Software Metrics Symposium (METRIC)*. IEEE, 137–142. https://doi.org/10.1109/METRIC.1997.637174

[42] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.

[43] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 65–74. https://doi.org/10.1145/2517208.2517215

[44] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *International Conference on Software Engineering (ICSE)*. ACM, 461–470. https://doi.org/10.1145/1985793.1985856

[45] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 17–24. https://doi.org/10.1145/2377816.2377819

[46] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-based Variability. In *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 33–42. https://doi.org/10.1145/1868294.1868300

[47] Henry Spencer. 1992. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX Conference*. 185–197.

[48] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *European Conference on Computer Systems (EuroSys)*. ACM, 47–60. https://doi.org/10.1145/1966445.1966451

[49] Markus Voelter. 2010. Implementing Feature Variability for Models and Code with Projectional Language Workbenches. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 41–48. https://doi.org/10.1145/1868688.1868695

[50] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-1-4615-4625-2