Leveraging relational concept analysis for automated feature location in software product lines

Nicolas Hlad, Bérénice Lemoine, Marianne Huchard, Abdelhak-Djamel Seriai

LEVERAGING RELATIONAL CONCEPT ANALYSIS FOR AUTOMATED FEATURE LOCATION IN SOFTWARE PRODUCT LINES *

Hlad Nicolas
LIRMM, CNRS
Univ. of Montpellier
Montpellier, France
hlad@lirmm.fr

Lemoine Bérénice
LIRMM, CNRS
Univ. of Montpellier
Montpellier, France
Lemoine@lirmm.fr

Huchard Marianne
LIRMM, CNRS
Univ. of Montpellier
Montpellier, France
huchard@lirmm.fr

Seriai Abdelhak-Djamel
LIRMM, CNRS
Univ. of Montpellier
Montpellier, France
seriai@lirmm.fr

ABSTRACT

Formal Concept Analysis (FCA) has been introduced for almost a decade as a suitable method for Feature Location (FL) on a collection of product variants. Even though FCA-based FL techniques allow to locate the core of a feature implementation, they do not propose a solution to trace feature interactions to their implementation. Thus, the extracted traceability links (traces) are too inaccurate, and, in the context of SPL extraction, cannot be used to generate complete products.

In this paper, we propose to complement the FCA-based FL techniques by leveraging the power of Relational Concept Analysis, an extension of FCA to multi-relational data. From two given formal contexts, one for the product's artefact and one for their features, our technique computes the traces that link the features and the feature interactions to their corresponding artefacts. Additionally, we introduce a stage that removes unnecessary features from the extracted traces, to make them easier to understand by an expert. Our FL technique can be applied at any artefact granularity (from files to statements) and independently from software languages.

The results show that our technique produces valid traces, from which we were able to completely rebuild the set of artefacts for each initial product. Moreover, they show that our trace reduction removes, on average, between 31% and 85% of unnecessary features from the traces.

***K*eywords** feature location · software product variants · software product lines · formal concept analysis · relational concept analysis

1 Introduction

In software engineering, a feature is often described as a specific software functionality, often referred to by a name and accompanied by a description [26]. In this context, a feature can be implemented in a software by a set of various elements (e.g., code, text files, images, etc.). We refer to this set of elements as the *artefacts*. For two different features, we hypothesize that this set should be distinct. The use of features allows describing a software according to the features it implements. Moreover, this is particularly useful when designing Software Product Lines (SPL) [25].

A particularity of features is that they can interact in a software. For instance, a particular software behavior can emerge from the joint use of two features, which is different from the disjoint union of the behaviors obtained when both features are used separately. An interaction can be perceived as negative (if the behavior is not the one expected) or positive (if the behavior is expected) [23, 24]. Thus, the feature interactions are often deliberately handled in the software implementation, if only to avoid a negative interaction.

* *Citation*: **Hlad et al. Leveraging Relational Concept Analysis for Automated Feature Location in Software Product Lines. DOI:** https://doi.org/10.1145/3486609.3487208**.**

Feature Location (FL) is a process that consists in finding in a software the elements that participate in the implementation of a software feature [21, 23]. Hence, FL is often employed to improve the maintainability and evolution of a software [22], and is, for instance, used when migrating software variants toward an SPL [41, 38, 39]. There are different techniques for FL: static analysis, dynamic analysis, or lexical analysis [21]. The goal is to create *traceability links* [21, 20] that associate the features to their implementation. We refer to traceability links as *traces* to evoke the links between a feature, or a feature interaction, and the artefacts of its implementation. However, we can distinguish between FL techniques applied on a collection of software variants and those applied on single software [20]. Indeed, in addition to the three previous techniques, the FL applied on multiple product variants can rely on the variability of the products.

Among others, some works have studied the application of Formal Concept Analysis (FCA) to analyze the static variability automatically and locate the features [18, 19, 38]. FCA is a knowledge analysis method based on lattice theory, which is used in these works as a clustering method: The clusters are composed of maximal object sets that share maximal attribute sets. The results obtained in these works were encouraging but limited: (1) the proposed techniques do not consider feature interactions and thus are unable to recover their traces; (2) the traces are inaccurate, often linking the wrong artefacts to the wrong feature (mainly because feature interaction were not considered, as we will illustrate in the next section); (3) these techniques are applied at a level of granularity which is too coarse for the artefacts, making them unable to capture a feature's implementation located inside a method statements. For these reasons, the FCA-Based FL techniques are ineffective because they miss a part of the variability information when migrating to an SPL, and thus have not been widely adopted.

In this paper, we propose to extend and complement the FCA-based automated FL techniques. Our technique focuses on employing FCA to build and work with a unique structure that contains numerous information between the software variants' data, such as: the implication, mutual exclusion, co-occurrence constraints, and the affiliation of artefacts and features to their products.

In more detail, our first contribution is the definition of a complete process allowing the recovery of feature implementations and feature interactions. In our technique, we rely on FCA and its Relational Concept Analysis extension (RCA) [16] to retrieve the traces for features and feature interactions. Furthermore, as discussed by Kastner et al. [17], the granularity of the artefacts has an impact over the construction of an extracted SPL. For the traces to be accurate, it is necessary to provide a FL technique that is adaptable to a fine granularity of artefacts, which is what we propose here. We obtain our fine-grained artefacts using a technique proposed by [30]. This technique allows us to produce a formal context for the artefacts (required for FCA) where each statement is identified as an individual artefact.

Our second contribution regards the reduction of the length of our extracted traces. We focus on diminishing the number of features involved in a trace, especially for the interactions, and thus producing a more comprehensible trace. These traces could then be easier to understand by experts, and we plan their application in code maintainability scenarios, such as generating annotations in the code at the location of the feature implementations.

This paper is structured as follows: Section 2 presents our motivation over a running example; Section 3 introduces FCA and RCA which are used in our approach; Section 4 presents our FL techniques; Section 5 presents our results over a set of different SPLs; Section 6 compares our approach with the related work; Finally Section 7 concludes with a summary of this work and our perspectives.

2 Motivation

This section presents a motivating example to outline our proposal, and introduces the research questions.

Running Example. PrinterSPL is a software product family used for demonstration purposes only. It simulates the behavior of printing features. This product family is built on top of six features: *Base*, the feature that gathers the common and mandatory functionalities of the product variants; *Printer*, which represents the printing feature and is refined in two sub-features *Recto_Verso* and *Picture*, both respectively allowing double-sided and image printing; *Scanner* corresponding to the paper scanner feature; finally, *Color* allowing color printing and scanning. There are fourteen product variants in this software family (see Table 1).

FCA for Feature Location. Feature Location consists of the association of each artefact to at least one feature and vice versa. In general, FL techniques assume that each product's features are known, which is also the case in our paper.

For FL applied on a set of product variants, the FL based on Formal Concept Analysis (FCA) speculates that associations between features and artefacts are based on their co-occurrence inside the same products. These approaches

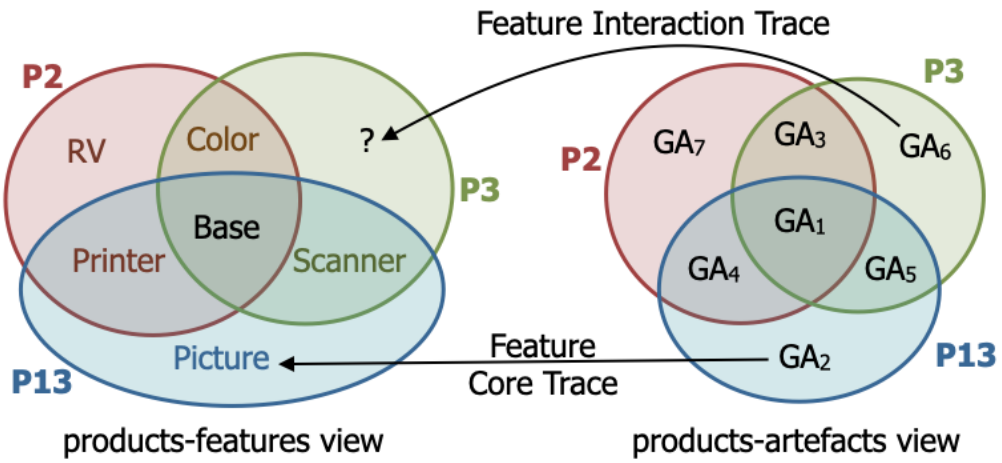

Figure 1: Feature Location problem illustration

use the clustering capability of FCA [2] to divide and reduce their search spaces, to associate the products-features space to the products-artefacts space. Venn diagrams can represent this division. Let us consider a side example where only $P2$, $P3$ and $P13$ exist in PrinterSPL. Figure 1 shows an example of the division applied to these products. The left-hand side shows the products-features Venn diagram, e.g. the red disc on the left is the set of features of product $P2$; the right-hand side shows the products-artefacts groups Venn diagram, e.g. the red disc on the right is the set of artefact groups of $P2$. We notice that some features directly associate with artefacts, so a common hypothesis is to consider these artefacts as the core implementation of a feature. In our case, the core implementation of $Picture$ is the group of artefacts GA_2.

However, the artefact group GA_6, does not have a direct correspondence to any feature. FCA was used for FL on products variants by two works [38, 39]. In these works, they apply FCA with the previous hypothesis to associate the artefacts to the features. However, in cases like with GA_6, they propose to set these artefacts to one of $P3$'s features, either $Color$ or $Scanner$. To do so, they use lexical analysis to find a lexical proximity between the GA_6 artefacts and the description of one of the two features.

However, when setting GA_6 as e.g. $Scanner$ implementation, a problem emerges if we use the traces to recover the implementation of the initial products. In this example, $P13$ also implements $Scanner$, but without the GA_6 group. By considering the implementation of $Scanner$ as GA_6+GA_5, we unintentionally add artefacts to the implementation of $P13$, which changes the product implementation, and possibly compromises $P13$'s behavior.

Other works not based on FCA, like the approach of Fisher and al. [42], speculate that these artefacts do not implement the body of a feature but an interaction between multiple features. Our work proposes a FL technique for product variants based on FCA to find and correctly associate the missing interactions. The main objective of our work is to associate each artefact group to either a feature or a feature interaction. We hypothesize that the implementation of a feature interaction does not share artefacts with the core implementation of the features mentioned in the interaction.

Research Questions. Therefore our first research question is:

- *RQ1: "How to create a fully automated feature location technique using FCA/RCA and dealing with feature interactions?"*

However, as we will see in this paper, our interaction traces extraction method can produce traces with numerous redundant features involved. To preserve the interaction traces' readability and capture more precisely the features involved in the interaction, we propose a traces reduction stage. It leads us to our second research question:

- *RQ2: "How to reduce the size of interaction traces extracted from FCA?"*

3 Background

This section presents the techniques used for the FL, namely FCA and RCA.

3.1 Formal Concept Analysis

Formal Concept Analysis (FCA) [15] is a general unsupervised classification and clustering method in its primitive version. It allows, starting from a data description called a *formal context*, to form *concepts* and a hierarchy between these concepts called the *concept lattice*. A Formal Context (FC) is a binary relation representing the ownership between *objects* and *attributes*. Table 1 and Table 2 show two formal contexts. The first one (FC_{pf}) shows software products associated with the features they provide. The second one (FC_{pa}) associates software products to the artefacts that implement them.

Table 1: Product-Features Formal Context (FC_{pf})

FC_{pf}	Base	Printer	Recto_Verso	Color	Scanner	Picture
P_1	x	x	x			
P_2	x	x	x	x		
P_3	x			x	x	
P_4	x				x	
P_5	x	x	x		x	
P_6	x	x	x	x	x	
P_7	x	x		x	x	x
P_8	x	x				x
P_9	x	x	x			x
P_{10}	x	x		x		x
P_{11}	x	x	x	x		x
P_{12}	x	x	x	x	x	x
P_{13}	x	x			x	x
P_{14}	x	x	x		x	x

Table 2: The abridged Product-Artefacts Formal Context (FC_{pa})

FC_{pa}	A1	A2	....	....	A28	A29	A30	A31
P_1	x	x	....	....				
P_2	x	x	....	....				
P_3	x	x	....	....				
P_4	x	x	....	....				
P_5	x	x	....	....				
P_6	x	x	....	....				
P_7	x	x	....	....	x	x	x	x
P_8	x	x	....	....				x
P_9	x	x	....	....				x
P_{10}	x	x	....	....	x	x	x	x
P_{11}	x	x	....	....	x	x	x	x
P_{12}	x	x	....	....	x	x	x	x
P_{13}	x	x	....	....				x
P_{14}	x	x	....	....				x

A concept (or *cpt* for short) represents the association of a maximal group of objects with the maximal group of attributes they share. Each concept is split into two parts: an *intent* and an *extent*. The intent represents the attribute set. Respectively, the extent represents the object set.

A hierarchy between concepts is formed from a partial order based on the intent and extent inclusion: a concept cpt_{Sub} specializes another concept cpt_{Super} when cpt_{Sub}'s intent (resp. cpt_{Super}'s extent) contains cpt_{Super}'s intent (resp. cpt_{Sub}'s extent). This partial order allows a form of inheritance between the concepts. Then a diagram representing a concept hierarchy only shows, for a concept, its *reduced intent*, deprived of its (top-bottom) inherited attributes and its *reduced extent*, deprived of its (bottom-top) inherited objects. For example, the concept cpt_{56} named *cpt_Feature_56* in Figure 4, has for reduced intent $\{Scanner\}$ and for reduced extent $\{P4\}$. Even though cpt_{56} *full intent* contains cpt_{56}'s attributes and all its parents' attributes; while its *full extent* contains its objects, and all its children's objects. Thus cpt_{56} intent is $\{Base, Scanner\}$ and its extent is $\{P3, P4, P5, P6, P7, P12, P13, P14\}$.

Several hierarchies can be built with FCA, the concept lattice being the hierarchy containing all concepts. This work will use the hierarchy restricted to the concepts that introduce at least one object or one attribute, called the AOC-Poset

(Attribute-Object Concept partially ordered set). Figure 3 and 4 respectively represent the AOC-Posets of FC_{pa} (cf. Table 2) and FC_{pf} (cf. Table 1).

Extracting Relations Several constraints or relations between the attributes of concepts can be extracted from an AOC-Poset. In particular, the *implication*, *co-occurrence*, and *mutual exclusion* relations. The implications are deduced from the inheritance links between introducer concepts: the attributes introduced by a parent concept are implied by the attributes introduced by its children. On the other hand, mutual exclusion is present between attributes of two concepts whose extents do not intersect. A mutual exclusion means that the attributes of two concepts can never be found together. Carbonnel et al. [28, 36] propose a technique and a tool to extract these different relations.

3.2 Relational Concept Analysis

Relational Concept Analysis (RCA) [16] is a FCA extension that creates multiple connected concept hierarchies to learn and cluster from directed relations between multiple formal contexts. As input, RCA takes a *Relational Context Family* (RCF). An RCF contains two sets, a set of formal contexts and a set of relational contexts. A *Relational Context* (RC) represents a unidirectional relation between two types of objects. More precisely, it is a binary relation, oriented from the objects of a *source* formal context to the objects of a *target* formal context.

To compute the concepts, RCA creates quantified relational attributes $qr(C)$, where q is a quantifier (e.g. existential $\exists$ or universal $\forall$), r is a relational context, and C is a concept from the lattice of the target formal context of r. Therefore, a relational attribute describes the objects of one formal context by their links to the concepts of another formal context. In this work, we use the $\exists$ quantifier, which establishes a link between a product and either an artefact concept or a feature concept when the product owns at least one element of the concept extent. As output, RCA produces multiple concept hierarchies, e.g. multiple AOC-Posets, one for each formal context.

Note that for an AOC-Poset built from a formal context that is the source of one or several relational contexts, we find two types of attributes in this AOC-Poset's concept intents: *native attributes* (i.e. original attributes from the formal context); *relational attributes* (i.e. attributes pointing through a relation to a concept of the target formal context of one of the relational context).

In the diagrams, when the target concepts are *introducers*, they are replaced by their own native attributes to simplify the reading and be used in our approach.

3.3 Our Specific Use of RCA

For our study, the Relational Context Family is composed of a formal context FC_p empty of native attributes whose objects are the products (configurations), FC_{pa} describing the products by their artefacts (it is the 'artefact view' on products), FC_{pf} describing the products by their features (it is the 'feature view' on products), and two relational contexts connecting the products to their respective views: $FC_p \rightarrow FC_{pa}$ and $FC_p \rightarrow FC_{pf}$. We obtain three AOC-Posets: the two AOC-Posets of Figures 3 and 4, and a relational AOC-Poset (Figure 5) whose concepts expose groups of products (extents) with their shared groups of relational attributes towards artefacts or feature concepts (or native attributes when these concepts introduce native attributes).

For example, the concept named *Concept_FCp_75* contains two relational attributes (quantifier $\exists$ is implicit): *FCpa(A27)* introducing the artefact *A27* and *FCpf(Scanner)* introducing the *Scanner* feature. In this paper, we use this type of relation to extract our traces. Therefore, the output of FCA/RCA is a set of AOC-Posets.

4 Feature Location Technique using FCA/RCA

This section presents our Feature Location Technique, depicted in Figure 2. Our technique is a four stages process. (1) An *Initialisation* stage, which builds the AOC-Posets. (2) A *Traces Extraction* stage, which links a feature or a set of interacting features to their implementation (i.e. a set of artefacts). This stage is decomposed into two sub-stages: (A) a *Core Traces Extraction*, extracting the traces that link a feature to its proper implementation; (B) an *Interaction Traces Extraction*, extracting the traces that link multiple features to the implementation of their interaction (i.e. a set of artefacts). (3) A *Trace Transformation* stage, which turns a trace into a Boolean expression (i.e. propositional logic formula). And finally, (4) a *Traces Reduction* phase, which uses the Boolean algebra on the logical formula to reduce the number of propositions, and thus to obtain a smaller trace.

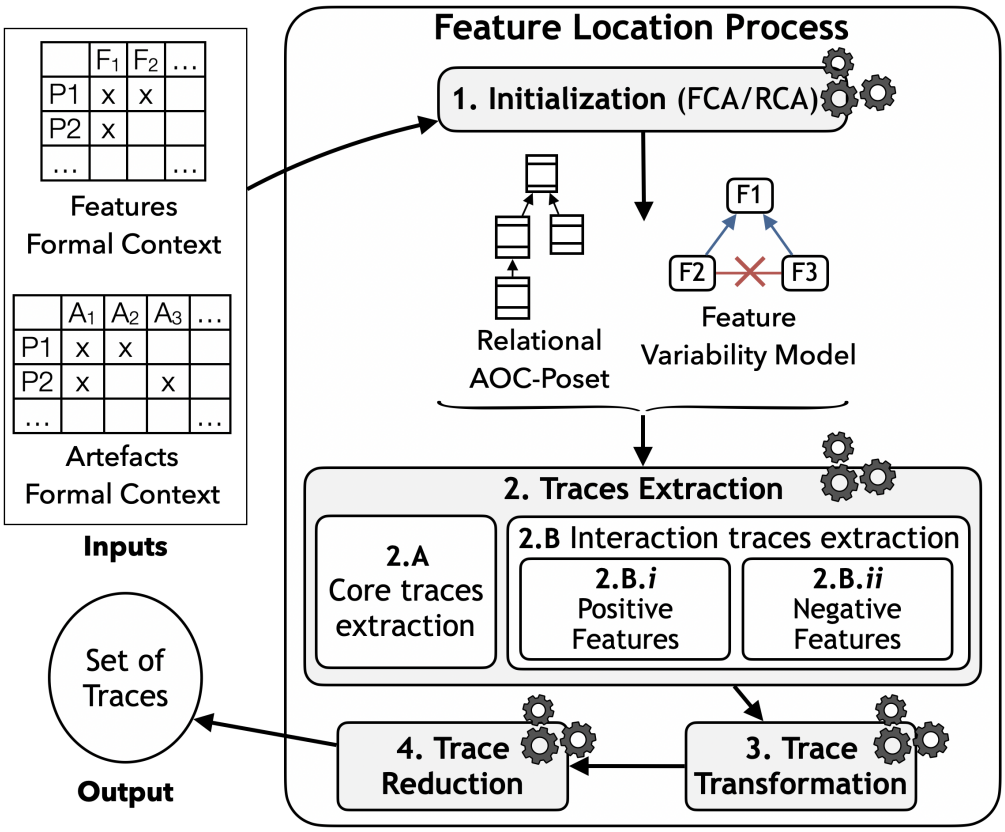

Figure 2: Our Feature Location Technique Overview

4.1 Initialization

This stage takes as inputs two tables: FC_{pa} and FC_{pf}. It builds FC_p, and two Relational Contexts (RC) for $FC_p \rightarrow FC_{pa}$ and $FC_p \rightarrow FC_{pf}$, to obtain the whole RCF. It applies RCA to the RCF and outputs the AOC-Posets. For the motivating example, these are in Figures 3, 4 and 5, respectively called AOC_{art} (products to artefacts), AOC_{feat} (products to features) and AOC_{rel} (products to combined artefacts and features).

4.2 Traces Extraction

The traces extraction stage consists of associating artefacts to their corresponding features, using AOC_{rel}. In AOC_{rel}, the artefacts are already gathered as groups, inside introductory concepts. These groups represent co-occurring artefacts that always appear together in the set of products. The AOC_{rel} contains two concepts' types: concepts that *Introduce Artefacts and Features*, in the set called $Concepts_{iaf}$; and concepts that only *Introduce Artefacts*, in the set called $Concepts_{ia}$. In the following, we sometimes will use $concept_{iaf}$ (resp. $concept_{ia}$) to denote any element of $Concepts_{iaf}$ (resp. $Concepts_{ia}$). Note that since there is always a possible implementation for the feature, there is no concept inside AOC_{rel} that can only introduce features.

Our feature location is defined as matching the right artefact group to the right features or feature interactions. Our approach, similar to those of previous works [38, 39] aims at finding a correspondence between a set of features and an artefact group when both appear in the same product.

Since products implement features, we can describe the presence of an artefact group (in a product) according to the products' features. However, some artefact groups can also be included because of the absence of particular features. Therefore, to trace the artefacts to the features or feature interactions, we define the notion of *apparition context* as a set of positive features (selected in the product) and negative features (not selected). An apparition context is thus a condition, in terms of features, under which an artefact must appear (or not) in a product. Hence, we represent a trace as a set of apparition contexts, forming a global condition to the presence of the artefact groups.

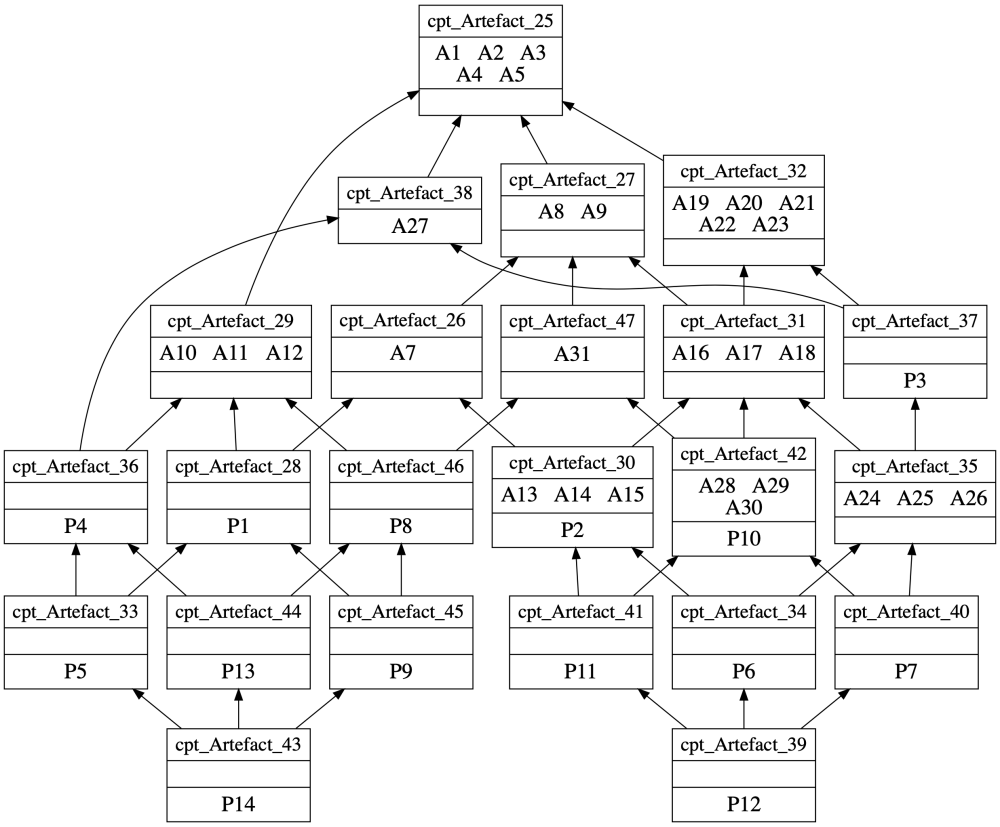

Figure 3: Artefacts AOC-Poset AOC_{art} built from FC_{pa}

4.2.1 (A) Core Traces Extraction

Algorithm 1: Core_Traces_Extraction

Data: AOC_{rel}: the relational AOC-Poset
Result: $CoreTraces$: the set of core traces

```
1 begin
2     CoreTraces ← [];
3     for each concept cpt ∈ Concepts_iaf, in AOC_rel do
4         ac ← new ApparitionContext(cpt.features);
5         trace ← new CoreTrace(cpt.artefacts, ac);
6         CoreTraces ← CoreTraces + trace
7     return CoreTraces
```

As previously mentioned, the AOC_{rel} has concepts (denoted by cpt) that gather features and artefacts (in $Concepts_{iaf}$) that share the same product set. Thus, an artefact group from a $concept_{iaf}$ defines the *core implementation* of this concept's feature. As such, these artefacts are the minimal set of artefacts required for this feature to work. Moreover, since the artefact group and the features appear together in a $concept_{iaf}$, there is only one possible apparition context (ac) to associate with these artefacts. This apparition context contains only the concept's features, as positive ones.

For example, a trace associating an apparition context containing the Scanner feature and the artefact A27 is extracted from the $concept_{iaf}$ named *Concept_FCp_78* in AOC_{rel} (cf. Figure 5). We call the traces extracted from this type of association *core traces*.

Algorithm 1 represents the core traces extraction process. For each concept cpt in the set of $Concepts_{iaf}$ (Line 3), a new apparition context composed of the features introduced by cpt is created (Line 4). Then, a new core trace associating this apparition context with the artefacts group introduced by cpt is formed (Line 5).

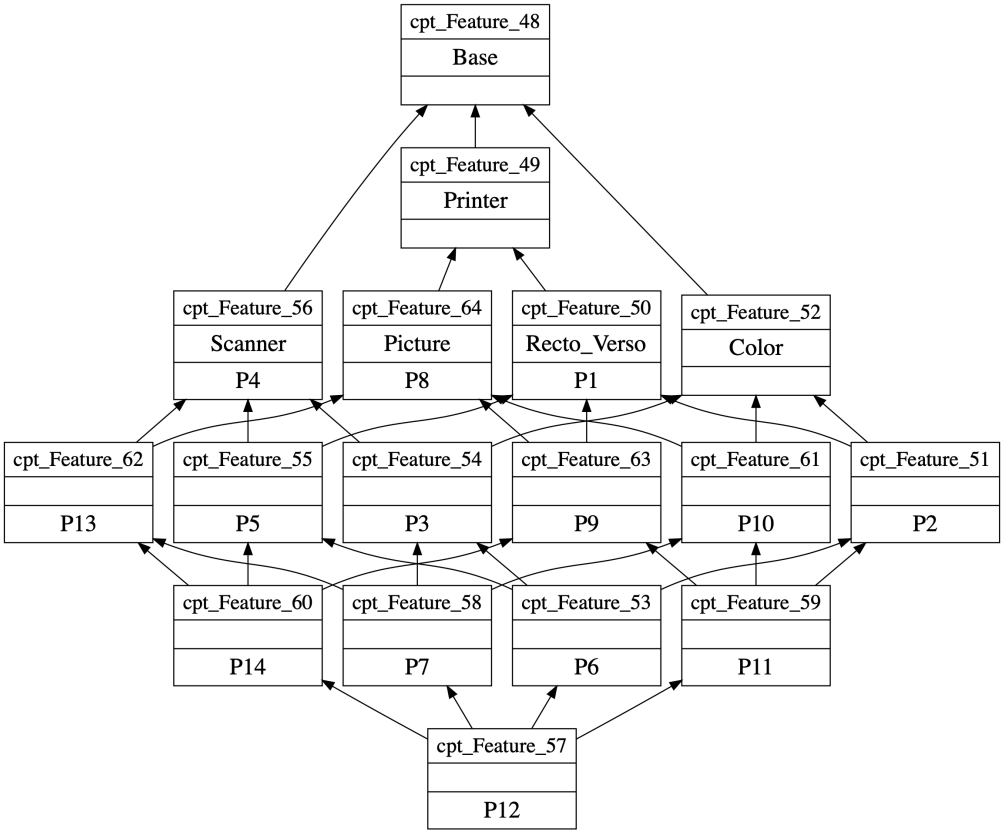

Figure 4: Features AOC-Poset AOC_{feat} built from FC_{pf}

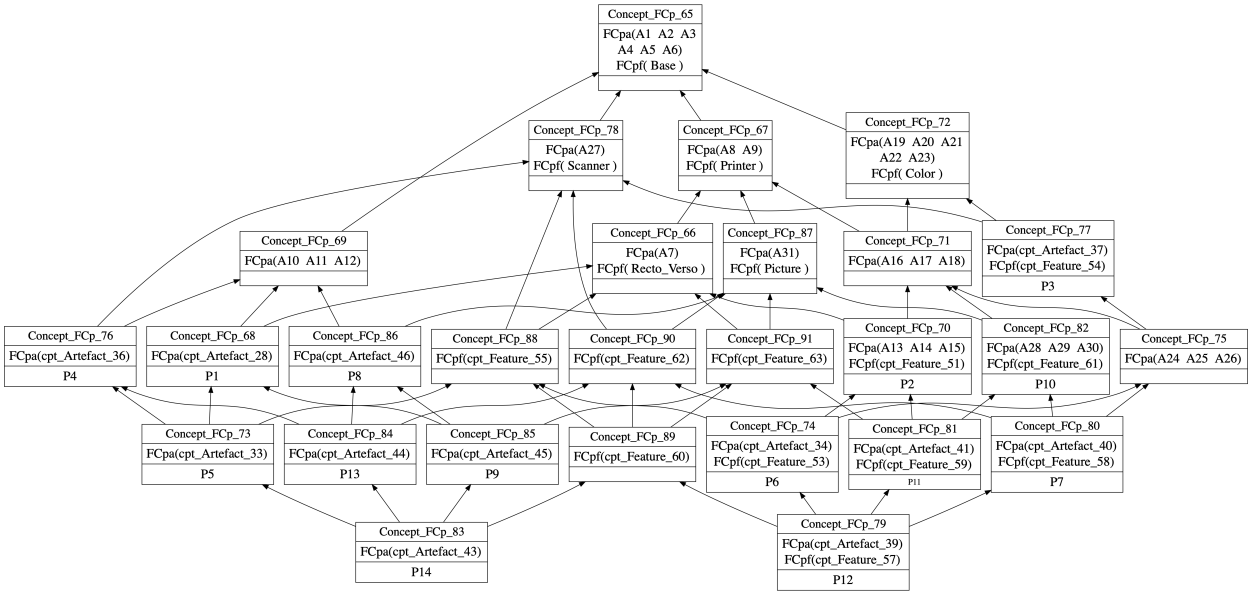

Figure 5: Relational AOC-Poset AOC_{rel}

Through the AOC_{rel} construction, each feature is introduced inside one distinct element of $Concepts_{iaf}$. Thus, at the end of this stage, all the features have been associated to their core implementation (i.e. artefact group) by a core trace. However, some artefact groups are yet to be associated; they are the ones introduced in elements of $Concepts_{ia}$.

4.2.2 (B) Interaction Traces Extraction

Since all features have already been traced to their core implementation (i.e. artefacts), the remaining artefact groups must correspond to specific implementations that occur when multiple features are interacting. Contrarily to a $concept_{iaf}$ of AOC_{rel}, a $concept_{ia}$ does not explicitly associate an artefact group to a feature interaction.

The traces that we search here are called *interaction traces*. Once again, finding the interaction traces consists of extracting the apparition contexts of an artefact group for a specific $concept_{ia}$. These apparition contexts will be determined by retrieving the features of each product where the group of artefacts in question appeared. The reduction stage will then remove the features that would be in excess in the interaction traces (see Section 4.4).

Moreover, some apparition contexts will have negative features this time, since a group of artefacts can be needed in the absence of a feature in a product (like a default version of a feature).

The negative portion of an apparition context is found by searching for all the mutual exclusions that a $concept_{ia}$ has with any other $concept_{iaf}$ in AOC_{rel}. Because artefacts of a $concept_{iaf}$ are the core implementation of a feature, these mutual exclusions give us the features that are conflicting with the presence of the $concept_{ia}$'s artefacts. To find the mutual exclusions between the concepts of an AOC-Poset, we rely on a technique proposed by Carbonnel et al. in [36].

This stage is divided into two parts: (i) the extraction of positive features of all apparition contexts, and (ii) the extraction of negative features for these apparition contexts.

Algorithm 2: Positive_Features_Extraction

Data: AOC_{rel}: the relational AOC-Poset
Result: $InteractionTraces$: the set of interaction traces with their apparition contexts (ac) containing their positive features

```
1  begin
2      InteractionTraces ← [];
3      for each concept cpt_x ∈ Concepts_ia, in AOC_rel do
4          productsInCpt ← getExtentOf(cpt_x, AOC_rel);
5          trace ← new InteractionTrace(cpt_x.artefacts);
6          for each product P ∈ productsInCpt do
7              ac ← new ApparitionContext(P.features);
8              trace ← trace + ac
9          InteractionTraces ← InteractionTraces + trace
10     return InteractionTraces
```

(i) Extracting the Positive Features. Algorithm 2 describes the creation of new apparition contexts ac with their positive features. For each concept cpt_x in $Concepts_{ia}$ (Line 3), it comes down to finding the features of each product that contains the artefact group $cpt_x.artefacts$ associated to cpt_x (Line 4).

First, this consists of getting all the products from the full extent of concept cpt_x. Then, for each product P, an apparition context ac is created, and contains all the features in P as positive features. Finally, this ac is added to the interaction trace of $cpt_x.artefacts$ (Lines 5–8).

For example, *Concept_FCp_75* (or cpt_{75}) in AOC_{rel} (right hand side of Figure 5) is in $Concepts_{ia}$. From the extent of cpt_{75}, the products that contain the artefact group ($products\ InCpt$) are $\{P6, P7, P12\}$ (see the sub-concepts of cpt_{75}). Thus artefact group of cpt_{75} is associated to three apparition contexts. $P6$ configuration is $P6_{conf} = \{Base, Printer, Scanner, Recto\_Verso, Color\}$, therefore the positive features of the first apparition context of cpt_{75} are the ones from $P6_{conf}$. And its other two apparition contexts mention the features of $P7$ and $P12$.

(ii) Extracting the Negative Features. Algorithm 3 describes how the negative features of each apparition context are extracted. First, for each cpt_x that is in $Concepts_{ia}$, it extracts all the $Concepts_{iaf}$ elements that are in mutual exclusion ($Concepts_{mutex}$) with cpt_x (Lines 3–4).

Then, the features introduced by a $Concepts_{mutex}$ element are added as negative features to all the apparition contexts of the trace associated to cpt_x (Lines 5–6).

Algorithm 3: Negative_Features_Extraction

Data: $InteractionTraces$: the interaction traces whose apparition contexts contain their positive features; and AOC_{rel}: the relational AOC-Poset
Result: $InteractionTraces$: where interaction trace are completed with their negative features (if necessary).

```
begin
    for each trace T ∈ InteractionTraces do
        cpt_x ← get the concept with T's artefacts as attributes, in AOC_rel;
        Concepts_mutex ← get all the Concepts_iaf elements in mutual exclusion with cpt_x;
        for each concept cpt_m ∈ Concepts_mutex do
            T.addNegativeFeatures(cpt_m.features);
    return InteractionTraces
```

Note that the extent of a concept is computed from its sub-concepts. Thus if a concept cpt_x is in mutual exclusion with another concept cpt_m, so are cpt_x's sub-concepts (as there is no common sub-concept between cpt_x and cpt_m [28]). Therefore, all apparition contexts extracted from cpt_x will share the same negative features.

For example, there is a mutual exclusion between the concepts *Concept_FCp_69* and *Concept_FCp_72*, respectively named cpt_{69} and cpt_{72}, in AOC_{rel}. Concept cpt_{72} introduces the Color feature while cpt_{69} introduces only artefacts. Therefore, the artefact group introduced by cpt_{69} cannot be present when the Color feature is selected. So, every apparition context of cpt_{69} will have Color as a negative feature.

Since our algorithm has visited all the concepts that introduce artefact groups (from $Concepts_{iaf}$ and $Concepts_{ia}$), every artefact is traced to features. We expose the interaction traces extracted from our PrinterSPL example in Table 3.

4.3 Traces Transformation

This section presents how traces are transformed into a *well-formed formula* of the propositional logic. These types of formulas are often used when generating a product from the SPL, mainly when using a preprocessing approach [6, 7]. In such a case, the formula is formed by first considering a feature as an atomic proposition. To interpret a formula, an atomic proposition (that represents a feature) is assigned to *True* if its corresponding feature is selected to be in the product. For an unselected feature, its corresponding proposition in the formula is assigned to *False*. Thus, artefacts associated with a formula (i.e. the trace) are included inside a product implementation if their formula is interpreted as *True*.

To construct a formula, we set each apparition context as a set of features in conjunction. For example, the artefacts group in $Concept\_FCp\_69$ (cf. Figure 5) associated to trace T_{11} has seven apparition contexts (see trace T_{11} in Table 3).

In our example, the formula F associated to the apparition context ac_1 is $F(ac_1) = Base \wedge Recto\_Verso \wedge Printer \wedge \neg Color$. For ac_2, its formula is $F(ac_2) = Base \wedge Scanner \wedge Recto\_Verso \wedge Printer \wedge \neg Color$, and so on. The set of apparition contexts of a specific trace represents an artefacts group's distinct possible appearance conditions. As such, they are joined by disjunctions (i.e. OR). Thereby, the formula of T_{11} is the disjunction of all T_{11}'s apparition contexts ($F(T_{11}) = F(ac_1) \vee F(ac_2) \vee$ [...] $F(ac_7)$), as shown in Formula 1.

$$
\begin{aligned}
\boldsymbol{F(T_{11})} = ((&Base \wedge Recto\_Verso \wedge Printer \wedge \neg Color) \\
\vee\ (&Base \wedge Scanner \wedge Recto\_Verso \wedge Printer \wedge \neg Color) \\
\vee\ (&Base \wedge Scanner \wedge Recto\_Verso \wedge Printer \\
&\wedge\ Picture \wedge \neg Color) \\
\vee\ (&Base \wedge Recto\_Verso \wedge Printer \wedge\ Picture \wedge \neg Color) \\
\vee\ (&Base \wedge Scanner \wedge \neg Color) \\
\vee\ (&Base \wedge Scanner \wedge Printer \wedge Picture \wedge \neg Color) \\
\vee\ (&Base \wedge Printer \wedge Picture \wedge \neg Color))
\end{aligned}
\tag{1}
$$

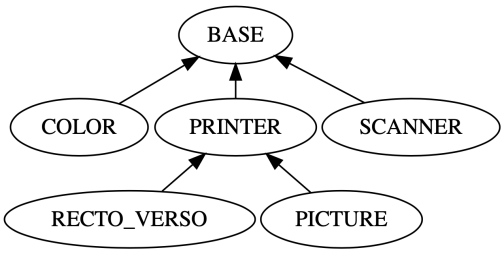

Figure 6: Feature Variability Model, extracted from the AOC-Poset of Fig. 5.

4.4 Traces Reduction

With the trace transformation, the output formula of each trace can have a significant size (see Formula 1). Furthermore, most of the information inside the formula is redundant, making the overall formula challenging to read and thus to comprehend. To make these formulas more comprehensible, we propose a reduction phase that removes unnecessary features involved in a formula. Since the formulas represent the traces, reducing the size of the formulas amounts to reduce the traces' size. In addition to making a trace more understandable, this also exposes the main interacting features in that trace more accurately.

Our reduction process is divided into four steps: i) a simplification step using the Boolean algebra; ii) the removal of the common features; iii) the factoring of the formula; and iv) the deletion of the implied features.

Note that for the reduction ii) and iv), we rely on an extracted variability model, depicted in Figure 6. The feature variability model is a diagram that represents the variability constraints among the features, extracted from the AOC-Poset AOC_{rel} (or from AOC_{feat}, since both contain the same constraints). We obtain this model by following the process described in [30], which is inspired by the process introduced by [28]. As we shall see, the main idea of step ii) and iv), is to remove from a formula some information that is already contained in the variability model of Figure 6.

Step i) consists in a reduction of the logical formulas following the different *laws* of the Boolean algebra [27] (i.e. Associativity, Commutativity, Distributivity, Identity, Idempotence, Absorption, etc.). Following these laws, Formula 1 associated to the trace T_{11} is transformed in Formula 2:

$$\begin{aligned} \boldsymbol{F(T_{11})} = ((&Base \wedge Recto\_Verso \wedge Printer \wedge \neg Color) \\ \vee\ &(Base \wedge Scanner \wedge \neg Color) \\ \vee\ &(Base \wedge Printer \wedge Picture \wedge \neg Color) \end{aligned} \tag{2}$$

In step ii), we remove the common features of the formulas. Since the common features are found in all products, i.e., mandatory, their inclusion in a formula is redundant information. We can find the common features at the top of the variability model, e.g. it is $Base$ in Figure 6. This step is trivial, and in our example of trace T_{11}, Formula 2 becomes Formula 3:

$$\begin{aligned} \boldsymbol{F(T_{11})} = ((&Recto\_Verso \wedge Printer \wedge \neg Color) \\ \vee\ &(Scanner \wedge \neg Color) \\ \vee\ &(Printer \wedge Picture \wedge \neg Color) \end{aligned} \tag{3}$$

In step iii) we factorize the formula. Whenever the formula permits it, the factorization helps to reduce the size of the formula further. For our example, the factorization of Formula 3 turns it into Formula 4.

$$\begin{aligned} \boldsymbol{F(T_{11})} = &(Scanner \vee (Printer \wedge (Picture \vee Recto\_Verso))) \\ \wedge\ &\neg Color \end{aligned} \tag{4}$$

In step iv) we are interested by using the features' implication constraints, found in the AOC-Poset, to replace some conjunctions and disjunctions of propositions by a single proposition. The set of implication constraints between features is represented by arrows between feature nodes in Figure 6 (e.g. $Scanner \Rightarrow Base$, $Picture \Rightarrow Printer$).

Table 3: Traces of PrinterSPL before reduction

N^o	**Trace Extraction Results**	
	Ctx	Apparition Contexts
T_1	ac_1	$\{Color\}$
T_2	ac_1	$\{Recto\_Verso\}$
T_3	ac_1	$\{Scanner\}$
T_4	ac_1	$\{Picture\}$
T_5	ac_1	$\{Printer\}$
T_6	ac_1	$\{Base\}$
T_7	ac_1	$\{Base, Color, Recto\_Verso, Printer, Scanner\}$
	ac_2	$\{Base, Color, Recto\_Verso, Printer, Scanner, Picture\}$
	ac_3	$\{Base, Color, Printer, Scanner, Picture\}$
T_8	ac_1	$\{Base, Color, Recto\_Verso, Printer\}$
	ac_2	$\{Base, Color, Recto\_Verso, Printer, Scanner\}$
	ac_3	$\{Base, Color, Recto\_Verso, Printer, Scanner, Picture\}$
	ac_4	$\{Base, Color, Recto\_Verso, Printer, Picture\}$
T_9	ac_1	$\{Base, Color, Printer, Picture\}$
	ac_2	$\{Base, Color, Printer, Scanner, Picture\}$
	ac_3	$\{Base, Color, Recto\_Verso, Printer, Scanner, Picture\}$
	ac_4	$\{Base, Color, Recto\_Verso, Printer, Picture\}$
T_{10}	ac_1	$\{Base, Color, Recto\_Verso, Printer\}$
	ac_2	$\{Base, Color, Recto\_Verso, Printer, Scanner\}$
	ac_3	$\{Base, Color, Recto\_Verso, Printer, Scanner, Picture\}$
	ac_4	$\{Base, Color, Recto\_Verso, Printer, Picture\}$
	ac_5	$\{Base, Color, Printer, Scanner, Picture\}$
	ac_6	$\{Base, Color, Printer, Picture\}$
T_{11}	ac_1	$\{Base, Recto\_Verso, Printer, not(Color)\}$
	ac_2	$\{Base, Recto\_Verso, Printer, Scanner, not(Color)\}$
	ac_3	$\{Base, Recto\_Verso, Printer, Scanner, Picture, not(Color)\}$
	ac_4	$\{Base, Recto\_Verso, Printer, Picture, not(Color)\}$
	ac_5	$\{Base, Scanner, not(Color)\}$
	ac_6	$\{Base, Printer, Scanner, Picture, not(Color)\}$
	ac_7	$\{Base, Printer, Picture, not(Color)\}$

We use the implication constraints to reduce the formula's size in this order: first by covering the conjunctive clauses, then only after, the disjunctive clauses. So first, we propose to replace conjunctive clauses such as $F_x \wedge F_y$, by F_x if $F_x \Rightarrow F_y$. Since F_x implies F_y, we can expect F_y to be always implemented in a product when F_x is. Thus having both $F_x \wedge F_y$ in the same proposition is redundant information, and we propose to remove F_y.

As a second part, if a formula contains a disjunction of propositions, and all of these propositions are exactly the set of premises of the same implication's conclusion, then we replace the whole disjunction with the conclusion of the premises. For instance, in the formula, we see a clause $Picture \vee Recto\_Verso$. The implication in the feature diagram in Figure 6 shows that both $Picture \Rightarrow Printer$ and $Recto\_Verso \Rightarrow Printer$ are the only implication constraints with $Printer$ as conclusion. This means that since $Picture$ and $Recto\_Verso$ are all the sub-features of $Printer$, having a disjunction between $Picture$ and $Recto\_Verso$ in the formula comes down to only having $Printer$. Therefore, we propose to replace $(Picture \vee Recto\_Verso)$ in our formulas with $Printer$. Formula 5 is the final output of the trace reduction, and Table 6 shows all reduced traces of our example, with their associated artefacts.

$$\boldsymbol{F(T_{11})} = (Scanner \vee Printer) \wedge \neg Color \tag{5}$$

Table 4: Case Study Overview

Case-study	*Features*	*Products*	*Artefacts*	*LoC*
PrinterSPL	6	14	31	74
DrawSPL	5	13	327	589
Elevator1-4	8	16	1055	1903
GameOfLife	17	11	820	1280
ArgoUML-app	10	10	123766	139106

Features: total number of features
Products: number of products for the evaluation
Artefacts: number of identified artefacts
LoC: total number of Lines of Code

5 Evaluation

This section focuses on the evaluation of our feature location technique through its application on five case studies.

First, we present an evaluation to validate the relevance of the traces extracted by our approach. Our hypothesis is as follows: if the extracted traces correctly associate the right artefacts with the right features or feature interactions, we can reconstruct the different sets of artefacts of each product from the traces that will be interpreted with the feature configuration of each product. It will thus be a question of using the configuration of the various products to interpret the logical formulas of each trace. We then convert the set of artefacts from the traces whose interpretation of the formula returned *True*. If this set of artefacts is equivalent to the one found in the products, we can assume that the traces correctly associate the artefacts and the features. Otherwise, it means that some artefacts have been associated with the wrong features or interactions; and thus, our traces are invalid.

In the second part of our evaluation, we present the impact of our reduction process on the traces, by measuring the size of the traces after reduction. We also present the total execution time, from the extraction of the traces to their reduction.

5.1 Data Collection

For the evaluation, we used 5 case studies, all shown in Table 4. Like introduced before, PrinterSPL is a product family created in order to illustrate the approach. DrawSPL is a product line of geometric shape drawing software introduced in [42]. ArgoUML-app is the main plugin of ArgoUML-SPL [2], which is an open-source project transformed into an SPL of UML-Modeling tools. ArgoUML-app has between 1371 and 1502 files of code (depending on the product), over the 2404 files in total in ArgoUML-SPL. Finally, we used two available SPL which are examples in FeatureIDE[3]: Elevator1-4, which simulates different elevator variants; and GameOfLife, an SPL to simulate cellular automaton.

5.2 Experimentation Protocol

5.2.1 Java Prototype

Our entire approach has been implemented inside a Java prototype. We used the tool *RCAExplore* [35] to generate the AOC-Posets from the relational context families. In addition, we used the tool *CLEF*[4] [36] to extract the implication and mutual exclusion constraints of the AOC-Posets, which are necessary to the traces' extraction and reduction phase respectively. We run this prototype on a 13 inch MacBook Pro, 2.3GHz Intel 4-Cores i5, 16Go RAM and 256 SSD.

5.2.2 Protocol

We apply the following experimental protocol over each case study shown in Table 4.

(1) **Trace Validity.** Our first evaluation consists of determining the validity of our trace extraction. We depicted it in Figure 7. For a particular case study (CS), we take in entry the formal context made from all the products' artefacts (as in Figure 2), and the one made from the product's features (as in Figure 1). From here, we extract the traces using

[2]https://github.com/marcusvnac/argouml-spl
[3]https://featureide.github.io/
[4]https://gite.lirmm.fr/jcarbonnel/CLEF

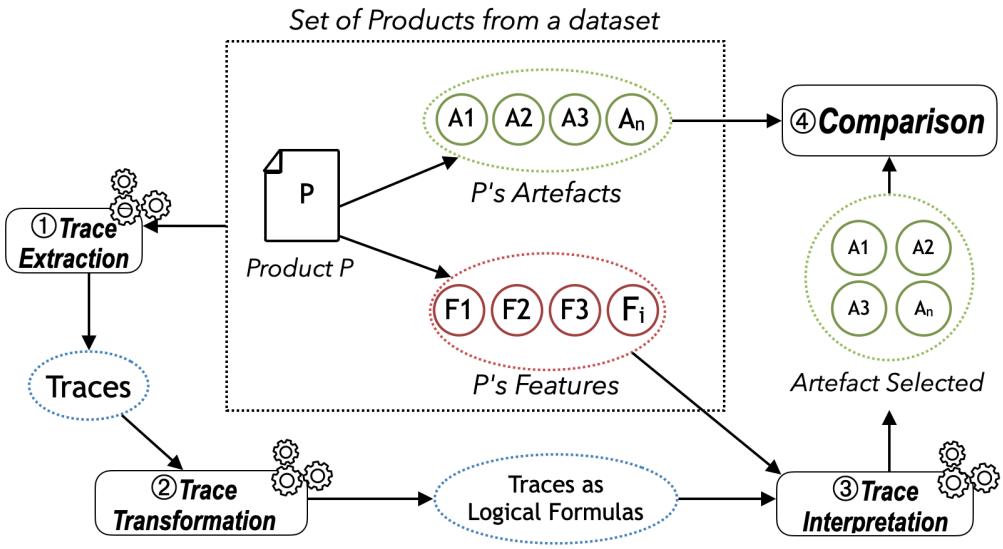

Figure 7: Our Trace validity Protocol

Table 5: Case Study Average Results Overview

Case-study	#Traces	*Average # of non distinct features in traces*	
		Before reduction	After reduction
PrinterSPL	11	10.64	1.64 (-84.62%)
DrawSPL	15	9.73	2.27 (-76.71%)
Elevator1-4	24	18.33	5.13 (-72.05%)
GameOfLife	9	15.67	4.11 (-73.76%)
ArgoUML-app	33	19.70	13.45 (-31.69%)

our feature location technique, and we transform them into logical formulas. After that, for each product P of a case study CS, we try to re-construct the artefact configuration of P, using P's features to interpret all the extracted traces. Finally, we compare the artefacts set selected by the trace interpretation with the original artefacts set of P. A 100% match means that our extracted traces have associated the artefacts to the right features.

② **Trace Reduction.** The goal of this second step is to highlight the impact of the trace reduction. We measure the traces' size by counting the number of features (not distinct) present in the trace. Thereby, right after the trace extraction, we collect the size of each trace before and after reduction.

5.3 Results

For the trace validity, every artefact obtained from the trace has had a perfect match with the ones from the original product. These results seem to validate our FL technique over our set of case studies.

For the trace reduction, an overview of the average traces' sizes (before/after) of each case study is proposed in Table 5. Based on our results over these case studies, our traces are on average reduced by 67,77%. We observed that our reduction has a greater impact on some case studies than on others. For example, DrawSPL's traces were reduced by an average of 76.71%, while ArgoUML-app only sees a reduction of 31.69% on average. We assume that this difference is due to the initial proportion of the traces, as ArgoUML-app traces are way more substantial initially. To summarize Table 5, all extracted traces have benefited from our trace reduction.

Moreover, in terms of execution time for the entire process in Figure 7, our approach took 3 sec for PrinterSPL, 4 sec for DrawSPL, 10 sec for Elevator1-4, 11 sec for GameOfLife and finally 26 minutes and 9 sec for ArgoUML-app. These execution times surpass what a team of developers could do by manually locating the features inside the products.

Table 6: Reduced traces of PrinterSPL as logical formulas

N^o	**Logical Formula**	**Artefacts**
T_1	$Color$	A_{19} to A_{23}
T_2	$Recto\_Verso$	A_7
T_3	$Scanner$	A_{27}
T_4	$Picture$	A_{31}
T_5	$Printer$	A_8, A_9
T_6	$Base$	A_1 to A_6
T_7	$Color \wedge Scanner \wedge Printer$	A_{24}, A_{25}, A_{26}
T_8	$Color \wedge Recto\_Verso$	A_{13}, A_{14}, A_{15}
T_9	$Color \wedge Picture$	A_{28}, A_{29}, A_{30}
T_{10}	$Color \wedge Printer$	A_{16}, A_{17}, A_{18}
T_{11}	$(Printer \vee Scanner) \wedge \neg Color$	A_{10}, A_{11}, A_{12}

5.4 Threats to Validity

Even though the results are promising, the evaluation was only achieved on a short number of product families, with few features, and so the results need to be taken cautiously.

Unfortunately, our current prototype only works on Java code since we had to rely on an automated technique to build the artifacts formal context (FC) at fine-grained level [30], and it is currently limited to Java. Moreover, we struggled to find SPLs or products with an open Java source code and with more than a dozen features. Thus, to validate our FL technique, we either need to prove our algorithms or to test them on a larger dataset. However we are confident that our FL technique should work with any programming language and with any artefact granularity, since our analyses are solely based on the formal contexts.

We must also stress the fact that our technique enables us to find the traces, which in themselves enable us to find all the artefacts of the products. However, we do not reconstruct the product code. This remains consistent with our initial hypothesis: we consider as input the formal contexts of the products (artefacts and features), but not their source code.

As a limitation, our approach can fail to precisely locate a feature implementation in a case of co-occurring features in the products set. This limitation is shared by other approaches that based the FL on the variability analysis of the product [38, 39] To solve this issue, [38, 39] have relied on lexical analysis, using techniques such as LSI [33], to find a lexical proximity between the artefacts and a feature description. However, the results of the lexical analysis in this context are unreliable [34]. Our future works should focus on this problem.

Finally, our technique does not distinguish concrete and abstract features. Abstract features are usually ”refined” into concrete sub-features. We understand that an abstract feature lacks an implementation in the products, but its concrete sub-features do. If abstract features were mentioned in the list of product features, they would necessarily co-occur with their sub-features. Thus it would bring back to the *feature co-occurrence* problem. If we can distinguish the abstract features from the concrete ones, we could reduce the co-occurrences by only keeping the concrete features in the trace. Otherwise, our technique cannot address the co-occurrence problem.

6 Related Work

This section presents research work on feature location techniques. The literature often groups FL approaches that apply FL on a single product with those that apply it on a family of product variants [21, 22, 20]. Since we are interested in FL on a family of products, we propose here a related work on the studies which share this topic. However, we mention that [12, 11, 10] use FCA for FL in a single software.

6.1 Existing Static Feature Location Techniques using FCA

Both Xue and al. [38] and Salman and al. [39, 19] proposed a FL combining FCA and Latent Semantic Indexing (LSI). Their approaches compute Minimal Disjoint Sets (MDS) to divide the search space (i.e. the variable sets of features and artefacts) and apply FCA on these MDS. Then, the feature sub-spaces are mapped to the artefact sub-spaces using LSI. In order for LSI to work with a maximum of information, the artefact granularity has to be coarse (at files level). This makes their approach unsuitable when applied to fined-grained artefacts (at statement level).

Our work is different on many points. Firstly, our technique can be performed without taking into account the level of granularity of the artefacts. Secondly, they give MDS as the entry for FCA. On the contrary, we give the formal context (artefacts and features) to FCA/RCA. Thirdly, they use textual analysis, assuming that a lexical proximity exists between the features and the artefacts, whereas we only rely on FCA/RCA. Finally, their approaches cannot recover the original products' artefacts since their traces do not have precision and recall of 100%.

Martinez et al. [44, 43] proposed a generic (supporting multiple artefacts types) and adaptable (possibility to add your own adapters) framework for the construction of SPLs from product variants, called *BUT4Reuse*. Their approach consists of determining common *blocks* of artefacts (i.e. artefacts present in all products) and variable blocks (i.e. artefacts present in some but not all products). To associate these blocks to features, they use a heuristic based on the fact that a block is associated with a feature if their appearance in products is co-occurring. Although it is not mentioned in their paper, it seems that the *BUT4Reuse* implementation of this heuristic may rely on FCA. However, in their approach, the artefact granularity is coarser, the statements inside methods are ignored, which impacts the precision of their traces. Moreover, the blocks representing interaction are treated as individual *unnamed* features. Therefore, when creating a product, the developers must select the blocks manually to compose the products. They also need to manually include the blocks corresponding to the implementation of an interaction.

Compared to our approach, none of the existing FCA-based feature location techniques considered the features interactions. Moreover, these approaches are defined to work with a coarse granularity (i.e. Class/Method) and do not provide solutions for smaller granularity (i.e. statement level). Contrarily, our technique can be used at any granularity level without changing the validity of the extracted traces (we evaluated it at statement-level in Section 5).

6.2 Existing Feature Location without FCA

Fischer et al. and Linsbauer et al. [8, 3] proposed a FL on a family of product variants in the context of a tool called *ECCO*. We compare our technique with the one detailed in [3]. Like our technique, theirs extracts sufficiently precise traces to recover all the artefacts of the original products after extraction. Like in [8, 3], we tested our FL technique on DrawSPL, GameOfLife and a part of ArgoUML. However, the versions used are different (not always the same number of features or products) since we are unable to find the same data used by their study.

Linsbauer et al. use the notion of modules that seems to roughly correspond to conjunctive clauses of positive or negative features. They describe their products by sets of *artefact-trees*, which correspond to pieces of the source code Abstract Syntax Tree (AST). By contrast, our FL is applied independently from the artefact type (whether the artefacts in the formal context represent source code, images, text, etc.). Their traces are complex associations between four-tuples of module sets (including those representing minimal and maximal traces) on one side and artefact-trees on the other side. Whereas our traces associate a logical formula, made of features names, with a set of artefacts; each logical formula exactly describes the apparition context of these artefacts.

ECCO trace extraction is an iterative process executed over each pair of products. It computes all possible modules pair-wise, gathering the positive and negative features within each module. By accumulating modules, their traces are refined to become more and more precise. Whereas in our techniques, the FL is done in "one-shot". First, the computation of the AOC-poset allows us to restrict our analysis to non-empty intersections of the products only (intersection of features and resp. artefacts) [5]. Then we extract core and interaction traces from the AOC-Poset; we rely on Boolean algebra and the extracted variability model to perform the trace reduction.

Finally, both techniques can be applied to migrate a family of product variants to an SPL. In that regard, variability models have to be built alongside the FL. ECCO proposes a feature dependency graph to model their variability constraints. However, this graph is computed from the source code structural dependency in the feature's code implementation, making ECCO restricted to source code artefacts. In our technique, a variability model (see Figure 6) can be found in the same structure as the one used for trace extraction, which is the AOC-Poset. By construction, the AOC-Poset contains all the valid feature-models ([32]) regarding the given formal context products-features [28], but expresses only logical semantics. In addition, the AOC-Poset based methodology is generic and applies to any product description. Thus it is not limited to source code or feature sets.

```
1.  // Before Reduction of T10
2.  /*#if ((COLOR && BASE && RECTO_VERSO && PRINTER)
3.  || (COLOR && BASE && RECTO_VERSO && PRINTER && SCANNER)
4.  || (COLOR && BASE && RECTO_VERSO && PRINTER && SCANNER
5.  && PICTURE)
6.  || (COLOR && BASE && RECTO_VERSO && PRINTER && PICTURE)
7.  || (COLOR && BASE && SCANNER && PICTURE && PRINTER)
8.  || (COLOR && BASE && PICTURE && PRINTER))
9.  */
10. switch(printerType)
11. {  […]
 …  -----------------------------------
 …
1.  // After Reduction of T10
2.  //#if COLOR && PRINTER
3.  switch(printerType)
4.  { […]
```

Figure 8: The impact of the trace reduction when generating annotations for artefacts of T_{10} (see Table 6), for the SPL implementation of PrinterSPL.

7 Conclusion & Perspectives

In this paper, we have presented our proposed Feature Location technique using FCA/RCA. We have demonstrated over different case studies that our technique can extract the traces between the artefacts and the features, as well as the ones between the artefacts and the feature interactions. As a result, for each product of a case study, we are able to retrieve its set of artefacts by interpreting the extracted traces with the product's features. We have also presented our method to reduce the extracted traces. The output traces contain far fewer features and are thus more comprehensible.

As a perspective, we plan to reconstruct the code of the initial products, using the traces extracted by our technique. For this, we can explore the FCA Pattern Structure [4], which allows us to use FCA while keeping an order among the attributes (a.k.a. the artefacts of the products). This could be the first way to keep a structure in the products. The second would be to apply our technique in the method proposed by Hlad et al [30]. This method proposes creating a common Artefact-Tree for all the products, in which all the artefacts of the SPL are stored. Using our technique, we can retrieve the implementations of the features and their interactions among these artefacts and use the Artefact-Tree structure to retrieve the products. Moreover, the same method proposes the generation of an annotated code as an implementation of the SPL. Our traces, once transformed into a logical formula, can be used to generate these annotations. We made an example in Figure 8 of how the traces can be used to generate annotations, but also how our trace reduction can improve the reading and comprehension of a trace. Moreover, in this context trace conciseness will be a major concern, thus we plan to investigate the nested property of the annotations to further reduce the extracted traces.

Data Availability Statement

Data from our experimentation can be found in [1]. It contains all five case studies, formal contexts, AOC-Posets, and files to visualize the extracted traces.

References

[1] Nicolas, H., Bérénice, L., Huchard, M. & Seriai, A. Leveraging Relational Concept Analysis for Automated Feature Location in Software Product Lines - Artefacts DataSet. (Zenodo,2021,10), https://doi.org/10.5281/zenodo.5544353

[2] Ignatov, D. Introduction to Formal Concept Analysis and Its Applications in Information Retrieval and Related Fields. *CoRR*. **abs/1703.02819** (2017), http://arxiv.org/abs/1703.02819

[3] Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. Variability extraction and modeling for product variants. *Softw. Syst. Model.*. **16**, 1179-1199 (2017), https://doi.org/10.1007/s10270-015-0512-y

[4] Ganter, B. & Kuznetsov, S. Pattern Structures and Their Projections. *Conceptual Structures: Broadening The Base, 9th International Conference On Conceptual Structures, ICCS 2001, Stanford, CA, USA, July 30-August 3, 2001, Proceedings*. **2120** pp. 129-142 (2001), https://doi.org/10.1007/3-540-44583-8

[5] Berry, A., Gutierrez, A., Huchard, M., Napoli, A. & Sigayret, A. Hermes: a simple and efficient algorithm for building the AOC-poset of a binary relation. *Ann. Math. Artif. Intell.*. **72**, 45-71 (2014), https://doi.org/10.1007/s10472-014-9418-6

[6] Aleixo, F., Freire, M., Alencar, D., Campos, E. & Kulesza, U. A Comparative Study of Compositional and Annotative Modelling Approaches for Software Process Lines. *2012 26th Brazilian Symposium On Software Engineering*. pp. 51-60 (2012,9), http://ieeexplore.ieee.org/document/6337893/

[7] Gacek, C. & Anastasopoules, M. Implementing product line variabilities. *Proceedings Of The ACM SIGSOFT Symposium On Software Reusability: Putting Software Reuse In Context, SSR 2001, May 18-20, 2001, Toronto, Ontario, Canada*. pp. 109-117 (2001), https://doi.org/10.1145/375212.375269

[8] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. The ECCO Tool: Extraction and Composition for Clone-and-own. *Proceedings Of The 37th International Conference On Software Engineering - Volume 2*. pp. 665-668 (2015), http://dl.acm.org/citation.cfm?id=2819009.2819132

[9] Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. Variability extraction and modeling for product variants. *Proceeedings Of The 22nd International Systems And Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*. pp. 250 (2018), https://doi.org/10.1145/3233027.3236396

[10] Poshyvanyk, D. & Marcus, A. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. *15th International Conference On Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada*. pp. 37-48 (2007), https://doi.org/10.1109/ICPC.2007.13

[11] Koschke, R. & Quante, J. On dynamic feature location. *20th IEEE/ACM International Conference On Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. pp. 86-95 (2005), https://doi.org/10.1145/1101908.1101923

[12] Eisenbarth, T., Koschke, R. & Simon, D. Locating features in source code. *IEEE Transactions On Software Engineering*. **29**, 210-224 (2003)

[13] Ryssel, U., Ploennigs, J. & Kabitzsch, K. Extraction of feature models from formal contexts. *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*. pp. 4 (2011)

[14] Loesch, F. & Ploedereder, E. Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. *11th European Conference On Software Maintenance And Reengineering, Software Evolution In Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands*. pp. 159-170 (2007)

[15] Ganter, B. & Wille, R. Formal Concept Analysis - Mathematical Foundations. (Springer,1999)

[16] Hacene, M., Huchard, M., Napoli, A. & Valtchev, P. Relational concept analysis: mining concept lattices from multi-relational data. *Ann. Math. Artif. Intell.*. **67**, 81-108 (2013)

[17] Kästner, C., Apel, S. & Kuhlemann, M. Granularity in software product lines. *Proceedings Of The 13th International Conference On Software Engineering - ICSE '08*. pp. 311 (2008), http://portal.acm.org/citation.cfm?doid=1368088.1368131

[18] Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S. & Salman, H. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. *Safe And Secure Software Reuse - 13th International Conference On Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Prxoceedings*. pp. 302-307 (2013), https://doi.org/10.1007/978-3-642-38977-1_22

[19] Eyal-Salman, H., Seriai, A. & Dony, C. Feature Location in a Collection of Product Variants: Combining Information Retrieval and Hierarchical Clustering. *SEKE: Software Engineering And Knowledge Engineering*. pp. 426-430 (2014,7), https://hal-lirmm.ccsd.cnrs.fr/lirmm-01291261

[20] Rubin, J. & Chechik, M. A Survey of Feature Location Techniques. *Domain Engineering, Product Lines, Languages, And Conceptual Models*. pp. 29-58 (2013), https://doi.org/10.1007/978-3-642-36654-3_2

[21] Dit, B., Revelle, M., Gethers, M. & Poshyvanyk, D. Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process.*. **25**, 53-95 (2013)

[22] Razzaq, A., Wasala, A., Exton, C. & Buckley, J. The State of Empirical Evaluation in Static Feature Location. *ACM Trans. Softw. Eng. Methodol.*. **28** (2018,12), https://doi.org/10.1145/3280988

[23] Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C. & Garvin, B. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. *Proceedings Of The 5th International Workshop On Feature-Oriented Software Development*. pp. 1-8 (2013), https://doi.org/10.1145/2528265.2528267, event-place: Indianapolis, Indiana, USA

[24] Soares, L., Schobbens, P., Machado, I. & Almeida, E. Feature interaction in software product line engineering: A systematic mapping study. *Inf. Softw. Technol.*. **98** pp. 44-58 (2018)

[25] Pohl, K., Böckle, G. & Linden, F. Software Product Line Engineering - Foundations, Principles, and Techniques. (Springer,2005)

[26] Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, A. Feature-Oriented Domain Analysis (FODA) Feasibility Study:. (Defense Technical Information Center,1990,11), http://www.dtic.mil/docs/citations/ADA235785

[27] Boole, G. The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning. (Cambridge University Press,2009)

[28] Carbonnel, J., Huchard, M. & Nebut, C. Modelling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions. *J. Syst. Softw.*. **152** pp. 1-23 (2019), https://doi.org/10.1016/j.jss.2019.02.027

[29] Carbonnel, J. L'analyse formelle de concepts: un cadre structurel pour l'étude de la variabilité de familles de logiciels. (Formal concept analysis: a structural framework to study variability in software families). (University of Montpellier, France,2018), https://tel.archives-ouvertes.fr/tel-02117875

[30] Hlad, N., Abdelhak-Djamel, S. & Christophe, D. IsiSPL: Toward An Automated Reactive Approach to Build Software Product Lines. (2021)

[31] Kästner, C. & Apel, S. Integrating Compositional and Annotative Approaches for Product Line Engineering.

[32] Batory, D. & O'Malley, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.*. **1**, 355-398 (1992)

[33] Deerwester, S., Dumais, S., Landauer, T., Furnas, G. & Harshman, R. Indexing by Latent Semantic Analysis. *J. Am. Soc. Inf. Sci.*. **41**, 391-407 (1990)

[34] Cruz, D., Figueiredo, E. & Martinez, J. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. *Proceedings Of The 13th International Workshop On Variability Modelling Of Software-Intensive Systems*. (2019)

[35] Dolques, X., Braud, A., Huchard, M. & Ber, F. RCAexplore, a FCA based Tool to Explore Relational Data. *Supplementary Proceedings Of ICFCA 2019 Conference And Workshops, Frankfurt, Germany, June 25-28, 2019*. **2378** pp. 55-59 (2019), http://ceur-ws.org/Vol-2378/shortAT5.pdf

[36] Carbonnel, J. CLEF, a Java library to Extract Logical Relationships from Multivalued Contexts. *Supplementary Proceedings Of ICFCA 2019 Conference And Workshops, Frankfurt, Germany, June 25-28, 2019*. **2378** pp. 45-49 (2019), http://ceur-ws.org/Vol-2378/shortAT3.pdf

[37] Carbonnel, J., Huchard, M. & Nebut, C. Exploring the variability of interconnected product families with relational concept analysis. *Proceedings Of The 23rd International Systems And Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*. pp. 90:1-90:8 (2019), https://doi.org/10.1145/3307630.3342407

[38] Xue, Y., Xing, Z. & Jarzabek, S. Feature Location in a Collection of Product Variants. *19th Working Conference On Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. pp. 145-154 (2012), https://doi.org/10.1109/WCRE.2012.24

[39] Salman, H., Seriai, A., Dony, C. & Al-Msie'deen, R. Recovering traceability links between feature models and source code of product variants. *Proceedings Of The VARiability For You Workshop - Variability Modeling Made Useful For Everyone, VARY '12, Innsbruck, Austria, September 30, 2012*. pp. 21-25 (2012), https://doi.org/10.1145/2425415.2425420

[40] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. The ECCO Tool: Extraction and Composition for Clone-and-own. *Proceedings Of The 37th International Conference On Software Engineering - Volume 2*. pp. 665-668 (2015), http://dl.acm.org/citation.cfm?id=2819009.2819132

[41] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. *30th IEEE International Conference On Software Maintenance And Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. pp. 391-400 (2014), https://doi.org/10.1109/ICSME.2014.61

[42] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. & Egyed, A. Enhancing clone-and-own with systematic reuse for developing software variants. (Gesellschaft für Informatik e.V.,2016), http://dl.gi.de/handle/20.500.12116/741

[43] Martinez, J., Ziadi, T., Bissyandé, T., Klein, J. & Traon, Y. Bottom-up technologies for reuse: automated extractive adoption of software product lines. *Proceedings Of The 39th International Conference On Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. pp. 67-70 (2017), https://doi.org/10.1109/ICSE-C.2017.15

[44] Martinez, J., Ziadi, T., Bissyandé, T., Klein, J. & Traon, Y. Bottom-up adoption of software product lines: a generic and extensible approach. *Proceedings Of The 19th International Conference On Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. pp. 101-110 (2015), https://doi.org/10.1145/2791060.2791086