Evolutionary Feature Dependencies: Analyzing Feature Co-Changes in C Systems

Sandro Schulze
Anhalt University of Applied Sciences
Köthen, Germany
sandro.schulze@hs-anhalt.de

Phillipp Engelke
Otto-von-Guericke University
Magdeburg, Germany

Jacob Krüger
Eindhoven University of Technology
Eindhoven, The Netherlands
j.kruger@tue.nl

***Abstract*—Configurable software systems and software product lines build on features as first class entities for reasoning about commonalities and variability among system variants. While it is desirable to have modular features, this is not always achievable and research has shown that features interact frequently, which can come with negative effects like security vulnerabilities or bugs. Intensive research has been conducted regarding how and when features interact, focusing primarily on the implementation level and the variability mechanism therein. However, besides such structural, *explicit* feature dependencies represented in the code, there may also be more subtle, *implicit* feature dependencies. In this paper, we build on the idea that the co-evolution of features (i.e., co-changes between features) can reveal implicit dependencies, and thus point to poor design decisions that result in additional maintenance effort. We present a technique for analyzing feature co-changes based on repository mining and association rule mining to identify features that commonly change together and to reveal implicit dependencies. Moreover, we provide a large-scale multi-case study on five C systems (e.g., Linux kernel) to evaluate whether and how frequent such evolutionary dependencies occur. Our results reveal that a) feature co-changes occur quite frequently (25 to 70 % of commits), b) a considerable amount of changes are supported by association rules (i.e, do not occur by chance), and c) several of these co-changes cannot be explained via explicit feature interactions. Overall, our technique and study complement existing research on feature dependencies and interactions by providing means for understanding implicit dependencies that are represented by feature co-evolution.**

***Index Terms*—software evolution, repository mining, code analysis, configurable software systems, association rule mining**

I. Introduction

Configurable software systems or actual software product lines build on a variability mechanism to enable or disable configuration options and thereby features, to customize a specific variant of the system [3], [42], [49], [54]. The most prominent and widely established of such mechanisms is the C preprocessor [15], [29], [30], [33] that builds on conditional compilation (e.g., annotating features with `#ifdef A – #endif` blocks). Different configurable features may depend on each other, for instance, one may require the other or they are tangled in the same conditional compilation annotation (e.g., `#if A && B`). Such dependencies are *explicitly* documented directly in the source code, and ideally stored in a variability model [4], [17], [40] and configuration tool to check whether a configured system variant is valid (i.e., fulfills all dependencies). After configuring the system, the C preprocessor removes all code that is within blocks of disabled features. So, some feature code that impacts a different feature as well may be removed accidentally. To ensure that all variants that can be configured work as intended, many testing techniques have been proposed to sample and test particularly explicit feature dependencies (i.e., sampling, combinatorial testing) [5], [51].

In contrast, there are many dependencies between features that are not (made) explicit, and thus far less obvious (e.g., a responsible developer knowing but not documenting a dependency). For example, two features may be located at completely different places in a system, yet they use or modify the same variable [23], [24], [44]. As a consequence, one feature may impair the behavior of another or the whole configuration may break, a case referred to as feature interaction problem [18]. Such *implicit* dependencies are ideally made explicit (e.g., in the variability model). However, considering a large number of features (e.g., >18,000 for the Linux kernel) [50], the cognitive challenge of understanding all feature interactions (called "`#ifdef` hell" regarding the C preprocessor) [6], [15], [26], [29], [35], and that interactions may not be directly visible [23], [24], [31], [44], it is not surprising that implicit feature dependencies can cause severe problems for software engineers. In fact, while we are aware that feature interactions are a prominent cause for bugs in configurable systems [1], [10], [34], [37], it is particularly challenging to identify, locate, and resolve these bugs—even if the causing dependency is explicit.

With our research in this paper, we aim to identify implicit feature dependencies to reveal possible design flaws, and to support developers with information on possible co-changes that must be considered during a system's development and evolution. To move into this direction, we build on an intuitive assumption: even implicit feature dependencies must be somehow represented in the developers' code changes. Specifically, even if no dependency is made explicit between two features in the source code, the developers may always change the two together in one commit or a change to one feature is typically followed by a change to a second one. Our idea is to exploit such change information using association rule mining and a recommender system to extract implicit feature dependencies. To evaluate our technique, we employed it on five open-source C systems: OpenVPN, libxml2, OpenLDAP, postgres, and the Linux kernel. Our results show that our technique can help reveal implicit feature dependencies in

such systems. Seeing the extent to which we could identify such implicit dependencies and that not all of them can be directly mapped to explicit dependencies, we argue that we have to pay more attention to analyzing implicit dependencies in configurable source code in the future.

In more detail, we contribute the following in this paper:

- We propose a technique for eliciting and analyzing implicit feature dependencies based on feature co-changes.
- We report a multi-case study in which we applied our technique on five open-source C systems of varying sizes.
- We publish our technique and analysis data in a persistent open-access repository[1] for others to use it.

Overall, we contribute a technique that complements existing research, and can help analyze hidden properties of features in configurable systems. So, we contribute a foundation for future research as well as a technique for practitioners to identify feature co-changes and implicit feature dependencies they may want to resolve or make explicit.

II. Association Rule Mining

The concept of *association rule mining* has been introduced by Agrawal et al. [2] with the goal of inferring relations, called *association rules*, between *entities* in a dataset. Originally, association rules are identified from a set of *transactions*, with each transaction encompassing a subset of all entities. As an example, we refer to the famous case of analyzing data about shopping carts: Given that multiple transactions (carts) include bread and butter (entitities), the corresponding association rule is $bread \rightarrow butter$, indicating that "*if you buy bread, it is likely that you also buy butter.*"

Association rule mining has been successfully applied for mining evolutionary coupling from (co-)change data in software repositories [16], [45], [53], [55]. In this context of evolutionary coupling, there are a few terms that differ from the original concept of association rule mining. First, the entities are the files (or, depending on the granularity, classes, methods, features, etc.) of the system. Second, the collection of transactions is the history $\mathcal{H}$ of the system with each transaction $T \in \mathcal{H}$ being a *committed change* to the system. Thus, a transaction encompasses a number of files that have been changed or added together, and thus creates a *logical dependency* [12].

Moreover, in the context of evolutionary coupling, an association rule is an implication `A` $\rightarrow$ `B` with `A` and `B` being *disjoint*. `A` is referred to as the *antecedent* and `B` is referred to as the *consequent*. Consequently, the association rule above implies that "*if files in A change, it is also likely that files in B are subject to change*". To reason about and validate the obtained association rules, several measures have been proposed, which we briefly introduce in the following.

1) Frequency: The frequency of a rule `A` $\rightarrow$ `B` specifies the number of transactions in $\mathcal{H}$ for which items in both, `A` and `B`, change. More formally, the frequency ϕ is defined as

$$\phi(A \rightarrow B) \stackrel{\text{def}}{=} |T \in \mathcal{H} : A \cup B \subseteq T| \tag{1}$$

[1]https://doi.org/10.5281/zenodo.8279433

Since the number of transactions can differ, and thus influences the frequency, this measure is usually normalized.

2) Support: The *support* of a rule `A` $\rightarrow$ `B` is the frequency of a rule divided by the number of transactions in the history. More formally, the support σ is defined as

$$\sigma(A \rightarrow B) \stackrel{\text{def}}{=} \frac{\phi(A \rightarrow B)}{|\mathcal{H}|} \tag{2}$$

Obviously, a rule with higher support is more likely to hold true. In contrast, rules with low support indicate events that occur rarely. Usually, a minimum threshold for the support is defined to filter out uninteresting association rules. Please note that due to the many changes typically employed to a software system which touch completely different pieces of code, the support for association rules on commits is usually much lower compared to the well-established use case of cart analysis.

3) Confidence: The *confidence* of a rule is a measure that describes the reliability of the inference made by that rule. It is defined as the frequency of the rule divided by the number of transactions that contain the antecedent. More formally, we can define the confidence as

$$\kappa(A \rightarrow B) \stackrel{\text{def}}{=} \frac{\phi(A \rightarrow B)}{|T \in \mathcal{H} : A \subseteq T|} \tag{3}$$

As we can see, the confidence measures the ratio of transactions in which `A` and `B` are present to the number of transactions in which `A` is present. In other words, the higher the confidence for a rule `A` $\rightarrow$ `B`, the more likely it is that if `A` changes `B` also changes.

III. Feature Co-Change Analysis

In this section, we present our technique for aggregating and analyzing data for extracting implicit feature dependencies. We show our overall, two-staged workflow in Figure 1 and provide more details on these stages and the individual steps in the following. Please note that we report on the actual implementation of our prototype including testing and consequent optimizations in Section IV. Within this section, we focus more in the conceptual design of our technique, which could be adapted for other programming languages and variability mechanisms than C and its preprocessor.

A. Data Aggregation

In the first stage of our technique (cf. Figure 1), we are mainly collecting data by analyzing commits of a repository, filtering out the relevant feature information, and exporting the remaining data to make it available for the second stage. Initially, we need to specify a (Git) repository containing the system we want to analyze. As our analysis is targeted at the C preprocessor as variability mechanism, the repository should contain a system written in C. Moreover, our technique requires a commit hash to specify a starting point for our analysis. Afterwards, our technique executes the following three steps for every commit in the repository.

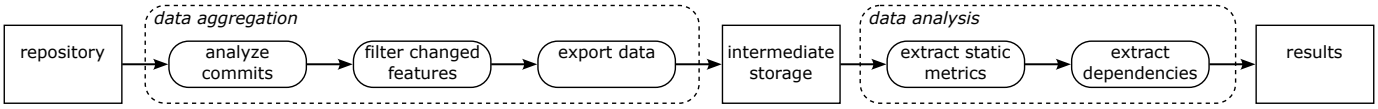

Fig. 1. Workflow of our feature co-change analysis.

1) Analyze Commits: In this step, we access the source code associated with a commit and analyze that code regarding the occurrence of features. Specifically, we are interested in identifying what features exist and what code locations correspond to what features (i.e., the C preprocessor annotations represent markers of feature locations). We require this information to map changes to specific parts of the source (i.e., within annotations) to the right features (i.e., specified in the respective annotations). To this end, we employ the tool CPPSTATS[2] [15], [29], [30] that, among others, provides a list of features and their locations in the source files of a C system. After this step, we have a list of features together with their locations (i.e., file name and start/end line) for the analyzed files in the given commit, which is the input for the next step.

2) Filter Changed Features: The main purpose of this step is to identify changed lines of feature code. To this end, we use two pieces of information. First, we take the change information for each file from the version-control data, which exposes details about changed and added lines of code for each file. Second, we use the information we obtained in the previous step, specifically the list of features and their locations. We combine both pieces of information by mapping them to each other, thereby identifying those lines of feature code that have been changed within a commit. In other words: for each modified line of code we check whether this line is also part of a feature or not. If so, we keep this information, otherwise we discard the change information. As a result of this step, we obtain a list of features associated with every line of code of this feature that has been changed between the last and the current commit.

3) Export Data: As the last step of this stage, we store the extracted and aggregated data from the previous steps. We do this in two different ways: Primarily, we store all pieces of information about the features (i.e., their location and changed lines of code) in a database that we can later query in the analysis stage. Additionally, we store this information locally in a CSV file. We mainly do this to store a backup in case the database is not accessible for whatever reason, particularly for testing our workflow and the pipelines we implemented. Then, we use the stored information within the analysis stage, which we explain next.

B. Data Analysis

In the second stage of our technique, we make use of the previously collected data to

(1) characterize the evolution of features via metrics and
(2) analyze this evolution with respect to feature co-changes to identify implicit feature dependencies.

[2]https://github.com/clhunsen/cppstats/

For the former, we collect several metrics that allow us to quantify how much and how often features have changed in a series of commits. We provide an overview of these metrics in Table I. Specifically, we are interested in feature-related (e.g., number of features, how often the features changed) and commit-related metrics (e.g., commits that change multiple features, number of commits with feature changes). Based on these metrics, we gain initial insights regarding the evolution of each system and can compare different systems with respect to questions, such as *how many of the features have changed* or *how many commits do really affect features?*

However, our main focus in this stage is on the second step of analyzing feature co-changes to extract implicit and evolutionary feature dependencies. To this end, we employ association rule mining as introduced by Agrawal et al. [2] and explained in Section II. Since we apply association rule mining in the context of co-evolving features, we make the following propositions regarding the terms used for this step. First, the *items* we consider are the *features* we extracted during the data aggregation stage. Second, the *transactions* considered within our technique are *feature sets*, that is all features that change together within a certain commit.

Given these propositions, we use the apriori algorithm for association rule mining proposed by Agrawal et al. [2] as follows. First, we create a list of all feature sets based on the data we collected from the repository mining and data aggregation stage. Since a certain feature can change in multiple places within a single commit, the corresponding feature set may contain this feature multiple times. To avoid rules like `feature A` $\rightarrow$ `feature A`, we remove all duplicate features within the feature sets. Then, we compute the support for each feature set. Given a certain support threshold (specified on a concrete instance of the algorithm), this allows to discard less meaningful or insignificant features sets that are below that threshold. For the remaining feature sets, we compute all possible association rules. Subsequently, we calculate the confidence for each of the generated rules. Again, this allows to filter out meaningless rules that are below a minimum confidence threshold. Eventually, we obtain a list of rules that are above the specified support and confidence, and thus should represent the implicit feature dependencies we aim to extract.

IV. Analyzing Co-Evolution in C Systems

In this section, we report a multi-case study [52] in which we employed our technique on five C systems and the C preprocessor as one concrete variability mechanism. We first introduce our research questions and the subject systems we used. Then, we present details on our research prototype, the conduct of our study, and on the challenges we experienced.

TABLE I
OVERVIEW OF THE METRICS WE EXTRACT IN THE DATA ANALYSIS STAGE.

feature-related metrics	**commit-related metrics**
total number of features	total number of commits with feature changes
number of features changed only *once*	number of commits with changes to only one feature (*single-feature commits*)
number of features changed more than once	number of commits with changes in multiple features (*multi-feature commits*)

A. Research Questions

As we summarize in Section VI, most existing research on configurable features has focused on analyzing snapshots (i.e., one version) of the respective systems. This allows to obtain more in-depth insights into the general structure of the source code and how features are implemented, tangled, scattered, and explicitly depending on each other. However, such studies have not been concerned with analyzing the version-control data to identify feature co-evolution and derive implicit dependencies. To assess to what extent such feature co-evolution occurs and represents implicit feature dependencies, we defined two research questions (RQs):

RQ$_1$ *How frequently do co-changes between features occur?* Even if features are made explicit via C preprocessor annotations, this does not mean that two ore more features are changed together between two versions of the same system. In fact, if the features were ideal cohesive units, we could assume that each feature could be changed in complete isolation. Of course, deviations from this ideal scenario occur in practice, and we investigated how frequently such deviations occur. Frequent co-changes of features may hint at design flaws that practitioners may want to resolve, particularly if they are not represented in the source code via preprocessor annotations.

RQ$_2$ *To what extent do feature co-changes reveal implicit feature dependencies?* If feature co-changes occur between versions of a system, these may imply association rules that our technique can mine. So, after understanding to what extent co-changes occur, we next aimed to assess whether these co-changes were logically linked (i.e., implicit dependencies) or happened just by chance. Addressing this research question, we aimed to understand whether feature co-changes are due to implicit dependencies between the features. Through an in-depth analysis, we aimed to provide insights whether we can identify actual dependencies or not based on the extracted association rules.

By addressing these two research questions, we aim to show that our technique works as intended and can provide useful insights on a software system's structure, feature co-changes, and (implicit) feature dependencies.

B. Subject Systems

For the prototype implementation of our technique, we use the tool CPPSTATS to analyze C code. We chose this tool, because the C preprocessor is the predominant variability mechanism in practice, enabling us to perform analyses on highly diverse software systems. In the end, we picked five C-based systems (aiming for different properties) that have been studied in previous works on the C preprocessor [6], [8], [29], [31], [43]. We display an overview of these five systems in Table II.

As we can see, the systems are from different domains and differ in size regarding the lines of code as well as the number of commits. We aimed for differences to reduce potential biases in our case study, which may otherwise occur if all systems are from the same domain or of similar sizes. Not surprisingly, since these are well-established open-source systems, all of them were still active when we conducted our analysis. While the systems' first commits have been committed at different points in time, we analyzed more than 15 years and 3,000 commits for each system. Please note that the change history of the Linux kernel before 2005 has not been migrated to Git, which is why we do not analyze the previous history. Still, most of the systems involve commits that are from before 2005, the year Git was released.[3] In contrast to the Linux kernel, these systems migrated their full version history into Git, which is why we could analyze these.

C. Prototype

We implemented our technique as a prototype based on containerization (i.e., Docker containers). In particular, we modularized the different logical steps of our technique (cf. Section III) into separate containers (e.g., the commit analysis and filtering of changed features are implemented in independent containers). We decided to employ this design to achieve several benefits. First, separating the individual steps into containers allows for a simple yet structured and efficient overall management of the different steps. Second, by using containers we can more easily parallelize tasks, if needed. For instance, for the commit analysis, we could start multiple containers at the same time, with each container analyzing a certain time range of the commit history. Finally, using containers greatly improves dependency management, which helped us deal with the several tools we use. Specifically, this was helpful for CPPSTATS, which relies on quite old or different library versions compared to our own code (e.g., requiring Python 2.7). Besides using containerization, we control the overall process of our technique via several Python scripts. Moreover, we use PostgreSQL as database for storing the results of our data aggregation, and we use an open-source implementation of the apriori association rule mining algorithm proposed by Agrawal et al. [2].[4] In the next two sections, we

[3] https://www.linuxjournal.com/content/git-origin-story

[4] github.com/eMAGTechLabs/go-apriori

TABLE II
OVERVIEW OF OUR SUBJECT SYSTEMS.

system	domain	# lines of code	# commits	covered period	url
OpenVPN	network service	79,216	3,176	26.09.2005–17.03.2022	https://github.com/OpenVPN/openvpn
libxml2	programming library	201,757	5,312	24.07.1999–03.04.2022	https://gitlab.gnome.org/GNOME/libxml2
OpenLDAP	network service	291,726	23,985	09.08.1998–11.04.2022	https://git.openldap.org/openldap/openldap/
postgres	database system	872,412	53,695	09.07.1996–22.03.2022	https://github.com/postgres/postgres/
Linux kernel	operating system	16,143,672	1,090,089	17.04.2005–02.04.2022	https://github.com/torvalds/linux

describe the concrete instrumentation of our prototype as well as its testing and consequent improvements throughout our multi-case study in more detail.

D. Conduct

We employed our technique on each subject systems individually, following the steps we describe in Section III.

1) Data Aggregation: In the first stage, we extracted for each commit of each system the features that were changed. To process the large number of commits, we executed all steps (cf. Figure 1) automatically. For this purpose, we implemented the prototype of our technique, which we tested via a command line interface that we connected to the continuous integration platform Woodpecker 10. This platform allowed us to run multiple Docker containers in succession while working on a shared directory on a local system. After testing that our technique worked as intended, we implemented a workbalancer that takes a repository and a list of commits to process as input. Then, the workbalancer executes our implemented technique in predefined Docker images. Our implementation of the workbalancer allowed us to use parallel executions, which was not possible in the standard Woodpecker.

2) Data Analysis – Static Metrics: For the first analysis step, we used a Python script to extract static metrics regarding how often features occur. To this end, we first removed multiple mentions of the same feature in a commit by using a Python set (i.e., each feature is only mentioned once even if it was changed multiple times within a commit). This allowed us to order commits based on how many different features these touch, and to distinguish between commits that changed only one ore multiple features—which we needed to know for the association rule mining. Finally, we reordered the data so that each feature points to the commits in which it was changed (instead of commits pointing to the features). As a result, we could more easily identify how many changes impact which features and collect our metrics (cf. Table I).

3) Data Analysis – Dependencies: For the second analysis step, we integrated the apriori algorithm to extract association rules for co-changing features. This algorithm requires two inputs: the minimum support and minimum confidence (cf. Section II). Both of these parameters are typically specified in percent. As the minimum confidence, we used 75 % for our case study, meaning that for a rule `A` $\rightarrow$ `B` to be considered relevant, 75 % of all commits that change feature `A` also must change feature `B`. We use the value of 75 % as a middle ground between typically used ones, such as a very conservative value like 90 % and a very aggressive value like 50 %. With the specified 75 %, we argue that the mined rules should properly represent relevant feature co-evolution, which would also be feasible for a recommender system without annoying developers with too many irrelevant recommendations.

Deviating from the standard, we defined the minimum support as an absolute number instead of a percentage for our case study. We decided to adapt this parameter, because association rule mining has been proposed for transactional data in supermarkets. In contrast, we are concerned with version-control data and features. While structurally representing the same data (i.e., item sets versus feature sets), the frequency of feature sets is lower compared to the original use case, due to the varying number of features that are changed in each commit. Specifically, each data entry in the transactional data of supermarkets typically used for association rule mining represents a relevant item set, but not all commits in the version-control system represent a feature co-change. Even more problematic, a (configurable) feature constantly changing in each commit would likely indicate a problem with the system or feature development, but this can occur regularly in the supermarket item sets for which association rule mining has been proposed (e.g., butter). So, instead of using a percentage for the minimum support (which could be too high depending on how often a feature occurs throughout all commits), we defined this parameter as 15 commits out of all feature commits (i.e., commits that change features). This means that to become part of an association rule, the involved feature must occur in at least 15 feature commits, which is a smaller fraction compared to all commits. During test runs, we found that this value is set high enough to filter out random co-changes, while not simply discarding all co-changes as occurring by chance.

4) Data Interpretation: As described, we built on metrics and their interpretation to elicit insights regarding feature co-evolution (**RQ$_1$**). So, our analysis and interpretation regarding these results builds on more quantitative data. In contrast, it does not make sense to perform a pure quantitative interpretation of the association rules (**RQ$_2$**), for instance, regarding how many rules our technique identified. For this reason, we performed an additional manual inspection of all association rules to provide insights into their properties and what they reveal about the individual systems.

E. Testing, Challenges, and Improvements

While engineering and testing the prototype of our technique, we experienced several engineering challenges and implemented multiple improvements, particularly to increase

the performance of the prototype. For instance, the first version of our prototype already built on Woodpecker to execute our analysis pipeline, using a Bash script and a list of commits as input. Unfortunately, we experienced that the execution time, particularly of CPPSTATS, was rather long. To solve this problem, we implemented a solution for parellizing the analyses of our technique. Our idea was to distribute the commits across the available CPU cores and to execute our script in parallel to analyze each commit individually. Unfortunately, this did not work, because Woodpecker assigns the same name to each container and Docker does not allow to execute multiple containers with the same name. We aimed to overcome this problem by implementing our workbalancer, which resolves naming collisions by adding the analyzed commit's unique hash to the Docker container's name.

After implementing this solution, we recognized that it could also help us resolve another problem we had with Woodpecker. Specifically, if a command inside a container fails and the execution stops, the container is not deleted automatically since the user would likely want to assess the failure. Unfortunately, this also meant that our complete pipeline was put to halt, since again the Docker container's name was already used, and thus we could not create a new container in the next (or same) run without manual intervention. To resolve this problem, the workbalancer checks whether a container with the same name already exits before creating a new one. If this is the case, we delete the old container, but print the logs to the console to enable us to analyze any further issues that may arise.

We performed tests and manual inspections of the results to identify and resolve bugs in our prototype. One particular problem arose from using CPPSTATS: Feature annotations can involve braces to build complex expressions on feature dependencies. Unfortunately, one of the expressions in OpenLDAP lacked a closing brace, which caused CPPSTATS to parse the expression together with the following code. We reported this issue to the CPPSTATS developers, but as a quick fix for our study (and since it was only one file), we manually processed the file regarding the involved feature changes.

Even though the previous means improved the performance of our technique and we checked its behavior through testing, we still noticed that the execution time of CPPSTATS itself was sometimes very long. For some larger commits of OpenVPN, CPPSTATS alone required two minutes, which made it infeasible to execute the analysis for several thousands of commits. To reduce the time needed by CPPSTATS, we first limited its analysis to C files only, since these include the relevant feature annotations we are concerned with. This also reduced the number of files we had to copy to create the right folder structure for CPPSTATS. Moreover, we implemented our technique to work directly in the working memory (tmpfs13) instead of on the local filesystem.

V. Results & Discussion

In this section, we first report, discuss, and summarize the results for each of our two research questions individually to derive consequent answers. Afterwards, we discuss potential threats to the validity of our work.

A. *RQ$_1$: Feature Co-Changes*

In Table III, we provide a statistical overview regarding the feature changes in each of our subject systems. Specifically, we compare for each system the number and ratios of feature commits out of all commits we analyzed, how many of these commits change a single or multiple features, and how many distinct features we found that changed once or multiple times.

1) Results: We can see considerable differences between the subject systems regarding all properties. For instance, OpenVPN has the highest share of feature commits with roughly 52.9 %, while the Linux kernel has the lowest share with roughly 10.6 %. This is in contrast to the Linux Kernel involving the most distinct features (18,140) by a large margin (postgres has 1,340). In fact, we can see that while the number of commits (and lines of code, cf. Table II) increases across the systems, the ratio of commits changing features actually decreases—even though the number of distinct features increases. This observation aligns with previous research, which has shown that larger C systems are often more configurable (i.e., involve more features), but that the ratio of source code related to the features (and thus changes to these features' code) decreases [15], [29], [31].

Furthermore, we can see that the ratio of commits that change multiple features at once decreases for higher numbers of feature commits and distinct features. Multiple features being changed in one commit can happen for various reasons. The fundamental reasons are

1) that different features may be implemented in separate files changed within one commit;
2) that separate features are changed within the same file of one commit; or
3) that the features explicitly depend on each other with respect to the changed source code.

For instance, the latter case can be caused by nested features (the code of one feature is encapsulated by another feature) or by complex feature expressions that combine multiple features through logical operators. While this reason connects to explicit feature dependencies, it is not directly visible that the features co-changed during a system's evolution—which still requires an analysis of the relevant source code and commits, and thus still represents implicit feature dependencies. The former two reasons do represent implicit feature dependencies, because the features are changed together but there is no explicit connection between them in the source code at all.

Lastly, we can see that quite a lot of the features we identified have been changed only once, which can have various reasons. For example, these features could be very stable, rarely used, renamed, or have been removed at some point (e.g., making the feature code mandatory without touching the actual feature code). Still, investigating features that have changed only once is out of the scope of our paper, but constitutes interesting future work. More importantly, we can see in Table II and Table III that the systems represent a proper sample for our study, since

TABLE III
RESULTS OF OUR FEATURE CO-CHANGE ANALYSIS FOR EACH SUBJECT SYSTEM.

		feature commits					distinct features		
system	**# commits**	**#**	**%**	**# sing. feat.**	**# mult. feat.**	**% mult. feat.**	**total**	**ch. once**	**ch. mult.**
OpenVPN	3,176	1,681	52.9	530	1,151	68.5	425	107	318
libxml2	5,312	2,632	49.5	1,411	1,221	46.4	509	158	351
OpenLDAP	23,985	7,762	32.4	5,020	2,742	35.3	1,300	426	874
postgres	53,695	6,399	11.9	4,026	2,373	37.1	1,340	311	1,029
Linux kernel	1,090,089	115,558	10.6	84,566	30,992	26.8	18,140	7,395	10,745

sing./mult. feat.: commit involves changes to a **sing**le or **mult**iple distinct **feat**ures
ch. once/mult.: the feature has been **ch**anged **once** or **mult**iple times in commits

they have diverse properties and involve extensive co-evolution of features—indicated by the numbers of commits changing multiple features and features changing multiple times.

2) Discussion: We can see that the ratio between co-change commits and all feature commits varies across the different subject systems, but does not drop below 25 %. This implies that co-changes of configurable features are likely to occur in other software systems that use the C preprocessor as well. The amount of co-changes may vary and the number of feature commits is no indicator for the number of commits changing multiple features. For instance, the Linux kernel has the lowest number of feature commits, since only 10.6 % of all commits involve feature changes, and also the lowest number of feature co-change commits with 25 % out of the feature commits. In contrast, a smaller system like OpenVPN has feature changes in nearly 53 % of its commits, with 68 % of these containing feature co-changes. This implies a wide spreading of the ratio of feature co-changes across different systems. By inspecting and comparing all systems, we can see a trend of larger systems (in terms of lines of code and commits) apparently involving fewer feature co-changes. One possible explanation for this could be a decreasing amount of feature code that has been found for such systems in related studies, but more in-depth analyses are needed to confirm this assumption.

RQ$_1$: Feature Co-Changes

• Feature commits with co-changes occur frequently in configurable systems that use the C preprocessor. • Configurable C systems with fewer lines of code and commits seem to involve a higher ratio of feature co-changes compared to C systems that are larger along these dimensions. •

3) Summary: Overall, we argue that our technique can provide useful insights into the co-evolution of configurable features, which seem to occur frequently for systems that use the C preprocessor. In turn, this means that it can be helpful to mine association rules to reveal implicit feature dependencies that are not specified in the source code. For researchers, our insights open new research directions on studying the co-evolution of configurable software systems, such as analyzing whether the co-changes are intentional or not, how developers document or deal with these changes, or how to untangle the different feature changes. To this end, we provide tooling and first insights regarding the extent to which co-evolution occurs in those systems, not only within commits that touch the often measured tangling degree (i.e., features that are part of explicit dependencies) [22], [29], [31]. For practitioners, we provide a technique that enables them to identify co-changes, which allows them to identify implicit feature dependencies (e.g., to document these), potential design flaws (e.g., unintended feature dependencies), or potential shortcomings in their processes (e.g., putting independent feature changes into one commit). Our insights on implicit feature dependencies in the next section advance into this research direction and provide additional tooling for practitioners.

B. *RQ$_2$: Implicit Feature Dependencies*

In Table IV, we display an overview of the total numbers of association rules we mined for each system. We distinguish between simple and complex rules. The former type involves exactly two features (e.g., `A` $\rightarrow$ `B`), while the latter type refers to cases in which changes to multiple features imply a change to another one (e.g., `[A, B, C]` $\rightarrow$ `D`). Next, we first analyze the general data before going into a more qualitative analysis of each individual systems.

1) Results: As we can see in Table IV, the number of association rules we identified varies strongly. In particular, there is an interesting gap regarding the number of distinct features compared to the number of rules. Specifically, postreges and OpenLDAP both have around 1,300 distinct features and 6,399 as well as 7,762 feature commits, but only 209 and 108 association rules, respectively. In contrast, libxml2 and OpenVPN are considerably smaller regarding their distinct features and commits, but for both we mined around 6,000 association rules. The Linux kernel as by far the largest system is somewhere in the middle with 2,477 association rules. In the last three columns, we show the average confidence, support, and minimum support regarding the association rules. We remark that we also extracted association rules for feature dependencies that are explicit in preprocessor annotations (e.g., code in nested features). These still are implicit dependencies caused by co-changes, and seeing that the two types of dependencies match has improved our trust that the association rule mining yields reasonable and useful insights—particularly for the remaining dependencies that are not represented in preprocessor annotations. We do not provide concrete numbers regarding how many implicit and explicit feature dependencies match, because this is out of scope for this work and would

TABLE IV
OVERVIEW OF THE ASSOCIATION RULES WE EXTRACTED FOR THE ANALYZED SUBJECT SYSTEMS.

		association rules					
system	**# distinct features**	**#**	**simple**	**complex**	**avg. confidence (%)**	**avg. support (%)**	**min support (%)**
OpenVPN	425	6,356	49	6,307	98	3.93	0.89
libxml2	509	5,896	11	5,885	95	0.64	0.57
OpenLDAP	1,300	108	44	64	97	0.31	0.19
postgres	1,340	209	42	167	97	0.33	0.23
Linux kernel	18,140	2,477	280	2,197	97	0.016	0.013

require a different analysis to capture and match the feature conditions specified within annotations. Next, we report our insights of inspecting the association rules of the individual systems to shed light into the differences we can observe.

For **OpenVPN**, we extracted 6,356 association rules, which are the most even though the system is the smallest. Our technique shows an average confidence in those rules of 98 % and average support of 3.39 % (minimum support of 0.89 %). A majority of the rules for OpenVPN contain features with the prefix `TARGET` in their name. During our manual analysis, we found that this prefix represents the target operating system, such as Android, Windows, or Darwin MacOS. Moreover, most rules point from one target operation system to another, which indicates that changes to OpenVPN for one target operating system also imply changes for other operating systems. This makes sense, since OpenVPN is designed for different operating systems with their own interfaces, meaning that many features must be adapted whenever behavior of the system changes.

For **libxml2**, we extracted 5,885 assoication rules with an average confidence of 95 % and an average support of 0.64 % (minimum support 0.57 %). While OpenVPN involved many features with the prefix `TARGET`, libxml2 involves a majority of features that contain the String `LIBXML`. Consequently, only 10 (antecedent, on the left side) and 13 (consequent, on the right side) out of the 5,896 association rules do not contain features comprising this String. So, most features are represented by the system's name and an additional descriptive String (e.g., `LIBXML_CATALOG_ENABLED`). Based on our manual analysis, this seems to be an internal naming convention. Out of the rules without such features, most contain some subset of `__CYGWIN__`, `_WIN32`, and `__DJGPP__`. `CYGWIN` is a tool for developers to port their Linux software to Windows. `DJGPP` allows to create executables for Windows and `WIN32` directly refers to this operating system. So, all three features are directly related to a specific operating system, and the names alone indicate a strong connection between them. Out of all association rules, 1,132 (19.2 %) contain the feature `LIBXML_DEBUG_ENABLED` on either side of the rule. This suggests that changes to the system's features often require changes to the debugging part, too. However, we argue that such implicit rules are not really problematic, since debugging is used for internal testing and most developers should be aware about the debugging feature.

For **OpenLDAP** we found only 108 association rules with an average confidence of 97% and an average support of 0.31% (minimum support 0.19%). In contrast to the previous two systems, OpenLDAP does neither use semantic prefixes extenisvely (e.g., `TARGET` for OpenVPN) nor the system's name. Still, in the manual inspection of the rules, the relations between the features become clear. For example, changes in `LDBM_USE_BTREE` and `LDBM_USE_DBHASH` can be found in rules together. `LDBM` seems to be a component related to the database management of OpenLDAP, which also involves different indexing structures that can be enabled or disabled (e.g., `BTREE`, `DBHASH`).

For **postgres**, we find similarly few association rules (209) compared to its size, which have an average confidence of 97 % and an average support of 0.33 % (minimum support 0.23 %). The rules are also similar to those for OpenLDAP, using neither an identifying prefix for specific sets of features nor the system's name. Still, we did again identify logical connections within the association rules. For example, some rules combine `HAVE_TM_ZONE` and `HAVE_INT_TIMEZONE`, which both refer to time zones. So, we consider it logical that these two feature change together, even though this is not made explicit. Identical to libxml2, `__CYGWIN__` and `WIN32` reappear in the same rules.

For the **Linux kernel**, we extracted 2,477 rules, which is much less compared to the much smaller OpenVPN and libxml2, even though the Linux kernel involves around 35 times the number of distinct features. Still, the average confidence in the rules remains at 97 % and the average support is very low with 0.016 % (minimum support 0.013 %). This low support is not surprising, as the number of feature commits and distinct features is far greater than in the other systems. Within the Linux kernel, many rules contain features with the prefix `CONFIG`, which marks different configuration options for building user defined variants. For instance, such features include Bluetooth options for the kernel used in Ubuntu Core for the Raspberry Pi10. The consequent association rules contain changes between different components that seem logical. As an example, changes to features that are located in different parts of the IPv6 component are interconnected, and we observed similar cases for configuration options related to keyboard support or the system architecture. These dependencies are not explicit, but our association rules show that such logically connected features are in fact changed together, and thus seem implicitly linked.

2) Discussion: The frequent co-changes we identified before hinted at implicit feature dependencies, and our association

rule mining revealed actual implicit dependencies that are represented in the version-control data. Across all systems, the average confidence for the rules is at least 95 % while the average support is below 4 % (below 1 % for four of the five systems). The low support is normal for our use case. A support of 50 % would mean that the same group of features is changed in 50 % of the feature commits, which would indicate a very change-prone (e.g., error-prone or "god") feature—which we would not consider normal for most software systems. In fact, previous research [45] indicates that few files are typically changed within a commit. So, having a high level of support for an association rule would be in conflict with such findings, since it would imply that the same set of features changed very often while most others remained untouched. Our insights improve our confidence that the association rule mining works as intended and can be a helpful means to uncover implicit feature dependencies.

While the number of association rules we could extract from the systems varies greatly, there is no apparent trend. However, for similar numbers of distinct features, the number of association rules is also similar. This indicates that there may still be a relation regarding the size or complexity of a system and the number of implicit feature dependencies, but more research is needed to shed light into this issue. During our manual analysis of the features, we could observe that most rules connect features that seem to logically belong together. So, features may be split into smaller units to customize more specific behavior. In contrast, for some other rules we could not identify logical connections between the involved features. We argue that these may be the most interesting implicit dependencies for developers and also researchers, since it is unclear where the dependency stems from, and thus such dependencies may imply design flaws.

On a final note, we have to put the high numbers for complex rules into context. During our manual analysis, we noted that such complex rules are often different combinations of the same features. Please note that we did not remove such "duplicates," because while the support stays the same (since it is based on the number of occurrences) the confidence can vary. This situation leads to higher values regarding the number of complex, and thus the total amount of, association rules. In future work, further processing can be used to consolidate these rules at the loss of confidence for the individual rules. However, since the confidence is an important value for a change recommendation system, this has to be designed carefully to propose a practical recommender system.

RQ$_2$: Implicit Feature Dependencies

• We can extract association rules from feature co-changes. • The consequent implicit feature dependencies seem reliable, since they involve many logical connections. • There are cases that demand for more in-depth investigations, since the implicit dependency is not obvious and may indicate quality flaws in a system. •

3) Summary: Overall, we argue that our technique for mining association rules is a helpful means for extracting implicit feature dependencies from feature co-changes. In particular, the association rules imply several logical dependencies that are not explicit in the source code, while also revealing some implicit dependencies that are not intuitive. Investigating these dependencies in more detail is a promising research direction to distinguish what constitutes an implicit feature dependency and how it impacts the comprehension or quality of a software system, for instance, whether implicit dependencies make it more likely that developers miss to update all relevant pieces of source code. Similarly, our technique can help practitioners identify dependencies that they may not be aware of or that are unintended, helping them to improve their understanding of the system and improve its quality.

C. Threats to Validity

In this paper, we proposed a technique for using association rule mining to identify feature dependencies that are implicit due to the features' co-evolution. While our technique works as intended, it can only indicate likely dependencies, but whether these are actually logical is a different question. We mitigated this internal threat by performing an in-depth manual analysis of all change rules and inspecting whether these are logical or may even match with dependencies explicitly specified in the source code of the respective system. Still, it remains a threat to the internal validity that we cannot judge which of the rules are meaningful. So, in future work, it would be helpful to combine our technique with other information sources like commit messages or interviews to provide a better understanding of the mined association rules themselves. Nonetheless, we can reliably analyze whether and where co-changes of features occur as well as whether these represent recurring patterns, which was our goal with this research.

As we described, we analyzed modified and added lines of code only. So, we neglected deleted lines, primarily because their extraction would have doubled the processing time and requires multiple additional steps in the data aggregation stage. Specifically, the line numbers of deleted code correspond to the file states in the previous commit. For every deletion, we would need to analyze the preceding commit and to precisely locate the deleted feature code in the current commit. While this remains an internal threat, this case is relevant for pure deletions exclusively. However, this impacts all features and systems equally and we still obtained a large number of co-changes for the features of each system. Consequently, we argue that this threat does not invalidate our results.

Regarding the external validity, we are aware that we used only five subject systems, all open-source, implemented in the same programming language, and using the same variability mechanism. As a consequence, our results are likely not entirely transferable to all other configurable software systems. This is why we emphasized that our study and findings correspond primarily to C systems, even though our idea and the association rule mining can be adopted for other variability mechanisms as well. We chose C with the C preprocessor due to their widespread use in practice and open source, making them the most relevant implementation techniques for

developing configurable systems Moreover, research has shown that industrial and open-source C systems are comparable [15], which is why open-source systems are feasible subjects. Seeing that some of our findings correspond to other studies improves our trust in the validity of our work. So, while this external threat remains, we argue that it should neither impair our insights on C systems nor invalidate our contribution of extracting implicit feature dependencies.

VI. Related Work

How developers implement configuration options within software systems has been extensively studied in different contexts [13], [29], [36], [46]. Most relevant with respect to our work in this paper are analyses of C preprocessor annotations and their properties as well as evolution. Due to its widespread use in industry and open-source software, the C preprocessor has actually been discussed and researched for a long time [15], [28], [48]. In particular, researchers have analyzed different properties of C preprocessor annotations like their distribution, scattering, or tangling across software systems to understand these properties' impact on program comprehension, code quality, and bugs [1], [6], [7], [10], [23], [25], [26], [29], [30], [32], [34], [35], [37], [43], [47]. Many of such studies are (indirectly) concerned with developers' comprehension of features and their dependencies, often arguing in favor of refactoring C preprocessor annotations towards a disciplined use (i.e., annotating complete lines only instead of individual words or even tokens). In contrast to such works, we are not concerned with program comprehension, bugs, or code quality, but with extracting implicit feature dependencies—which of course can relate to all of such concerns. Additionally, we employ an evolutionary analysis across all relevant system versions via the version-control system instead of a static analysis of one specific release.

Other researchers have proposed techniques and tools for analyzing variability and particularly C preprocessor annotations. For instance, TypeChef [21], PCLocator [27], or SuperC [14], among many others [22], have been proposed as static analyses for such annotations and the individual variants that can be derived. However, these techniques are similar to CPPSTATS in that they themselves do not analyze the evolution of feature changes, but the preprocessor annotations in one specific system version at a time. Other techniques are intended to extract feature dependencies or constraints from source code and other artifacts [19], [20]. For instance, Nadi et al. [38], [39] have developed a static analysis that is able to extract around 28 % of existing explicit dependencies from the C preprocessor. Similarly, Oliveira et al. [41] have extracted structural feature dependencies from different releases of C systems. The authors found that roughly 25 % of the explicit dependencies are changed between two releases. Rodrigues et al. [44] have empirically studied what types of explicit feature interactions occur to what extent in C preprocessor systems. However, they do not actually investigate means for extracting such dependencies. Ludwig et al. [31] have analyzed the meaning of different C preprocessor annotations (e.g., negations) and their impact on variability-related metrics. Their results indicate that not all preprocessor annotations can be clearly assigned to specific features. Fenske et al. [9] have analyzed the change-proneness of code that involves C preprocessor annotations over time. Their findings suggest that code with annotations is changed more often throughout a system's evolution, but if corrected for the size of those files these findings diminish. Fischer [11] has analyzed the evolution of configuration options across four versions of Bugzilla. However, Bugzilla does not use the C preprocessor and the analysis focuses on static changes only (e.g., configuration options added). While closely related to our work, we have a different goal and follow another idea compared to this existing research by analyzing feature co-changes in version-control data to extract implicit feature dependencies. In fact, we can see that integrating our technique with existing ones may yield great synergies to extract feature dependencies even more reliably and provide novel analyses regarding the evolution of configurable systems.

VII. Conclusion

In this paper, we have proposed a technique that extracts feature co-changes from configurable software systems' version histories and applies association rule mining to elicit implicit dependencies. We implemented our technique in a prototype that enabled us to analyze five open-source C systems regarding potential implicit dependencies. Our results indicate that our technique works as intended and can help researchers as well as practitioners identify implicit feature dependencies that are otherwise hidden in a system's evolution history. By manually inspecting the association rules, we learned that they indicate logical dependencies sometimes already represented in explicit preprocessor annotations. This underpins the value of our idea, but we see the primary benefits in identifying fully implicit feature dependencies that are otherwise missed and that may indicate quality or process issues.

As a consequence, we plan to improve our technique, incorporate further analyses, and conduct empirical studies to shed more light onto implicit feature dependencies. In particular, it would be interesting to directly compare explicit and implicit feature dependencies and to what extent the former differ from than latter one. Moreover, we envision a more detailed analysis on how developers deal with either of the two, for instance, regarding their awareness or documentation preferences for either. Also, it could be interesting to analyze how much the code changes of implicitly dependent features are scattered across the code base. Intuitively, we would assume that a high scattering degree makes it more difficult to know about all the change locations, which in turn may imply that a recommendation mechanism, based on our analysis, can be beneficial for developers Finally, larger studies with different metrics would be great to fully validate that the extracted rules are correct and meaningful.

References

[1] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. de Medeiros Ribeiro, and A. Wąsowski, "Variability Bugs in Highly Configurable Systems: A

Qualitative Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 10:1–34, 2018.
[2] R. Agrawal, T. Imieliński, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," in *International Conference on Management of Data (SIGMOD)*. ACM, 1993, pp. 207–216.
[3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.
[4] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches," in *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2012, pp. 173–182.
[5] E. Engström and P. Runeson, "Software Product Line Testing - A Systematic Mapping Study," *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.
[6] W. Fenske, J. Krüger, M. Kanyshkova, and S. Schulze, "#ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 255–266.
[7] W. Fenske and S. Schulze, "Code Smells Revisited: A Variability Perspective," in *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2015, pp. 3–10.
[8] W. Fenske, S. Schulze, D. Meyer, and G. Saake, "When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 171–180.
[9] W. Fenske, S. Schulze, and G. Saake, "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness," in *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2017, pp. 77–90.
[10] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel, "Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 65–73.
[11] S. Fischer, "A Case Study on the Evolution of Configuration Options of a Highly-Configurable Software System," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 630–635.
[12] H. Gall, K. Hajek, and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," in *International Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 190–198.
[13] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in Software Systems—A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 282–306, 2014.
[14] P. Gazzillo and R. Grimm, "SuperC: Parsing All of C by Taming the Preprocessor," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 323–334.
[15] 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," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.
[16] H. Kagdi, S. Yusuf, and J. I. Maletic, "Mining Sequences of Changed-Files from Version Histories," in *International Workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 47–53.
[17] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
[18] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, no. 4, pp. 58–65, 2002.
[19] C. Kästner, A. Dreiling, and K. Ostermann, "Variability Mining with LEADT," Philipps University of Marburg, Tech. Rep. BI2011-01, 2011.
[20] ——, "Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 67–82, 2014.
[21] A. Kenner, C. Kästner, S. Haase, and T. Leich, "TypeChef: Toward Type Checking #ifdef Variability in C," in *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2010, pp. 25–32.
[22] C. Kröher, S. El-Sharkawy, and K. Schmid, "KernelHaven – An Experimentation Workbench for Analyzing Software Product Lines," in *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM, 2018, pp. 73–76.
[23] J. Krüger, "Separation of Concerns: Experiences of the Crowd," in *Symposium on Applied Computing (SAC)*. ACM, 2018, pp. 2076–2077.
[24] J. Krüger, G. Çalıklı, T. Berger, T. Leich, and G. Saake, "Effects of Explicit Feature Traceability on Program Comprehension," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 338–349.
[25] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a Better Understanding of Software Features and Their Characteristics," in *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2018, pp. 105–112.
[26] J. Krüger, K. Ludwig, B. Zimmermann, and T. Leich, "Physical Separation of Features: A Survey with CPP Developers," in *Symposium on Applied Computing (SAC)*. ACM, 2018, pp. 2042–2049.
[27] E. Kuiter, S. Krieter, J. Krüger, K. Ludwig, T. Leich, and G. Saake, "PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2018, pp. 284–288.
[28] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef Confirmed Harmful: Promoting Understandable Software Variation," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2011, pp. 143–150.
[29] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.
[30] J. Liebig, C. Kästner, and S. Apel, "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code," in *International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.
[31] K. Ludwig, J. Krüger, and T. Leich, "Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?" in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 218–230.
[32] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The Discipline of Preprocessor-Based Annotations - Does #ifdef TAG n't #endif Matter," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 297–307.
[33] F. Medeiros, C. Kästner, M. de Medeiros Ribeiro, S. Nadi, and R. Gheyi, "The Love/Hate Relationship with the C Preprocessor: An Interview Study," in *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl, 2015, pp. 495–518.
[34] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating Preprocessor-Based Syntax Errors," in *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2013, pp. 75–84.
[35] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca, "Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 453–469, 2018.
[36] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring Differences and Commonalities between Feature Flags and Configuration Options," in *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. ACM, 2020, pp. 201–210.
[37] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, "An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 50–61.
[38] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining Configuration Constraints: Static Analyses and Empirical Results," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 140–151.
[39] ——, "Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, 2015.
[40] D. Nešić, J. Krüger, Ş. Stănciulescu, and T. Berger, "Principles of Feature Modeling," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 62–73.
[41] R. Oliveira, B. Cafeo, and A. Hora, "On the Evolution of Feature Dependencies: An Exploratory Study of Preprocessor-Based Systems," in *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2019, pp. 14:1–9.
[42] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[43] 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," *International Journal on Software & Systems Modeling*, vol. 16, no. 1, pp. 77–96, 2017.

[44] I. Rodrigues, M. de Medeiros Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi, "Assessing Fine-Grained Feature Dependencies," *Information and Software Technology*, vol. 78, pp. 27–52, 2016.

[45] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, and D. Binkley, "Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2016, pp. 201–212.

[46] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2020.

[47] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment," in *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2013, pp. 65–74.

[48] H. Spencer and G. Collyer, "#ifdef Considered Harmful, or Portability Experience With C News," in *USENIX Conference (USENIX)*. USENIX, 1992, pp. 185–197.

[49] M. Svahnberg, J. van Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[50] 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 *European Conference on Computer Systems (EuroSys)*. ACM, 2011, pp. 47–60.

[51] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–45, 2014.

[52] R. K. Yin, *Case Study Research and Applications: Design and Methods*. Sage, 2018.

[53] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[54] B. Zhang, S. Duszynski, and M. Becker, "Variability Mechanisms and Lessons Learned in Practice," in *International Workshop on Conducting Empirical Studies in Industry (CESI)*. ACM, 2016, pp. 14–20.

[55] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.