Detecting and Visualizing Implementation Feature Interactions in Extracted Core Assets of Software Product Line

Hamzeh Eyal Salman[1], Yaqin Al-Ma'aitah[2], Abdelhak-Djamel Seriai[3]
Software Engineering Department, Faculty of Information Technology, Mutah University, Karak, Jordan[1]
Computer Science Department, Faculty of Information Technology, Mutah University, Karak, Jordan[2]
LIRMM Laboratory, University of Montpellier, Montpellier, France[3]

***Abstract*—Recently, software products have played a vital role in our daily lives, having a significant impact on industries and the economy. Software product line engineering is an engineering strategy that allows for the systematic reuse and development of a set of software products simultaneously, rather than just one software product at a time. This strategy mainly relies on features composition to generate multiple new software products. Unwanted feature interactions, where the integration of multiple feature implementations hinders each other, are challenging in this strategy. This leads to performance degradation, and unexpected behaviors may happen. In this article, we propose an approach to detect and visualize all feature interactions early. Our approach depends on an unsupervised clustering technique called *formal concept analysis* to achieve the goal. The effectiveness of the proposed approach is evaluated by applying it to a large and benchmark case study in this domain. The results indicate that the proposed approach effectively detects and visualizes all interacted features. Also, it saves developer efforts for detecting interacted features in a range between 67% and 93%.**

Keywords*—Unwanted feature interaction; core assets; extractive approach; visualization; shared artifacts; implementation dependency

I. INTRODUCTION

Nowadays, software products have played a vital role in our daily lives, having a significant impact on industries and the economy. Software products are always described by their provided features. A feature is often defined as a specific software functionality that offers a service to end users (different from a feature in machine learning), often identified by a name and supplemented with a description [33]. In the context of software development, a feature can be implemented in software by a set of various elements belonging to different levels of abstraction (e.g., source code, requirements, architectural components, etc.) [9]. We refer to this set of elements as *artefacts*. Also, the concept of feature plays a pivotal role in software product line engineering (SPLE) which is an engineering strategy to support the production of a family of software products at the same time distinguished by their provided features [25]. This family is called the software product line (SPL) and it is the final output of SPLE.

SPLs are seldom built from scratch [21]. The most commonly used process to build SPLs is the extractive process [6] [19] [16]. In this process, the implementations of an already existing family of software variants, developed by clone-and-own, are reused to build SPL's core assets [34]. As features are important to build SPLs, the essence of the extraction process is feature location. Locating features in software variants aims to find source code artefacts that implement each feature, which is out of the scope of this article [23]. To build SPL from these extracted features, their implementations must be compatible and integrate with one another. Otherwise, the performance of the generated products will be degraded, and unexpected behavior may occur. This is known as *unwanted feature interactions* [13], and it occurs when multiple feature implementations are combined in a new product, and their behaviors are unexpected even if the implementation of each individual feature is working correctly and independent in their domain. This kind of feature interaction at the source code level is known in the literature as structural interaction or implementation dependency [19] [17]. However, other forms of feature interactions do not manifest as dependencies such as, logical dependency or domain dependency [17]. In this study, the implementation dependency within extracted features of product variants is only addressed as it exists mostly in SPL [18].

The implementations of extracted features from product variants are overlapped in some classes or methods as features interact in software [19] [13] [10]. These overlapped artifacts (shared artefacts) do not represent the core implementations of features but they are added to allow two or more features to work as a combination in their hosted software variants. When these shared artifacts are not properly isolated or encapsulated, changes made to one feature may inadvertently affect other features [10], leading to feature interaction. Also, when these features (with shared artefacts) are combined in a new software product in SPL, these features will not operate as expected. For example, ArgoUML is a well-known open-source software for UML modeling. The original source code of ArgoUML is re-engineered to create SPL called ArgoUML-SPL[1] by extracting optional features from the source code of ArgoUML [22] [8]. During the extraction process, there are features with shared source code classes (for example, *State* feature and the *Activity* feature). They share the following classes and others: *ModelElementInfoList*, *FigStateVerte*. The detection of feature interactions becomes increasingly difficult as the number of extracted features in the core assets grows. The number of feature interactions is exponential in relation to the number of features [2].

[1]https://github.com/marcusvnac/argouml-spl

In the literature, there are proposed approaches to detect feature interactions during or after the implementation phase [4] [27] [3] [31]. However, these approaches rely on a model checker, which poses challenges in practical application and lacks scalability to actual SPLs [14]. Other approaches were proposed to detect unwanted feature interactions late (on testing) in the development process after the product has already been implemented [30]. Therefore, we propose in this article an approach to detect and visualize all feature interactions in extracted core assets from implementations of software variants early. Here, early means that the detection process is performed before the derivation of SPL's products from the core assets. Our approach depends on an unsupervised clustering technique called Formal Concept Analysis (FCA) to achieve the goal [7]. The main contribution of this work is to provide insight for experts about shared implementation among extracted features and exclude it during the derivation process of SPL from the core assets. Such implementation does not have a direct correspondence to any feature [13].

To assess the effectiveness of the proposed approach, we applied it to a large benchmark case study within this field, known as ArgoUML-SPL. The core assets of this SPL are built using the extractive approach via reusing already existing software variants. The results indicate that the proposed approach effectively detects and visualizes all interacted features in ArgoUML-SPL's core assets. Also, the proposed approach saves developer efforts for detecting interacted features in a range between 67% and 93%.

The remaining work in this article is organized into four main sections. Section II presents our motivational example and background. Section IV details the proposed approach. Section V presents the obtained results with a discussion. Related work is listed in Section III. Finally, the article is concluded in Section VI.

II. Motivation Example and Background

A. Motivation

In this subsection, we present the motivation of our proposal. To simulate product variants, we use three products of a simple software product line called *Drawing Product Line(DPL)* [11]. This SPL is only used for clarification purposes. Each product is a subset of a combination of the following features. *DPL* feature is to handle a drawing area, *Line* feature is to draw lines, *Rect* feature is to draw rectangles, *Color* feature is to select a color, *Fill* feature is to fill the shapes, and *Wipe* feature is to clean the drawing area.

Fig. 1 shows the representation of selected variants in terms of feature and source code views. The left Venn diagram displays the feature view of these product variants. For example, the pink circle represents *ProductVariant2 (PV2)* with three features: Fill, Line, and DPL. The right Venn diagram displays the source code view of these variants. For example, the pink circle represents PV2 with four sets of *source code artefacts group (AG)*: AG1, AG2, AG5, and AG6. The links between feature and source code views displayed in the figure are implementation links. For example, the source code artefacts group 4 (AG4) implements the *Color* feature.

It is worth noting that most features in this mapping between feature-source code views are directly linked to or associated with an AG. This allows us to speculate that these AGs represent the core implementations of their corresponding features. In our motivation example, the core implementation of the *Color* feature is AG4. However, AG2 does not have a direct implementation with any feature. This is because AG2 is not a feature-specific implementation but it is shared source code artefacts between *Line* and *Rect* features. This shared implementation between features causes unwanted feature interactions when these features are combined together to create a new software product in an SPL. Usually, this type of interaction is not easy to detect by analyzing the implementation of each feature separately. Especially when these features are not developed from scratch but they are reused and collected from product variants over time. Therefore, in this article, we propose to use FCA to automatically detect and visualize such feature interactions in SPL's core assets before building new products from such features.

B. Background

This section introduces software product line engineering (SPLE) and formal concept analysis (FCA).

1) Software Product Line Engineering: It is a systematic reuse mechanism to support the development of multiple similar software products from common core assets [24]. A core asset is a reusable software artefact that includes source code, features, architectural components, test cases, etc. These artefacts are linked together to support the automatic derivation of new SPL members from the core assets. The development life cycle of SPL consists of two phases: domain engineering and application engineering phases. Fig. 2 shows these phases.

Domain Engineering: It is the first phase in the SPL life cycle that aims to develop SPL's core assets and define commonality and variability in terms of the provided features by SPL members. These commonalities and variability are managed by the feature model. The assets include any development artefacts. Typically, the core assets are not built from scratch but they are reused from already existing software variants developed using ad-hoc reuse techniques, such as clone-and-own. One of the important assets that can be reused from these variants are features and their implementations which are always available. This good practice to build core assets allows to reduce time to market and maximize the return on investment. In the literature, this practice is called extractive approach [32].

Application Engineering: This is the second phase in SPL's lifecycle which aims to derive software products from the established core assets in the previous phase. These products are called SPL. The derivation process is performed automatically by exploiting traceability links among core assets. Also, the derivation process exploits commonality and variability in these assets to provide multiple products to meet the different needs of customers at the same time.

2) Formal Concept Analysis(FCA): Formal Concept Analysis (FCA) is a lattice-based method employed for data analysis and knowledge representation [12]. In our case, FCA is utilized as an unsupervised clustering algorithm, identifying

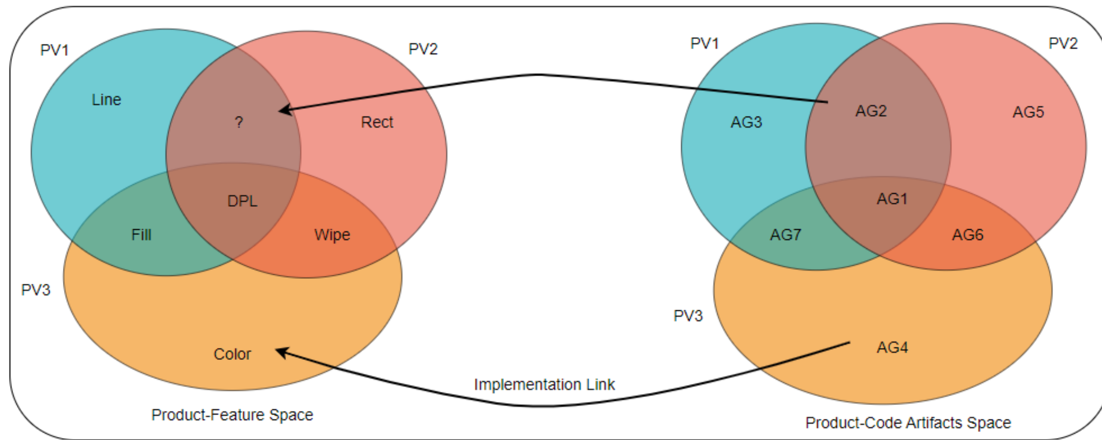

Fig. 1. Shared feature implementations problem.

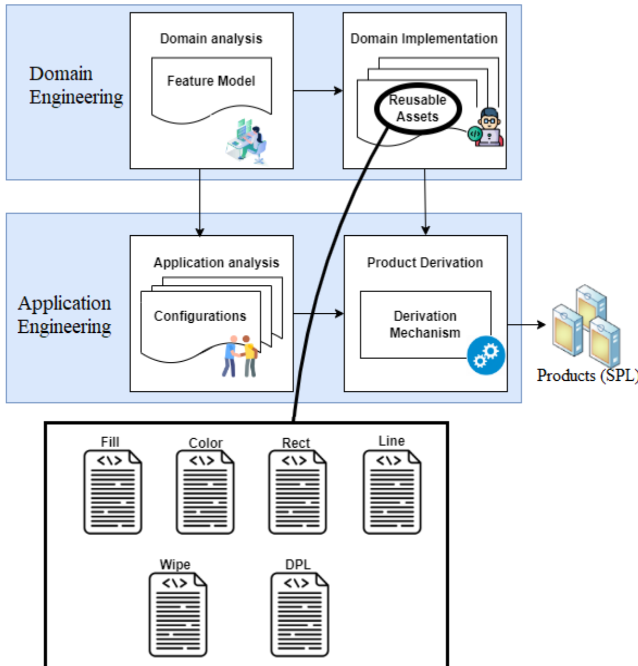

Fig. 2. Software product line phases [24].

significant clusters of objects with shared attributes. It accomplishes this by analyzing and structuring data according to the relationships between objects and attributes. It is currently applied to perform various tasks: valuable insights in software engineering, requirements analysis, software understanding, etc. To easily understand the FCA technique, it is illustrated with a familiar example. Consider a list of Mexican dishes as well as a list of ingredients for each dish, as shown in Table I. In this representation, dishes are listed in the rows, while the columns contain the respective ingredients for each dish.

Definition 1 (Formal Context): "A formal context is a 3-tuple $K = (O, A, R)$ where O and A are two sets, and $R \subseteq O \times A$ is a binary relation. Elements of O are called objects and elements of A are called attributes. A pair (o, a) of R means the object o owns the attribute a" [7].

The formal context corresponding to Mexican dishes and their ingredients is shown in Table II. As shown in this table, it shows the binary relationships between Mexican dishes and the ingredients they contain. Rows (objects) are dishes, columns (attributes) are ingredients, and cross marks (binary relations) determine which dishes own which ingredients.

For a given subset of objects $M \subseteq O$, then $M' = \{a \in A | \forall o \in M : (o, a) \in R\}$ is the set of common attributes. Also, for a given subset of attributes $B \subseteq A$, then $B' = \{o \in O | \forall a \in B : (o, a) \in R\}$ is the set of common objects. For example, assume that M = {*Enchiladas, Quesadillas, Tacos*} from Table II, the set of common attributes is M' ={*chicken, cheese, corn-tortilla*}. In the same way, if $B = (\{$*pork, rice*$\})$ then, B' = {*Burritos*}.

Definition 2 (Formal Concept): "Let $K = (O, A, R)$ be a formal context. A concept is a pair (E, I) such that $E \subseteq O$ and $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the concept extent and $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept. We denote by C_k the set of all concepts of K" [7].

For example, (*{Quesadillas, Tacos, Enchiladas}, {cheese, chicken, corn-tortilla}*) is a concept, while (*{ Nachos }, {cheese, vegetables, guacamole, beans}*) is not, because (*{ Nachos}*)' = *{cheese, vegetables, guacamole, beans}* while (*{cheese, vegetables, guacamole, beans}*)' = *{Nachos, Burritos}*.

Definition 3 (Concept Specialization Order): "Let K be a formal context, and let $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ be two formal concepts of C_K . C_1 is a specialization of C_2, denoted by $C_1 = (E_1, I_1) \leq_s C_2 = (E_2, I_2)$ if and only if $E_1 \subseteq E_2$ (and equivalently $I_2 \subseteq I_1$). C_1 is called a sub-concept of C_2. C_2 is called a super-concept of C_1" [7].

For example, ({Burritos}, {beans, rice, beef, cheese, guacamole, chicken, pork, vegetables, sour-cream, lettuce, flour-tortilla}) is a sub-concept of ({Burritos, Nachos}, {cheese, vegetables, beans, guacamole}). This is deduced by the definition of specialization order; one obvious property is that a sub-concept has (inherits top-down) the qualities of its super-concepts, but a super-concept has (inherits bottom-up) the objects of its sub-concepts.

Definition 4 (Concept Lattice): "Let C_K be the concept set of the formal context K. The concept lattice of K is the

TABLE I. MEXICAN DISHES AND THEIR INGREDIENTS

Mexican dish	Ingredients
Burritos	chicken, beans, rice, cheese, beef, pork, vegetables, guacamole, sour-cream, lettuce, and flour-tortilla
Enchiladas	chicken, cheese, sour-cream, and corn-tortilla
Fajitas	vegetables, cheese, guacamole, chicken, beef, sour-cream, lettuce, and flour-tortilla
Nachos	vegetables, beans, cheese, and guacamole
Quesadillas	chicken, corn-tortilla, beef, cheese, and flour-tortilla
Tacos	beans, cheese, lettuce, corn-tortilla, chicken, beef, and flour-tortilla

TABLE II. A FORMAL CONTEXT FOR MEXICAN DISHES

	chicken	beef	pork	vegetables	beans	rice	cheese	guacamole	sour-cream	lettuce	corn-tortilla	flour-tortilla
Burritos	X	X	X	X	X	X	X	X	X	X		X
Quesadillas	X	X					X				X	X
Enchiladas	X						X		X		X	
Nachos				X	X		X	X				
Fajitas	X	X		X			X	X	X	X		X
Tacos	X	X			X		X			X	X	X

concept set C_K provided with the partial order $\leq_K$, and is denoted by $(C_K, \leq_K)$" [7].

Fig. 3 displays the concept lattice corresponding to the formal context of Table II. This lattice is known as the Galois Sub-Hierarchy (GSH). It is a set of free empty concepts, each with at least one object or one attribute. Each concept in this lattice consists of three ordered counterparts: concept name, intent, and extent. Additionally, by examining the lattice, we can unveil numerous insights about these dishes, including their relationships with one another. For instance, concerning the presented Mexican dishes:

- Because cheese appears as the top concept's intent (*Concept_10*), which encompasses all the dishes in its extent, it can be deduced that all Mexican dishes contain cheese.
- When a concept has just Mexican dishes without ingredients, it signifies that these dishes lack specific ingredients and instead share common ingredients with dishes from other concepts. In *Concept_3*, for instance, *Nachos* inherits *guacamole*, *vegetables*, *beans* and *cheese* from other concepts.
- When a concept has ingredients and no dishes, it implies that these dishes are not exclusive to any particular Mexican dish; rather, they are shared by other dishes from different concepts. In *Concept_9*, for example, *beans* is shared between *Nachos*, *Tacos* and *Burritos* dishes.

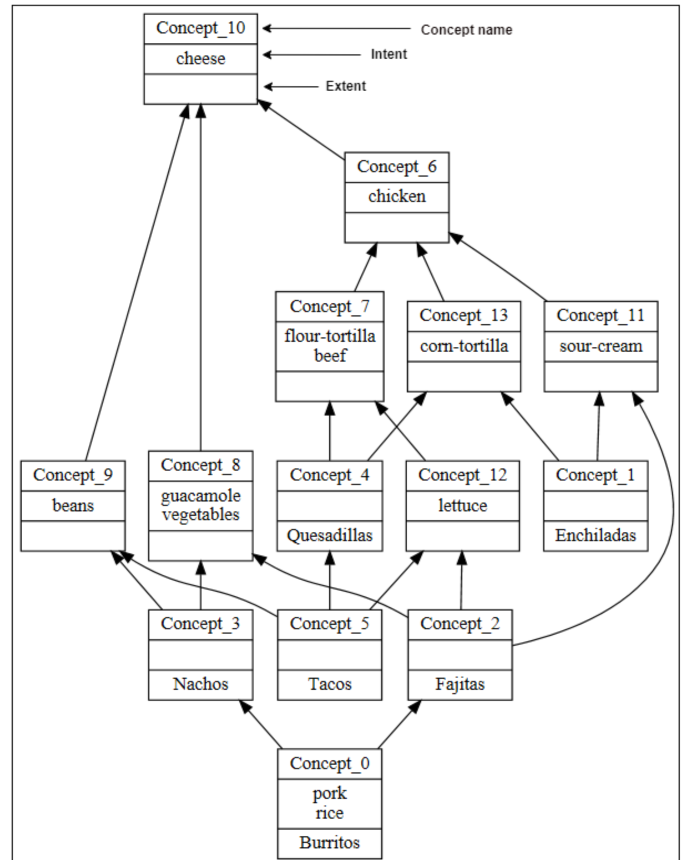

Fig. 3. GSH-Lattice for the formal context in Table II.

III. RELATED WORK

In the literature, unwanted feature interactions were studied in both SPL context and in single software products. Since we are interested in feature interactions in SPL, we present in this section only proposed approaches that detect unwanted feature interactions in SPL and exclude studies addressing such interactions in single software products.

Feature interaction approaches are classified into two categories based on the lifecycle phase when the feature interactions are detected [31]: *before actual coding* and *source code level*. The former detect feature interactions without the need to deal with the source code (focused on design, and requirements levels) [5]. However, the source code approaches detect such interactions using the source code. In this section, we present only studies in the second category since they are the closest to our topic in this article.

In [27], Scholz et al. proposed to use design by contract to detect feature interactions. The design-by-contract strategy includes preconditions, postconditions, and class invariants to specify the expected behavior of methods and classes. This strategy is performed using Java Modeling Language (JML) to specify the behavior of methods and classes, and then a model checker to identify unwanted interactions. In [3], Apel et al. proposed an approach based on feature-based specifications and verification to detect feature interactions. The feature implementations are annotated by feature specifications. Then, a model checker is used to automate the detection of feature interactions. Another similar approach was proposed by Apel et al. [4]. They provide a tool called *SPLVERIFIER* which is a model-checking tool for C-based and Java-based SPL. The above-mentioned approaches use a model checker, which is difficult to apply in practice and is not scalable to actual SPLs.

Also, these proposed approaches depend on prior knowledge about features, such as specifications, that are not always available for extracted features from legacy software variants.

Other studies [29] [20] [26] were proposed to parse the source code to identify feature interactions. Abstract syntax trees (ASTs) were built using different parsers: TypeChef, Java Compiler Tree API, and Fuji tool. These trees are used to compute dependencies among features. Schulze et al. [28] proposed a technique for analyzing feature co-changes based on association rule mining. This helps to identify features that commonly change together and to extract implicit feature dependencies. Korsman et al. [15] proposed a Python-based tool for automated reasoning of structural feature interactions in preprocessor directives of programs written in C and C++.

These studies do not support the detection of shared or common source code artefacts among extracted features from existing software variants. Also, they do not pay attention to classify interacted features into mandatory and optional interacted features for priority purposes. This is because these studies and other approaches presented in this section assume that features and their implementations are developed from scratch proactively or reactively.

IV. Proposed Approach

This section introduces our proposed approach for identifying and visualizing interacting features within the extracted core assets of the SPL. Initially, we provide a broad overview of the proposed approach. Subsequently, we delve into the specifics of each step in subsequent subsections.

A. An Overview

Fig. 4 gives an overview of the proposed approach. As shown in this figure, the approach takes a list of extracted features (from software variants) with their corresponding implementing artefacts. Then this input goes through four sequential steps to detect and visualize the interacted features. In the first step, these feature implementations are parsed to create a Feature-Artefact matrix. Then, a GSH-lattice corresponding to this matrix is built in the second step. In the third step, this lattice is reversed to detect the interacted features. Finally, the interacted features are categorized into mandatory or optional features based on the available feature model of those core assets.

B. Parsing Feature Implementations

In this step, the source code artefacts implementing each feature are parsed. These artefacts can be any level of granularity (fine and coarse granularity): package, class, methods, etc. This depends on the implementation artefacts for features. In this study, we assume that features are implemented by coarse granularity, such as classes and methods, since the proposed approach is evaluated using large case studies. Using the Eclipse Java Development Tool (JDT), the implementing source code of each feature is statistically analyzed to extract these classes and methods.

The output of this step is stored in a matrix called Feature-Artefact matrix. Rows represent source code artefacts, columns represent features, and cross signs refer to which artefact implements which feature. Table III is an example of such a matrix from our motivation example.

C. Building GSH Lattice of Features and Source Code Artifacts

After parsing the implementing artifacts for each feature, we rely on FCA in this step for detecting and visualizing shared implementing artifacts among features. We employ FCA because it enables the identification and visualization of source code artifacts shared among all features, subsets of features, and those exclusive to each feature. This capability arises from the hierarchical organization of lattice concepts.

To achieve the goal of this step, we use the Feature-Artefact matrix produced in the previous step (see Table III) as a formal context for FCA. Features and their implementing artifacts are attributes and objects in this formal context, respectively. The relation between a feature (*Line*) and a source code artifact (*A2*) refers to that this feature is implemented by this artifact. Using this formal context definition, we can generate a concept lattice comprising concepts composed of a set of source code artifacts shared by a set of features. Fig. 5 illustrates the resulting concept lattice, which represents a hierarchical arrangement of source code artifacts and features. In this lattice, each concept inherits its intents (features) from its ascendants (super-concepts) and its extents (source code artifacts) from its descendants (sub-concepts). Leveraging this lattice and FCA definitions (refer to subsection II-B2), we derive the following observations:

- The concept lattice includes isolated and linked concepts. The linked concepts together form a sub-hierarchy. The lattice may include more than one hierarchy. In Fig. 5, Concept_5 is an example of an isolated concept while the set consisting of {Concept_2, Concept_3, Concept_0} is an example of a sub-hierarchy.
- Each isolated concept in the lattice has non-empty intent and extent (For example Concept_5). The intent contains always a single feature and the extent represents the implementing source code artefacts for that feature. This implementation is the core implementation of the feature.
- The extent of each concept with empty intent in the sub-hierarchy (Concept_2) does not represent a core-feature implementation but it represents shared source code artefacts among features located in the intents of upper concepts (Concept_3 and Concept_0) in the same sub-hierarchy.

D. Detecting Interacting Features

This step aims to identify interacted features by traversing this produced lattice. The GSH lattice produced in the previous step visualizes interacted features by clustering shared source code artefacts among them. To end this, we propose Algorithm 1 to describe how to traverse the lattice.

In the beginning, the algorithm visits each concept in the lattice. Each concept with empty intent (cpt_e) (such as Concept_2 in Fig. 5) is the target and stored in a set called

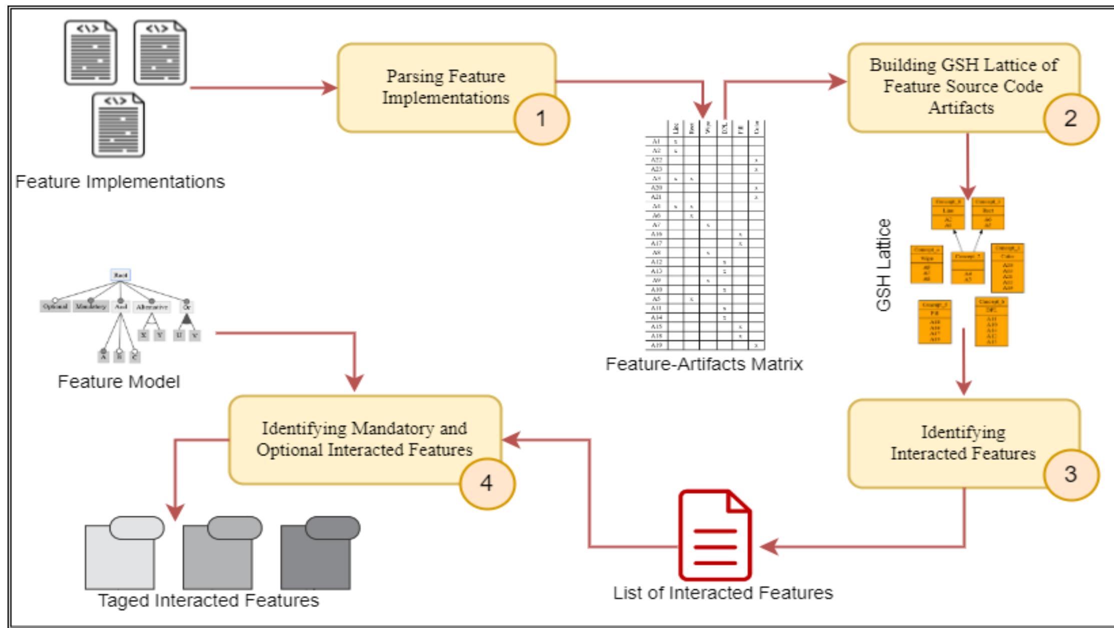

Fig. 4. An overview of the proposed approach.

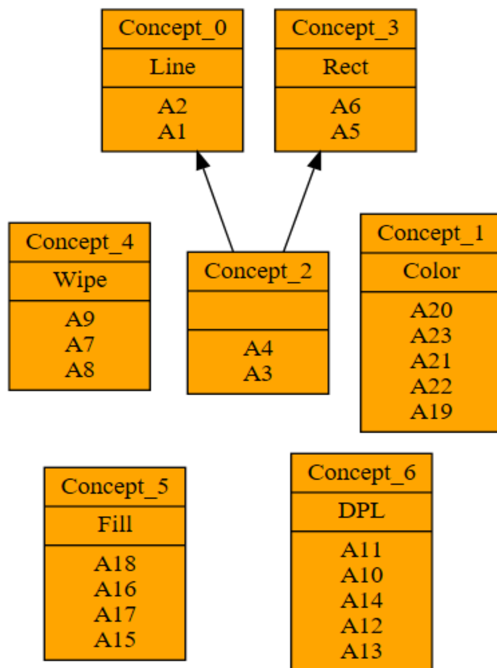

Fig. 5. GSH-Lattice for the formal context defined in Table III.

TABLE III. A FORMAL CONTEXT FOR FEATURE-ARTEFACT OF DRAWING PRODUCT LINE (DPL), A: REFERS TO A SOURCE CODE ARTEFACT

	Line	Rect	Wipe	DPL	Fill	Color
A1	x					
A2	x					
A22						x
A23						x
A3	x	x				
A20						x
A21						x
A4	x	x				
A6		x				
A7			x			
A16					x	
A17					x	
A8			x			
A12				x		
A13				x		
A9			x			
A10				x		
A5		x				
A11				x		
A14				x		
A15					x	
A18					x	
A19						x

CEI while other concepts are discarded since they are isolated concepts (lines 1–5). The extent of each concept in *CEI* represents shared source code artefacts among two or more features. These features are interacted features. To identify these features, we rely on the depth-first search algorithm (DFS) to get all upward reachable concepts from each concept in *CEI* and store them in a set called URC (upward reachable concepts) (lines 7-10). The intent of URC's concepts is interacted features. To extract these features from URC's concepts, we use a function called *getIntent()* (lines 11-13). Finally, for each cpt_e in *CEI*, we store cpt_e and its corresponding set of interacted features (SIFs) in a hashmap where cpt_e is the key and SIFs is the value (line 14).

Algorithm 1 Detecting Interacted Features

```
Input: FAL //Feature-artefacts Lattice
Output: IFs // HashMap of (concept, set<string>) called Interacted feature s
1  Set CEI ← Φ // Concepts with Empty Intent s
2  foreach (Concept cpt_e ∈ FAL) do
3      if (cpt_e.getIntent() is empty) then
4          CEI.add( cpt_e )
5      end
6  end
7  foreach (Concept con ∈ CEI) do
8      Set URC ← Φ //URC: Upward Reachable Concepts from Co2.
       Set URC ← DFS (con)
       SIFs ← Φ // SIFs: Set of Interacted Features
       foreach (Concept rc ∈ URC) do
9          SIFs.add (rc.getIntent())
10     end
11     IFs.put (con, SIFs)
12 end
13 return IFs
```

E. Detecting Mandatory and Optional Interacted features

After the identification of the interacted features, it is important to classify these features into mandatory and optional features. This classification is important for different reasons depending on the size and context of software products. For example, for prioritization, resource allocation, and estimation the effort should be spent to manage these interactions. In this aspect, we encounter four scenarios:

- If the interacted features are only mandatory features, each derived product from the core assets will behave in an unexpected way. Therefore, the interaction will negatively impact the entire generated SPL in the future.
- If the interacted features are mandatory and optional features, only derived products with at least one of these optional features will behave unexpectedly, as these products have duplicated source code artefacts.
- If the interacted features are two or more optional features, only derived products with at least two of these optional features will behave unexpectedly, as these products have duplicated source code artefacts.
- If the interacted features are mutual-exclusive optional features, the interacted features have no negative impact on the generated SPL.
  To perform the goal of this step, it takes the feature model as an input in addition to the list of interacted features identified in the previous step. This model is used to determine mandatory and optional features and other constraints, like mutual exclusive relations [1].

It is important to mention that also the GSH Lattice in Fig. 5 is utilized to visualize the implementation interactions within features. The shared source code artifacts are always located in the extent of concept(s) with empty intent (see Concept_2). These artifacts are not specific implementation of a feature but they are common between two or more features (Line and Rect features).

TABLE IV. Statistical Information for ArgoUML-SPL Features [8]

Feature	Package	Class	Method	LOC
State Diagram	0	48	15	3,917
Activity Diagram	2	31	6	2,282
Sequence Diagram	4	5	1	5,379
UseCase Diagram	3	1	1	2,712
Collaboration Diagram	2	8	5	1,579
Deployment Diagram	2	14	0	3,147
Cognitive Support	11	9	10	16,319

V. Experimental Results and Evaluation

In this section, we evaluate the proposed approach's effectiveness by applying it to a large benchmark case study on the subject.

A. Data Collection

In the literature, there is no ground truth dataset for implementation feature interactions. such datasets often depend on the context, domain, and nature of the software systems being considered. Therefore, to evaluate the proposed approach, we use the ArgoUML-SPL case study. The core assets of this SPL are built in an extractive way (reused from their existing variants). The implementation feature interactions within the features of this SPL are manually investigated.

ArgoUML, an open-source project written in JAVA, encompasses various UML diagrams and functionalities, including source code generation [8]. ArgoUML-SPL is derived from ArgoUML, wherein software products are generated from its source code base. This generation involves annotating the implementation of optional features with conditional compilation directives. The optional features include *Sequence Diagram, Collaboration Diagram, State Diagram, Activity Diagram, UseCase Diagram, Cognitive Support, and Deployment Diagrams*. Additionally, the source code base includes the implementation of a mandatory feature named Class Diagram, identified manually. Each feature in this case study is realized as a collection of packages and classes. Statistical details regarding the annotated features, such as the number of packages, classes, methods, and lines of code (LOC), are presented in Table IV. In this table, "Package," "Class," and "Method" represent the total number of annotated packages, classes, and methods constituting a feature implementation, respectively.

B. Evaluation Procedures and Research Questions

Two research questions are introduced in the course of this research work to evaluate the effectiveness of the proposed approach. These questions are as follows:

- RQ1: *How effectively the proposed approach can detect shared source code artefacts that cause unwanted feature interactions?*

- RQ2: *How much effort could the proposed approach save for the developer?*

To answer the first research question (RQ1), we validate the relevance of the detected shared source code artefacts across the implementations of ArgoUML-SPL's features. We apply the proposed approach to the feature implementations in the ArgoUML-SPL's core assets. Then, we investigate and analyze the shared artefacts to determine the relevancy of these artefacts to the resulting interacted features.

To answer the second research question (RQ2), we need an assessment method to measure the saved effort by the developer. To perform this, we propose a metric called the Development Effort Saving ratio (DES). This metric should be applied to products derived from common core assets of a given SPL. The idea behind the DES metric is to calculate the efforts that should be spent (but saved) by the developer to detect interacted features after generating products from the core assets. This effort will be saved when we detect interacted features early (in the core assets) and before generating products. Higher DES's values are higher detection efforts saved by the developer(s) and vice versa. The range of values in DES is 0 to 1.

DES mainly measures the occurrence of products with features that share artifacts. Therefore, we follow the following evaluation protocol to apply this measure:

1- We randomly generated three sets with different sizes (small, medium, large) of products from ArgoUML-SPL's core assets.

2- Determine the interacted features in each generated set.

3- Determine mandatory and optional interacted features.

4- We apply Eq. 1 and 2 for mandatory and optional features, respectively.

$$DES_m = \frac{(\sum all\ possible\ products - 1)}{(all\ possible\ products)} \quad (1)$$

$$DES_o = \frac{(\sum all\ impacted\ products - 1)}{(all\ generatd\ products)} \quad (2)$$

In these equations, *all possible products* refers to all valid software products that can be generated from the core assets, *all impacted products* refers to randomly generated products with unwanted feature interactions, and *all generated products* refers to all randomly generated products for evaluation.

C. Results

In this section, we answer the introduced research questions to validate the proposed approach.

1) The Relevancy of Shared artefacts to the resulting interacted features (RQ1): Table V lists shared source code classes among all feature implementations in the ArgoUML-SPL's core assets. Also, it shows interacted features and their type (mandatory or optional feature). As shown in Table V, all interacted features are optional and in pairs, as feature interactions exist mostly between two features [18]. For example, *Cognitive and Sequence* features share a class called *CrSeqInstanceWithoutClassifier*. Also, State and Activity features share 18 source code classes.

To validate whether the detected shared classes are relevant to the implementation of features contributing to the interaction or not, we manually investigate and analyze these shared classes and their inline comments. For example, we analyzed the shared classes between *State and Activity features*. We found that all these classes are related and implement *States and Events*. Also, by returning to the documentation of these features, we found that *State and Activity* features are similar [8]. *State* feature is used to graphically represent objects of a single class and track the different states of its objects through the system. *Activity* feature is used to graphically describe the system behavior as a set of activities, and these activities are the state of doing something. Also, we studied the shared classes between *Cognitive* and both *Sequence and Deployment* features. We discovered that *Cognitive* feature is a crosscutting feature in ArgoUML. This means that the Cognitive's implementation is spread over the implementation of other features, such as Sequence and Deployment.

In summary, the suggested approach can effectively detect interacting features in the core assets of ArgoUML-SPL by determining shared source code artifacts among these features, which answers the first research question. This is based on the obtained results in Table V.

Due to the size of GSH lattice corresponding to ArgoUML-SPL, we can not put it in the article but it is utilized to visualize implementation interactions within ArgoUML-SPL's features as explained in the illustrative example (see Section IV).

2) Saving Developers Efforts (RQ2): Table VI shows all unwanted feature interactions in three randomly generated sets (A, B, and C) of products from the core assets of ArgoUML-SPL. Also, the table shows the savings percentage of developer efforts (DES) to detect these interacted features in these generated products. As shown in this table, the range of DES's values for set A is [75% to 93%], set B is [80% to 90%], and set C is [67% to 92%]. The reason behind the high saving efforts in set A compared to other sets is that the products of set A contain interacted features more than those of other sets. This leads to spending more effort by developers to detect these features manually.

Table VII shows statistics about the saving efforts obtained by the proposed approach. As an overall evaluation, the amount of effort saved by the proposed approach in all generated sets is 67% to 93% which is a high range.

To summarize, the answer to the second research question indicates that the proposed approach effectively minimizes developers' detection efforts concerning interacting features. This is based on the findings shown in Tables VI and VII.

D. Threats to Validity

In this section, we list potential threats that could compromise the validity of our proposed approach. We found the following four main threats:

- We only used one case study to evaluate the effectiveness of the proposed approach. However, ArgoUML-

TABLE V. INTERACTED FEATURES IN THE ARGOUML-SPL CORE ASSETS

Shared Classes	Interacted Features	Feature Types
CrInterfaceWithoutComponent, CrObjectWithoutComponent, CrNodeInsideElement, CrInstanceWithoutClassifier, CrInstanceWithoutClassifier, CrClassWithoutComponent, CrObjectWithoutClassifier, CrWrongLinkEnds, CrNodeInstanceWithoutClassifier, CrWrongDepEnds, CrNodeInstanceInsideElement, CrComponentWithoutNode, CrNodeInstanceInsideElement, CrComponentInstanceWithoutClassifier	Cognitive-Deployment	optional
CrSeqInstanceWithoutClassifier	Sequence-Cognitive	optional
ModelElementInfoList, FigStateVertex, ButtonActionNewSignalEvent, ButtonActionNewCallEvent, FigFinalState, StateDiagramGraphModel, ButtonActionNewEvent, UMLSubmachineStateComboBoxModel, PropPanelStubState, FigTransition, StateDiagramRenderer, PropPanelSynchState, ButtonActionNewTimeEvent, UMLStubStateComboBoxModel, , ButtonActionNewChangeEvent, UMLSynchStateBoundDocument, StateBodyNotationUml, InfoItem,	Activity-State	optional
ActionAddClassifierRole, FigClassifierRole, SelectionClassifierRole	Collaboration-Sequence	optional

TABLE VI. DES RESULTS OF RANDOMLY GENERATED PRODUCTS OF ARGOUML-SPL

Interacted Features	Feature Types	Impacted Products	DES Value
DES's Results of Random 15 Product of ArgoUML-SPL (Set A)			
Cognitive-Deployment	optional	P4,P3,P2,P5	93%
		P9,P8,P7,P6	
		P11,P10,P12	
		P15,P14,P13	
Sequence-Cognitive	optional	P6,P5,P4,P3	92%
		P9,P10,P7,P8	
		P11,P13,P12	
		P15,P14	
Activity-State	optional	P6,P10,P9,P7	75%
Collaboration-Sequence	optional	P7,P8,P10	83%
		P15,P14,P11	
DES's Results of Random 37 Product of ArgoUML-SPL (Set B)			
Sequence-Cognitive	optional	P36,P34,P35,P33	80%
		P37	
Activity-State	optional	P19,P5,P7,P8,P4	90%
		P36,P22,P23,P20	
		P37	
Collaboration-Sequence	optional	P8,P12,P5,P9,P6	80%
		P15,P13,P37,P16	
DES's Results of Random 50 Product of ArgoUML-SPL (Set C)			
Sequence-Cognitive	optional	P50,P2, P1	67%
Activity-State	optional	P6,P5,P8,P9	92%
		P24,P22,P21	
		P36,P25,P37	
		P40,P39	
Collaboration-Sequence	optional	P23,P22,P2,P1	90%
		P25,P30,P29,P26	
		P33,P32	

TABLE VII. DES'S STATISTICS OF RANDOMLY GENERATED PRODUCTS OF ARGOUML-SPL

ArgoUML-SPL Set	Min	Average	Max	Standard Deviation
Set A (15 Product)	0.75	0.85	0.93	0.07
Set B (37 Product)	0.80	0.83	0.90	0.04
Set C (50 Product)	0.67	0.83	0.90	0.11

SPL is a large benchmark case study in this subject [8]. In addition, the proposed approach can be applied to others without extra work.

- The studied case study contains only interacted optional features and lacks interacted mandatory features. Based on DES equations and the obtained results, the DES results for interacted mandatory features will not differ much.
- The proposed technique assumes that the feature is implemented as a set of source code classes. However, features in smaller SPLs can be implemented as a collection of methods or other more detailed source code artifacts. However, the proposed approach can be adapted to consider any level of source code granularity to implement features.
- The amount of effort saved is assessed using a bespoke metric (DES). However, this metric reflects the reality where we manually detect the shared artifact among features to discover the detecting effort that could be spent if the proposed approach was not used.

VI. CONCLUSION AND PERSPECTIVES

In this article, we have proposed an approach to detect and visualize feature interactions in extracted core assets of SPLs early. The approach is based on an unsupervised clustering technique called Formal Concept Analysis. The application of the proposed approach on a benchmark case study in the subject shows that it is effective in detecting implementation feature interaction. Also, it reduces detecting efforts spent by developers in a range between 67% and 93%.

As perspectives, we plan to apply the proposed approach to other case studies with different granularities (fine and coarse grain) of feature implementations. Also, we plan to detect unwanted feature interactions based on other structural dependencies in source code (not only shared source code artefacts) among feature implementations. Moreover, we will try to detect domain or implementation feature interactions in which features are not already modularized.

REFERENCES

[1] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. On extracting feature models from product descriptions. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, page 45–54, New York, NY, USA, 2012. Association for Computing Machinery.

[2] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, page 1–8, New York, NY, USA, 2013. Association for Computing Machinery.

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

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491, 2013.

[5] L. N. Baldo, A. M. M. M. Amaral, E. OliveiraJr, and T. E. Colanzi. *Preventing Feature Interaction with Optimization Algorithms*, pages 265–283. Springer International Publishing, Cham, 2023.

[6] D. Beuche. Transforming legacy systems into software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 275, New York, NY, USA, 2013. Association for Computing Machinery.

[7] J. Carbonnel, K. Bertet, M. Huchard, and C. Nebut. Fca for software product line representation: Mixing configuration and feature relationships in a unique canonical representation. *Discrete Applied Mathematics*, 273:43–64, 2020. Advances in Formal Concept Analysis: Traces of CLA 2016.

[8] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 191–200, Washington, DC, USA, 2011.

[9] H. Eyal Salman. Leveraging a combination of machine learning and formal concept analysis to locate the implementation of features in software variants. *Information and Software Technology*, 164:107320, 2023.

[10] H. Eyal Salman, A.-D. Seriai, and C. Dony. Feature-level change impact analysis using formal concept analysis. *Int. J. Softw. Eng. Knowl. Eng.*, 25:69–92, 2015.

[11] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400, 2014.

[12] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations*. Springer-Verlag, 1999.

[13] N. Hlad, B. Lemoine, M. Huchard, and A.-D. Seriai. Leveraging relational concept analysis for automated feature location in software product lines. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2021, page 170–183, New York, NY, USA, 2021. Association for Computing Machinery.

[14] S. Khoshmanesh and R. R. Lutz. The role of similarity in detecting feature interaction in software product lines. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 286–292, 2018.

[15] D. Korsman, C. D. N. Damasceno, and D. Strüber. A tool for analysing higher-order feature interactions in preprocessor annotations in c and c++ projects. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '22, page 70–73, New York, NY, USA, 2022. Association for Computing Machinery.

[16] C. Krueger. Easing the transition to software mass customization. In F. van der Linden, editor, *Software Product-Family Engineering*, pages 282–293, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[17] C. Kästner, S. Apel, S. ur, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. 08 2009.

[18] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 105–114, New York, NY, USA, 2010. Association for Computing Machinery.

[19] L. Linsbauer, E. R. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 131–140, New York, NY, USA, 2013. Association for Computing Machinery.

[20] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Variability extraction and modeling for product variants. SPLC '18, page 250, New York, NY, USA, 2018. Association for Computing Machinery.

[21] J. Martinez, W. K. G. Assunção, and T. Ziadi. Espla: A catalog of extractive spl adoption case studies. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC '17, page 38–41, New York, NY, USA, 2017. Association for Computing Machinery.

[22] J. Martinez, N. Ordoñez, X. Tërnava, T. Ziadi, J. Aponte, E. Figueiredo, and M. T. Valente. Feature location benchmark with argouml spl. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, page 257–263, New York, NY, USA, 2018. Association for Computing Machinery.

[23] H. Mili, I. Benzarti, A. Elkharraz, G. Elboussaidi, Y.-G. Guéhéneuc, and P. Valtchev. Discovering reusable functional features in legacy object-oriented systems. *IEEE Transactions on Software Engineering*, 49(7):3827–3856, 2023.

[24] K. Pohl, G. Bckle, and F. J. van der Linden. Software product line engineering: Foundations, principles and techniques. Springer Publishing Company, Incorporated, 2010.

[25] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005.

[26] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi. Assessing fine-grained feature dependencies. *Information and Software Technology*, 78:27–52, 2016.

[27] W. Scholz, T. Thüm, S. Apel, and C. Lengauer. Automatic detection of feature interactions using the java modeling language: An experience report. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[28] S. Schulze, P. Engelke, and J. Kruger. Evolutionary feature dependencies: Analyzing feature co-changes in c systems. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 84–95, Los Alamitos, CA, USA, oct 2023. IEEE Computer Society.

[29] S. Schuster, S. Schulze, and I. Schaefer. Structural feature interaction patterns: Case studies and guidelines. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, New York, NY, USA, 2014. Association for Computing Machinery.

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

[31] L. R. Soares, P.-Y. Schobbens, I. do Carmo Machado, and E. S. de Almeida. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, 98:44–58, 2018.

[32] Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, page 109–118, USA, 2010. IEEE Computer Society.

[33] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *2012 19th Working Conference on Reverse Engineering*, pages 145–154, 2012.

[34] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. Feature identification from the source code of product variants. In F. R. MENS T., CLEVE A., editor, *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 417–422, Los Alamitos, CA, USA, 2012.