Bridging the Gap between User Stories and Feature Models by Leveraging Version Control Systems: a Step towards Software Product Line Migration

Thomas Georges[a,b], Marianne Huchard[a], Mélanie König[a,b], Clémentine Nebut[a], Chouki Tibermacine[c]

[a] *LIRMM, Univ Montpellier, CNRS, Montpellier, France*
[b] *ITK -Predict & Decide, Montpellier, France*
[c] *IRISA, University of Southern Brittany, Vannes, France*

Abstract

Context: Throughout the software lifecycle, a significant amount of knowledge is accumulated around the source code. In our work, we focus on agile software requirements, particularly user stories, and on issues and merge requests in version control systems, that have been opened for implementing user stories. **Objectives**: The objective of this paper is to present a method that leverages this knowledge to guide an SPL migration. **Methods**: We consider merge requests in version control systems as the link between user stories (requirements) and the source code (implementation). The method combines Natural Language Processing (NLP) and clustering to identify features from user stories and hierarchically organize them. Relational Concept Analysis (RCA) is then used to compute logical rules from the hierarchy of features, using their links with the products and the source code. The logical rules are finally transformed into constraints in the produced feature model. **Results**: The method was implemented and evaluated on a dataset from an industrial partner. The results showed the efficiency of our method in synthesizing feature models for an SPL migration of the partner's code base. **Conclusion**: The proposed method synthesizes feature models to guide an SPL migration based on agile software development practices and demonstrates its effectiveness on a real industrial dataset.

Keywords: Software Product Line, Agile Process, User Story, Reengineering, SPL domain engineering, Feature Model, Natural Language Processing, Formal Concept Analysis

1. Introduction

Software Product Lines (SPL) [1] are families of related software systems that share a common core while differing in specific features. SPL provide a means to develop highly configurable systems, enabling the rapid creation of new software by adapting the common core to a specific set of features. The features, both common to all products or specific to a subset of them, are usually organized in a feature model [2], modeling which features are required in each product, which are optional, as well as constraints linking the features. While SPL are now a well-established way to efficiently produce highly-configurable software products, migrating an existing software family [3] into an integrated SPL platform is still challenging and organizations may be reluctant to adopt the approach. Hesitations often arise from the lack of standard procedures guiding the process, as well as doubts on the cost/benefit ratio [4]. Although companies own large bases of well-documented code, they still manually build tailored applications from the base code to meet their clients' requirements, sometimes with a clone-and-own strategy, *i.e.* duplicating an existing product and adapting it to the new specific features.

One of the difficulties during the migration process is to extract a relevant feature model from the existing products. This task consists in identifying all the features of the various software products [5], then identifying the common ones and the specific ones, and organizing them in a model specifying the constraints existing between the features (*e.g.* which feature implies which other feature, or which feature cannot appear in the same product as another feature). In this paper, we address this issue in the context of products built with an agile process, specifically where requirements are defined using user stories, and a version control system (VCS) [6] is used to store and trace project artifacts. A user story is a general explanation of a software feature, written in natural language from the perspective of the end user. A user story is usually written using a common template or format. In the following, we use the most common template: the Connextra template [7]. User stories are nowadays widely used in a large part of software projects to define requirements. VCS platforms like GitLab or Github are a widespread support, not only to manage code artifacts, but also as a central part of project management, directly using integrated tools to manage the user stories or integrating external tools in the VCS platform. Therefore, our approach is suitable for a large part of projects.

In this context, we have developed an automated feature model synthesis approach based on the knowledge collected from the user stories and the way they are linked to the product code through traceability links established in the VCS. We have implemented this automated feature model synthesis method using a combination of different techniques.

- Natural Language Processing (NLP) is used to analyze user stories and identify an initial set of concrete features.
- Vectorization and Clustering are used to consider the initial set of concrete features and identify abstract features. These features enable us to build preliminary hierarchies of features with refinement relationships.
- Formal Concept Analysis (FCA [8]) and Relational Concept Analysis (RCA [9]) are applied to the previous results and also take as input the links established by the VCS between the user stories and the code repository. These links connect user stories to files that implement them, going through links between issues, merge operations, changes and then files. On top of this knowledge, FCA and RCA infer interconnected groups of products, features, roles and code files.
- Finally, a logical analysis of the interconnected groups is performed to infer logical constraints between features and to refine the feature model so that it most closely reflects real-world products.

We evaluated this method on a dataset built from our industrial partner's code base and the documentation on their VCS. Our industrial partner provides decision-support software systems for farmer and agricultural advisors. The results showed that our feature model synthesis approach using different sources of knowledge provides relevant feature models on the industrial dataset. In addition, these feature models can easily be modified by an expert to produce the actual feature model of the product line.

The rest of the paper is organized as follows. Section 2 presents an overview of the approach, including the necessary artifacts. We then detail each step of our synthesis process in Sect. 3, illustrating it with a small example drawn from the dataset provided by our industrial partner. Section 4 describes the evaluation with the actual industrial dataset, followed by a discussion and an analysis of the obtained results. Related work is

presented in Sect. 5. We summarize the contributions and outline future perspectives of this work in Sect. 6.

2. Background and overview

In this section, we provide background on Software Product Lines (Sect. 2.1), then we give an overview of our approach and detail its inputs (Sect. 2.2).

2.1. Software Product Lines

Software Product Lines (SPL) are families of products sharing a common basis but also owning variation points. Usually, both common and variant characteristics of an SPL are called *features* and are organized in a *feature model*. An example of feature model is given in Fig. 1 for an SPL of web browsers. It has a main tree structure, with decorated edges. In the example, `WebBrowser` is the root of the feature model. The edges represent parent-child relations between features: to have a child present, the parent has to be present. `Navigation` is a mandatory feature under `WebBrowser`: it has to be present if `WebBrowser` is present. `TextToSpeech` and `VoiceControl` are optional, they may be present or not. `Tabbing` and `Spatial` form an or-group under `Navigation`: if navigation is present, there can be tabbing or spatial navigation, or both. `Classic` and `Advanced` form a xor-group under `TextToSpeech`: if `TextToSpeech` is present, it has to be `Classic` or `Advanced`, but not both. Xor-groups are also called alternative groups. Some constraints on features cannot be expressed with tree edges, they are called cross-tree constraints, and often presented in textual form, below the feature model. In the example, the presence of `Advanced` text-to-speech implies the presence of the `VoiceControl` feature, while the features `VoiceControl` and `Spatial` are mutually exclusive. A product corresponds to a *configuration* of features, which must be valid w.r.t the feature model. For example, the configuration {`WebBrowser`, `Navigation`, `Tabbing`, `VoiceControl`} is valid, but the configuration {`WebBrowser`, `Navigation`, `Tabbing`, `TextToSpeech`} is not valid, it is incomplete, as it violates the alternative group constraint. According to this constraint, when `TextToSpeech` is present, either `Classic`, or `Advanced` option has to be chosen. As another example, the configuration {`WebBrowser`, `Navigation`, `Spatial`, `VoiceControl`} is not valid because it violates the exclusion cross-tree constraint "$\neg(Spatial \wedge VoiceControl)$"

appearing under the feature model of Fig. 1 stating that `Spatial` cannot be present at the same time as `VoiceControl`.

Each feature may be abstract or concrete. Abstract features are used to structure and document the feature model, but are not directly bound to implementation artifacts. Concrete features are bound to implementation artifacts; one may note that leaves of a feature tree should be concrete features.

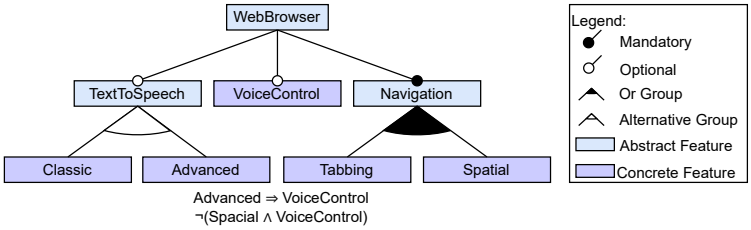

Figure 1: An example of a feature model for a WebBrowser product line

2.2. Overview of the proposed process

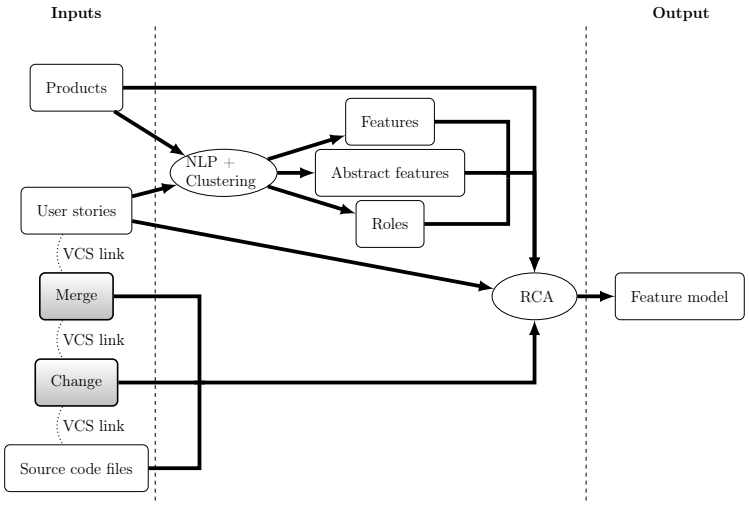

Figure 2: Process Overview: Automated process from products, user stories linked to the source code via VCS links (inputs) to the feature model (output) using NLP, clustering, and RCA. Rectangle boxes represent artifacts, while ellipses represent processes. Grey boxes indicate artifacts originating from the VCS.

Our process, outlined in Fig. 2, is an automated method for feature model synthesis from the sets of user stories of a product family. The process is enriched by the use of links between the user stories and the code repository managed by a VCS platform. First, we apply natural language processing (NLP) techniques to identify fine-grained features and the roles present in the user stories. Next, we use clustering techniques to group related features into candidate abstract features. At this stage, the result is a tree structure for the feature model: the root at the top level, abstract features on the second level, and concrete features as the leaves, each linked to an abstract feature. Finally, we apply RCA to infer logical constraints, enriching the feature model with information about mandatory versus optional features and with cross-tree constraints.

2.3. Inputs: User stories and code repository managed in a VCS

The inputs of our approach are the product user stories and the code repository managed by the VCS.

User stories [7] are a usual way to define requirements in agile projects. Each *user story* expresses, in natural language, a description of a software system's features from the user's point of view. A *user story* is written following a common template or format. In what follows, we employ the widely used Connextra template [7], so that each user story adheres to the following format: "As a [persona], I want to [action], so that [goal]".

- *"persona"* is a typical user: personas represent, as the roles, groups of individuals who can perform some actions.
- *"action"* denotes what the persona wants to perform within the software system. We treat this action as the feature conveyed by the user story.
- *"goal"* is the intended persona's objective; it explains why the persona performs the action.

Our industrial partner does not express the goal in the user stories, so in the rest of the paper the user stories follow the form: "As a [persona], I want to [action]". User stories are supposed to be available in the VCS, that guides us to identify the relations between the requirements and the code files, as summarized in Fig. 3. In this work we assume that user stories are directly stored in the VCS in the form of dedicated issues, that is a common

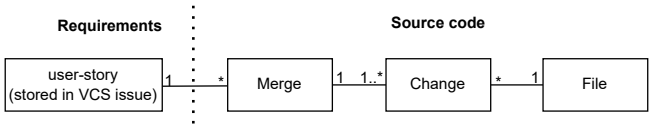

Figure 3: Focus on the links between user requirements (user stories) and code files through the VCS. A user story is implemented by at least one merge which is a set of at least one change applied to one file.

practice. However, they can be stored in an external tool, like Jira, as soon as they are linked to the operations on the files through the VCS.

The second element we use is the code repository managed by the VCS, which records the changes that occurred during the code production and provides traceability links between the different artifacts. More precisely, we use the information of merges, and the code files themselves. A *merge* is a classical operation in VCS platforms. It corresponds to a global modification/commit in a project. It contains a non empty list of changes (creation, edition or deletion) in the project's files. We assume that each merge refers to a user story. As soon as a user story is implemented, it is thus referred by at least one merge. The *code files* we deal with refer to all the created, edited or deleted files involved in the implementation of a user story.

2.4. Output: Feature model

The output of the process is a feature model representing the features of the products, along with their relationships and constraints. The root of the feature model represents the product family. Under the root, a hierarchy of features is organized using *optional* and *mandatory* relationships, as well as *or groups* and *alternatives*. Features may be concrete (the leaves of the feature tree are concrete) or abstract (abstract features are a way to organize the variability in the feature model). The output feature model also contains cross-tree constraints in the form of logical expressions involving features. We consider boolean features, as the features may be present or absent in the product, and we do not introduce cardinalities or other advanced notations in the feature model. To build the feature model, we use *FeatureIDE* [10] to transform the description of the feature model generated by our approach into a feature model diagram.

3. Feature model synthesis

In this section, we introduce our feature model synthesis method. It is illustrated with a small example drawn from our industrial partner's case study (Sect. 3.1). We detail the two main parts of the overview figure (Fig. 2): the NLP and clustering part, drafted in Fig. 4 (Sect. 3.2), and the RCA part, drafted in Figures 6 and 7, with a detailed summary in Fig. 15 (Sect. 3.3). Finally, Sect. 3.4 introduces the tools.

3.1. Illustrative example

For confidentiality reasons, we cannot share all the company's dataset and the obtained results. However, we present samples from these. The three products of the example are decision-support software systems for farmers and agricultural advisors, dedicated respectively to the culture of almonds, vines and orchards. The inputs are the user stories of the products, with the related artifacts from a GitLab instance including the code file references.

To collect user stories and the related VCS artifacts, we leverage the possibilities offered by the VCS, which conveys traceability between the source code and the user stories that we extract using the VCS API. The user

Table 1: Relation between user stories (rows) and products, i.e. software systems (columns) that they specify. `Almond`, `Vine` and `Orchard` are three decision-support software for crop management. A User story can be part of the specification of several software systems.

	Products		
User Stories	Almond	Vine	Orchard
As a farmer I can refresh the predicted weather		x	x
As a farmer I can CRUD plots	x	x	x
As a farmer with a plot I can edit the parameters of a plot (current season)		x	x
As a farmer with a plot I can sort my plots in the list			x
As a farmer with a plot I can filter my plots in the list			x
As a farmer with a plot I can export observation data for a plot		x	
As a farmer with a plot I know when my plots will be in danger		x	x
As a farmer irrigator I can manage my irrigations and recommendations in my favorite unit	x		
As a farmer irrigator I choose my preferred irrigation unit in my user-settings		x	
As a farmer irrigator I can view my irrigation recommendations in my favorite unit	x	x	
As an admin I can CRUD a farmer		x	x
As an admin I can relaunch all failed simulation	x	x	x

stories of the three products are shown in Table 1. They do not contain the "So that" part, as it is the practice of our industrial partner to omit it. Table 2 shows a shortcut version of data populating the model of Fig. 3. A user story, restricted to its feature part to simplify the table, is connected,

Table 2: Relation between features (rows) and names of the files of the code repository (columns) that implement them. A feature can be implemented in different files and several features can be implemented in the same file. This mapping describes the links between requirements and their implementation artifacts.

	Files											
Features	admin	weather	farmer	plotList	danger	irrigation	recommendation	export	observation	plot	user-settings	alert
can refresh the predicted weather		x	x									
can CRUD plots			x							x		
can edit the parameters of a plot current season			x							x		
can sort my plots in the list				x						x		
can filter my plots in the list				x						x		
can export observation data for a plot								x	x			
know when my plots will be in danger					x							x
can manage my irrigations and recommendations in my favorite unit						x	x					
choose my preferred irrigation unit in my user-settings						x					x	
can view my irrigation recommendations in my favorite unit						x	x					
can CRUD a farmer	x		x									
can relaunch all failed simulation	x	x										

through this table, directly to the files implementing it, i.e. bypassing, for the sake of the simplified illustration, the merge and change artifacts of the VCS.

3.2. Towards a feature tree: NLP for role extraction, and for fine-grained and coarse-grained feature extraction

Natural Language Processing (NLP) techniques have two purposes in our approach. The first is the identification of the roles and the features from the user stories. We consider in the following that an action corresponds to a small-grained feature. The second purpose is the identification of clusters of features to determine abstract (coarse-grained) features.

We first decompose each user story, separating the role and the feature (Step NLP 1 in Fig. 4). Due to the well-defined format of the user stories, we can parse them to identify the first nominal group, and the feature as the content that follows. This step results in features and links between features and roles. Table 3 shows the links between features and roles. In the user story *As a farmer, I can CRUD plots*, the role is *farmer* and the feature is *can CRUD plots*. The link between the feature and the role appears as a cross in the table.

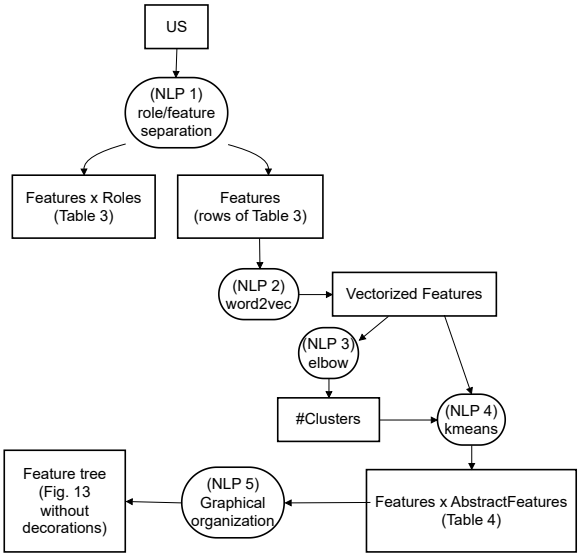

Figure 4: NLP and clustering steps: user stories are separated into role and feature parts (NLP 1); word2vec is applied to features (NLP 2); resulting vectorized features are analyzed with elbow to compute a cluster number (NLP 3); k-means is applied to obtain abstract features (NLP 4); the features (including abstract ones) are used to draw the skeleton of the feature model, i.e. the feature tree (NLP 5).

Then the f features are clustered into c clusters thanks to a pipeline of three methods. First we use a pre-trained *word2vec* [11] model for vectorization to represent our features as vectors (Step NLP 2 in Fig. 4). In our process the word2vec model was pre-trained on *word2vec-google-news-300* with 3 million words and sentences. The second step of our pipeline is to determine the optimal number of clusters by using the *elbow method* (Step NLP 3 in Fig. 4). The elbow method gives the optimal number of clusters by calculating the *within-cluster-sum of squared errors* (WSS) and picking the number of clusters when the WSS stops decreasing. It is an heuristic used to determine the optimal number c of clusters based on the evolution of the variance for each number of clusters. In the final step (Step NLP 4 in Fig. 4), once the features had been vectorized and the optimal number of clusters determined, we apply k-means clustering [12]. K-means method arranges our features in the number of clusters given by the elbow method. Each obtained cluster (group of features) is assumed to have consistent semantics, and is candidate to be an *abstract feature*.

Table 3: Relation between features (rows) and roles (columns) that have access to them. Some roles are specializations or extensions of others, e.g. `farmerwithplot` specializes `farmer`.

	Roles			
Features	farmer	farmerwithplot	farmerirrigator	administrator
can refresh the predicted weather	x			
can CRUD plots	x			
can edit the parameters of a plot current season	x	x		
can sort my plots in the list	x	x		
can filter my plots in the list	x	x		
can export observation data for a plot	x	x		
know when my plots will be in danger	x	x		
can manage my irrigations and recommendations in my favorite unit	x		x	
choose my preferred irrigation unit in my user-settings	x		x	
can view my irrigation recommendations in my favorite unit	x		x	
can CRUD a farmer				x
can relaunch all failed simulation				x

In order to name our clusters, we identify the most important words by removing the stop words and using the most frequent words remaining in the cluster. More precisely, after removing the stop-words from the clustered features of a user story, we keep at most the three most frequent words to name the involved cluster. These names will be renamed or refined at a later stage by an expert while reviewing the feature model. After the NLP step, we obtain the user stories, the roles, the features and the abstract features that populate the model of Fig. 5. Table 4 shows abstract features that can be obtained in our example after a run of the NLP process. Two of them group more than one feature and aim to generalize or describe the features of the group. For example *plot unit list* generalizes features related to plot list or to unit.

Table 4: Relation between Features (rows) and Abstract Features (columns).

	Abstract Features				
Features	refresh predicted weather	crud plots	plot edit parameters	plot unit list	crud farmer
can refresh the predicted weather	x				
can CRUD plots		x			
can edit the parameters of a plot current season			x		
can sort my plots in the list				x	
can filter my plots in the list				x	
can export observation data for a plot			x		
know when my plots will be in danger				x	
can manage my irrigations and recommendations in my favorite unit				x	
choose my preferred irrigation unit in my user-settings				x	
can view my irrigation recommendations in my favorite unit				x	
can CRUD a farmer					x
can relaunch all failed simulation			x		

The backbone of a feature model is obtained using a graphical representation of this table as a tree (Step NLP 5 in Fig. 4). Below the root node of this tree, we introduce sub-nodes associated with the abstract features. Then each feature appears as a node linked to the abstract feature to which it belongs. For our example, we obtain the tree structure of the feature model shown in Fig. 13.

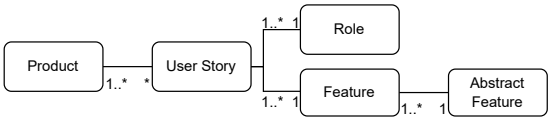

Figure 5: A Model of Products, User stories, Roles, Features and Abstract Features.

3.3. Towards a Feature Model: FCA/RCA to Derive Logical Constraints that Enhance the Feature Tree

The tree derived from the NLP step can be considered as an organization of features using simple ontological semantics. To obtain a feature model, we enhance this initial structure with logical constraints. The logical constraints are of several types. Some indicate which features are mandatory or optional (default choice), some highlight *or groups* or *alternatives*, lastly, others capture cross-tree constraints. Our method consists in exploiting the available information on products (ranging from user stories to VCS data) using Formal Concept Analysis (FCA, presented by Ganter et al. [8]) for its capability to structure and formalize this information in various forms, including logical formulas.

We proceed as summarized in Fig. 6. From the inputs (analyzed user stories linked to products, features and code repository managed by a VCS, see Figures 3 and 5), we apply an extension of FCA called Relational Concept Analysis (RCA [9]) to extract a base of implications representing the logical constraints. More specifically, we rely on the widely used Duquenne–Guigues base of implications [13].

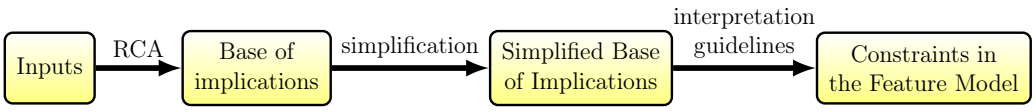

Figure 6: Extraction of FM constraints.

Figure 7 is a zoom on this first step, i.e. RCA application. The inputs are products, user stories, roles, features, abstract features, merges, changes and files, structured as mentioned in Figures 3 and 5. The handled elements (products, user stories, etc.) are linked through several relations, for example products are concerned by several user stories, and user stories are implemented through a succession of merges that concern changes in files. Our objective is to observe what are the shared links, so as to infer elements or groups of elements that are always present together, never present together, etc. From the inputs, we build two kinds of tables, called Formal Contexts (FC) and Relational Contexts (RC). Those tables are formally described in next sections, but simply said Formal Concepts describe entities (products, user stories, etc) with basic characteristics they have, while Relational Contexts describe entities with the links they entertain with other entities. The links follow the structure of the relations of Figures 3 and 5. RCA analyzes these tables and can produce a set of observed rules, in the form of a base of implications. These implications indicate that for example a given set of features is always seen with a given set of changes of files. The raw implications that are obtained are then simplified, and used to constrain the feature model.

The rest of this section is organized as follows. We first draw the main lines of logical constraints extraction with FCA in Sect. 3.3.1 and explain how they can enhance the feature model. As our inputs (Figures 3 and 5) follow an Entity-Relationship model, we use RCA to extract the base of implications representing the logical rules, as detailed in Sect. 3.3.2. The raw implications obtained using RCA are then simplified through an automated process described in Sect. 3.3.3, to obtain more understandable and actionable implications. Then guidelines to interpret the RCA implications into constraints in the feature model are applied, as presented in Sect. 3.3.4. We illustrate these guidelines using the whole example to obtain a candidate feature model, intended to be refined afterwards by a human expert. Table 5 describes the added value of each technique and conceptual structure we use.

3.3.1. Logical Constraints extraction with FCA

As reported by Galasso et al. [14], there is a continued track of research that leverages FCA for representing and extracting logical constraints in the field of SPLE. The simplest form of data on which this extraction can be made is a set of product configurations, i.e. products and their respective

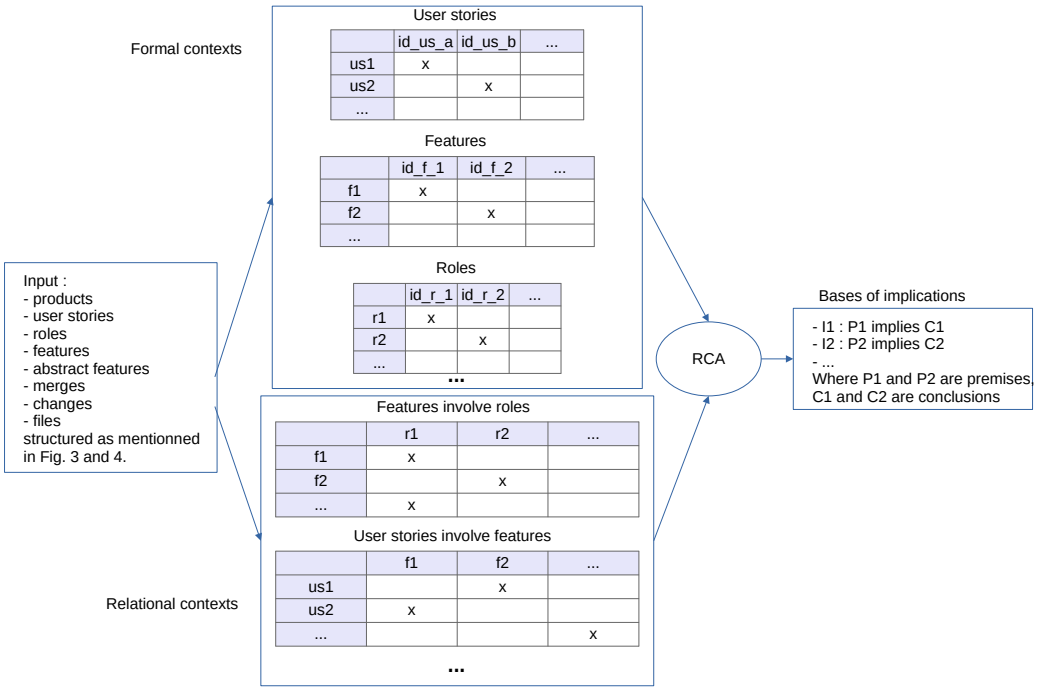

Figure 7: Obtaining a base of implications using RCA.

feature lists. Table 6 shows a reduction (with shorten names) of Table 1 to the products and user story features, considering that there is a one-to-one mapping between the user stories and their features. Although this is not the general case (see Fig. 5), it does hold in our practical case study. Using FCA vocabulary, such a table is called a *Formal Context*, with *objects* (the software products) described by *attributes* (the features). FCA provides two main outcomes for data analysis: concept lattices and propositional logical expressions (e.g. implications).

Table 5: The added value of each of the techniques and conceptual structures.

Technique	**Added value**
FCA and RCA	Extract structured logical relations from our user stories, features, and code.
Base of Implications	Produces minimal, non-redundant logical dependencies.
Simplification	Reduces redundancy in the implications and improves readability of the implication base.
Feature Model Constraints	Maps simplified implications into constraints: mandatory/optional features, or/alternative groups, and cross-tree constraints.

Concept lattices (and other derived structures) classify objects according to their shared attributes in concepts organized in a hierarchy, offering a graphical view. The concept lattice associated with the formal context of Table 6 is shown in Fig. 8. Each concept is an object group (extent) associated with the group of their shared attributes (intent). In addition, in a concept, none of the associated groups can be extended: if an object is added, an attribute is lost (it cannot be shared by the added object); symmetrically, if an attribute is added, an object is lost (it cannot have the added attribute). Concepts are organized in a specialization hierarchy based on group inclusion. As attributes are top-down inherited and objects are bottom-up inherited, a simplified form of concepts is used in the representations. In Fig. 8, `Concept_product_13` thus groups the bottom-up inherited objects `Vine` and `Almond` that share the top-down inherited attributes `CRUDplots`, `relaunch`, and the introduced attribute `view`.

A logical view can be extracted from the concept lattice, or from the formal context. For example, Galasso et al. [15] analyzed the concept lattice to derive specific logical constraints that usually come with a feature model such as co-occurrences, binary implications, mutual exclusions or candidates *or groups* or *alternatives*. This approach may generate numerous constraints, and among them accidental ones have to be discarded. Another approach is to leverage bases of implications, that has always been an active track of FCA domain [16, 17]. For an attribute set A, an implication is a pair of two attribute sets (P, C), with premise $P \subseteq A$, and conclusion $C \subseteq A$. It is denoted by $P \Rightarrow C$. When P is empty, this is denoted as $\Rightarrow C$. (P, C) is a valid implication in a formal context if any object that owns attributes of P also owns attributes of C. A base of implications is a set of valid, non-redundant implications from which any other valid implication can be derived using inference rules. Each implication has an *absolute support level* (or *support* for short), which corresponds to the number of products holding the rule (having the attributes of the premise).

Table 6: Formal context Products × Features, based on the running example. The feature name is reduced to the first word, except when a disambiguation is needed (e.g. CRUD).

product	refresh	CRUDplots	edit	sort	filter	export	know	manage	choose	view	CRUDfarmer	relaunch
Vine	x	x	x			x	x		x	x	x	x
Almond		x						x		x		x
Orchard	x	x	x	x	x		x				x	x

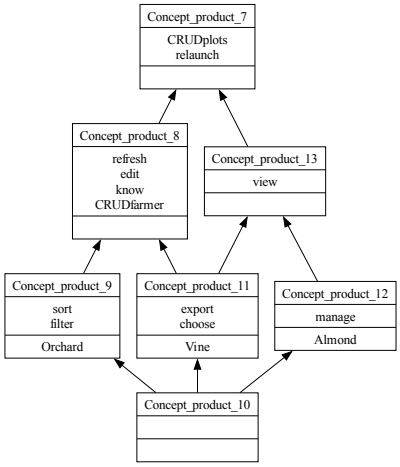

Figure 8: Concept lattice of the formal context of Table 6.

(I1) <3> ⇒ CRUDplots, relaunch
(I2) <2> CRUDplots, CRUDfarmer, relaunch ⇒ refresh, edit, know
(I3) <2> CRUDplots, know, relaunch ⇒ refresh, edit, CRUDfarmer
(I4) <2> CRUDplots, edit, relaunch ⇒ refresh, know, CRUDfarmer
(I5) <2> refresh, CRUDplots, relaunch ⇒ edit, know, CRUDfarmer
(I6) <1> CRUDplots, choose, relaunch ⇒ refresh, edit, export, know, view, CRUDfarmer
(I7) <1> CRUDplots, manage, relaunch ⇒ view
(I8) <1> CRUDplots, export, relaunch ⇒ refresh, edit, know, choose, view, CRUDfarmer
(I9) <1> CRUDplots, filter, relaunch ⇒ refresh, edit, sort, know, CRUDfarmer
(I10) <1> CRUDplots, sort, relaunch ⇒ refresh, edit, filter, know, CRUDfarmer
(I11) <1> refresh, CRUDplots, edit, know, view, CRUDfarmer, relaunch ⇒ export, choose
(I12) <0> refresh, CRUDplots, edit, export, know, manage, choose, view, CRUDfarmer, relaunch ⇒ sort, filter
(I13) <0> refresh, CRUDplots, edit, sort, filter, export, know, choose, view, CRUDfarmer, relaunch ⇒ manage

Table 7: DBGI of Table 6. Implications are ordered by decreasing absolute support (indicated between < >).

There exists a range of implication bases with various properties [16, 17]. In this work, we choose the Duquenne-Guigues base of implications (DGBI) for its concision, as the DGBI is a minimal cardinality base [13]. For Table 6, the DGBI contains 13 implications with supports equal to 0, 1, 2 and 3 respectively. For example the following implication holds for the two products `Vine` and `Orchard`, hence has absolute support 2:

$$CRUDplots, CRUDfarmer, relaunch \Rightarrow refresh, edit, know \quad [Imp]$$

In the DGBI, the implication set is non-redundant, yet individual implications may still contain redundant information, leaving room for simplification. For example another implication is (with support 3, thus held by the 3 products):

$$\Rightarrow CRUDplots, relaunch$$

Using this information, implication $[Imp]$ can be simplified as:

$$CRUDfarmer \Rightarrow refresh, edit, know$$

One can note that all rules are valid, even if the support is not maximal. The rules have a support that corresponds to the number of objects to which they apply. There may be objects to which they do not apply (the support is then not maximal) but they remain true. Indeed, the rules are implications and when the premise is false, the evaluation of the implication is true.

The whole set of implications for Table 6, without simplification, thus as they are produced by the original DGBI definition and the implementing tools, is shown in Table 7. Note that the implications are usually computed directly from the formal context.

We give below pieces of information that can be extracted from the implications, and how they could enhance the feature model. Some of them can directly be deduced from a single implication, others can be deduced from groups of implications.

(I1) indicates two mandatory features (CRUDplots, relaunch). (I7) highlights the fact that manage implies view. (I11) indicates that having refresh and view implies having export. (I2) to (I5) show the co-occurrence of CRUDfarmer, know, edit, and refresh, as they are equivalent. Indeed, after removing the two mandatory features CRUDplots and relaunch, each feature among CRUDfarmer, know, edit and refresh implies the three other features. (I6) and (I8) show the co-occurrence of export and choose, for similar reasons. (I9) and (I10) show the co-occurrence of sort and filter. From (I12) and (I13), whose support is 0, we can deduce mutual exclusions: we do not have at the same time export and manage (I12), we do not have at the same time export and sort (I13).

There is a strong connection between the implication base and the set of concept intents [18]. A procedure to obtain the concept intents is as follows. For every feature set S_0, apply every implication $P \Rightarrow C$ of the base whose premise P is included in S_0, giving a new feature set $S_1 = S_0 \cup C$. Then iterate the step on S_1, giving S_2, and so on, until $S_i = S_{i-1}$ (no new feature is discovered at a step).

For example, if we take set $\{view\}$, by (I1), we obtain $\{view, \textbf{CRUDplots}, \textbf{relaunch}\}$ (intent of Concept_product_13) and no other implication applies.

If we take $\{manage\}$, by (I1), we obtain $\{manage, \textbf{CRUDplots}, \textbf{relaunch}\}$. I7 can now be applied, giving $\{manage, CRUDplots, relaunch, \textbf{view}\}$ (intent of Concept_product_12) and no other implication applies.

If we take $\{sort, export\}$, by (I1), we get $\{sort, export, \textbf{CRUDplots}, \textbf{relaunch}\}$; by (I8) we get $\{sort, export, CRUDplots, relaunch, \textbf{refresh}, \textbf{edit}, \textbf{know}, \textbf{choose}, \textbf{view}, \textbf{CRUDfarmer}\}$; by (I10) we get $\{sort, export, CRUDplots, relaunch, refresh, edit, know, choose, view, CRUDfarmer, \textbf{filter}\}$; then by (I13) we complete by manage, and we get the whole feature set $\{sort, export, CRUDplots, relaunch, refresh, edit, know, choose, view, CRUDfarmer, filter, \textbf{manage}\}$ (intent of bottom Concept_product_10).

This shows the role of implications with support 0, when they exist, which is to support completion of a feature set to obtain the bottom concept intent, which rarely corresponds to a valid configuration.

It can be observed that this procedure enumerates configurations corresponding to three situations: (1) a valid configuration (when the corresponding extent introduces a product, e.g. intent of Concept_product_12 which introduces `Almond`) —this means that the product has exactly the whole feature set of the concept, i.e. inherited and introduced features; (2) an incomplete configuration (when the corresponding extent does not introduce a product, e.g. intent of Concept_product_13, which groups `Vine` and `Almond` introduced in subconcepts); (3) an invalid configuration when the concept intent is the one of the bottom concept and the extent of the bottom concept is empty. The last case happens when there is no product having all features, which is a current case. Only one invalid configuration (the whole feature set) can be produced of this type, it is thus easy to discard.

A theoretical result is that it is equivalent to dispose of the formal context or of the concept lattice (one can be computed from the other); from the implication base it is possible to know the structure of the concept lattice; but it is not possible to infer if an intent is the attribute set of an object (the extent part), or an intersection of several attribute sets, or the bottom [8].

The foregoing explanation is important to know in order to avoid overinterpreting what the implication base means, and to appropriately use it. The base and the supports have to be mainly used as a set of observed (or not observed when support is 0) valid dependencies between features. The dependencies can serve to exhaustively enumerate all valid configurations, and find incomplete ones also (with the particular case of the often invalid bottom configuration). But the set of not observed dependencies is not exhaustive, only those that serve to obtain all configurations are present in the base. Thus all mutual exclusions between features cannot be extracted, contrarily to the work made by Carbonnel et al. [15]. With the DGBI, we gain concision at the cost of missing explicit expression of some mutual exclusions by this approach. We hypothesize in our work that many mutual exclusions are accidental, due to the low number of analyzed products.

3.3.2. Logical Constraints extraction with RCA

In its simplest setting, FCA allows to extract variability in a set of products described by features. In more complex settings, other artifacts and more complex descriptions may participate to induce variability constraints.

This is the case in our dataset, with the artifacts presented in models of Figures 3 and 5. Before going into the complexity of our whole model, we explain how Relational Concept Analysis (RCA, [9]) can be applied to extract concept lattices and implications in a simplified model composed of products, features, and abstract features that group the features. This simplified model is shown in Fig. 9. Features, which were native attributes of the product with FCA, are now promoted as objects with their own description: in our case, features are described by the abstract features they belong to.

Figure 9: Simplified model to illustrate the application of RCA to relational models. The model is composed of products, features, and abstract features.

The input dataset for RCA, also called relational context family (RCF), is close to an entity-relationship model. It is composed of two kinds of contexts (i.e. tables): formal contexts and relational contexts, as illustrated in Table 8 that shows the RCF associated with the model of Fig. 9. A formal context is a set of objects, a set of native (also called primitive) attributes, and a relation which describes whether an object has an attribute. For example, a formal context `Product` associates a product (object) with its identifier (attribute); another formal context `Feature` associates a feature (object) with its identifier (attribute). A relational context is a relation between objects from the same formal context or between objects of two formal contexts. For example a relational context `product2feature` connects a product (object of the formal context `Product`) to its features (objects of the formal context `Feature`). The RCF of Table 8 comprises three formal contexts: `Product`, `Feature`, and `Abstractfeature`. The native attributes of these three formal contexts are here simple identifiers, and the formal contexts are thus diagonal tables. In rows, we have the objects, and in columns, we have identifiers of objects. In this particular case, objects and their identifiers are denoted by the same text in the table, but they refer to elements of different natures. This same Table 8 also shows the two relational contexts of the RCF: `product2feature` connects products to their features, while `feature2abstractfeature` connects features to the abstract features that group them into clusters.

With RCA, concept lattices are built through an iterative process. At the initial step, one concept lattice is built for each formal context, thus based only on the primitive attributes. In the following steps, each formal

Table 8: Relational Context Family PFA based on the running example reduced to products (P), their features (F), and the abstract features (A) that group the features. The feature name is reduced to the first word, except when a disambiguation is needed (e.g. CRUD). The abstract feature name is reduced to the initials of its terms. When objects are described by their identifiers, this gives diagonal contexts (e.g. the object `refresh` identifier is `refresh`). Tables `Product`, `Feature` and `AbstractFeature` are the formal contexts. Tables `feature2abstractFeature` and `product2feature` are the relational contexts.

Product	Vine	Almond	Orchard
Vine	x		
Almond		x	
Orchard			x

Feature	refresh	CRUDplots	edit	sort	filter	export	know	manage	choose	view	CRUDfarmer	relaunch
refresh	x											
CRUDplots		x										
edit			x									
sort				x								
filter					x							
export						x						
know							x					
manage								x				
choose									x			
view										x		
CRUDfarmer											x	
relaunch												x

Abstractfeature	rpw	cp	pep	pul	cf
rpw	x				
cp		x			
pep			x		
pul				x	
cf					x

feature2abstractFeature	rpw	cp	pep	pul	cf
refresh	x				
CRUDplots		x			
edit			x		
sort				x	
filter				x	
export			x		
know				x	
manage				x	
choose				x	
view				x	
CRUDfarmer					x
relaunch			x		

product2feature	refresh	CRUDplots	edit	sort	filter	export	know	manage	choose	view	CRUDfarmer	relaunch
Vine	x	x	x			x	x		x	x	x	x
Almond		x						x		x		x
Orchard	x	x	x	x	x		x				x	x

context, that is the source of a relational context, is extended with relational attributes. A relational attribute is an expression composed of a scaling operator (e.g. $\exists$), the name of a relational context (e.g. product2feature or feature2abstractfeature), and a concept from the target formal context (e.g. a concept built on top of the formal concept Product or on top of the formal context AbstractFeature respectively). For example, let us consider an excerpt of the concept lattices built by RCA and presented in Fig. 10. In Fig. 10 the intent and extent of several concepts seem to be identical, but in the extents we find objects, and in the intents we find identifiers of objects (this occurs when the context is diagonal, as already explained).

At step 0, abstract feature concepts CAF_31 and CAF_32 respectively group and represent single abstract features cp and pep. At step 1, the rela-

tional attribute ∃feature2abstractfeature (CAF_32) is generated and it is assigned to features edit, export, and relaunch as they are all connected to pep by at least one feature2abstractfeature link. The existence of this relational attribute induces the creation of CF_37, a feature concept which groups edit, export, and relaunch. At step 2, the relational attribute ∃product2feature (CF_37) is generated and assigned to all products, as all of them own at least one feature among those of CF_37 extent. Other operators can be used, such as ∀ or ⊇, and variants with percentages, but in this work, we only used ∃. The creation of these relational attributes is made systematically and they are used to extend the initial formal contexts. The extended formal contexts are then used to compute implication bases, as in the FCA case. One implication base is computed for each extended formal context and they can serve various purposes, as explained later.

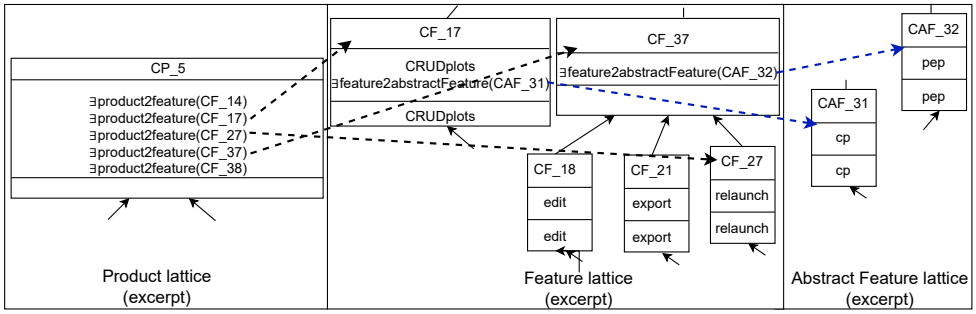

Figure 10: Excerpt of the concept lattices for products, features and abstract features produced by RCA. Dotted arrows (manually added on the figures produced by FCA4J) highlight the connections which are done between the concepts through the relational attributes.

3.3.3. Simplification of rules in the rule base

The obtained rules in the implication base are quite difficult to read, and the raw rules may be simplified, in four different and complementary ways.

- First, the relational attributes present in the premise may be very long and difficult to interpret. In some cases, they can be simplified, as explained in the following. This simplification is made automatically.
- Second, the relational paths can be simplified for a better readability. This simplification is also explained in the following, and is also made automatically.

- Third, in certain cases, cardinalities in the treated data may lead to simplifications. When a user story has a 1:1 relationship with a feature, we simplify it automatically by using the feature directly. This simplification improves readability by reducing the complexity of the relational attributes. In general, a feature can be in several user stories (as shown in Fig. 5) but in the dataset of our industrial partner, a feature is in a single user story. We have thus a 1:1 cardinality between feature and user story, and we have removed the user story entity from the model and the input of our algorithms (as seen in Fig. 9), without loss of information. This simplification is made automatically and can be added or removed as necessary.

- Fourth, there may exist redundancies in the basis that may be cleaned, as explained in Sect. 2. This cleaning could be automated with a logical approach, however, from an ontological point of view, discussions with several domain experts led us to let the expert remove the redundancies, as redundancies are sometimes useful from an ontological point of view, to ease the comprehension of concepts.

To alleviate and simplify the reading of the relational attributes and of the implications, we use a simplification adapted from [19]. In this simplification, the target concept C of a relational attribute is replaced by the attributes of the intent of C restricted to the introduced attributes. These introduced attributes are taken at the time of the concept creation. For example, ∃feature2abstractfeature(CAF_32) is rewritten into ∃feature2abstractfeature(pep), as CF_32 has been created to introduce pep. This is done recursively, and ∃product2feature(CF_37) is rewritten ∃product2feature(∃feature2abstractfeature(CAF_32)) and then ∃product2feature(∃feature2abstractfeature(pep)). When the intent contains more than one introduced attribute, they are connected with symbol &. When the intent contains no introduced attribute, it is replaced by the attributes of the intent of C, that are thus all inherited. The result is shown in Fig. 11, where the relational attributes of Fig. 10 have been rewritten using this mechanism.

Table 9 shows the product formal concept of Table 8 extended with the relational attributes rewritten with the simplification. ∃p2f(...) is a shorten version of ∃p2f(refresh &CRUDplots &edit &sort &filter &export &know &manage &choose &view &CRUDfarmer &relaunch).

To illustrate which new information can be obtained with this setting, we discuss the implication of support 3 which is included in the DGBI and is as

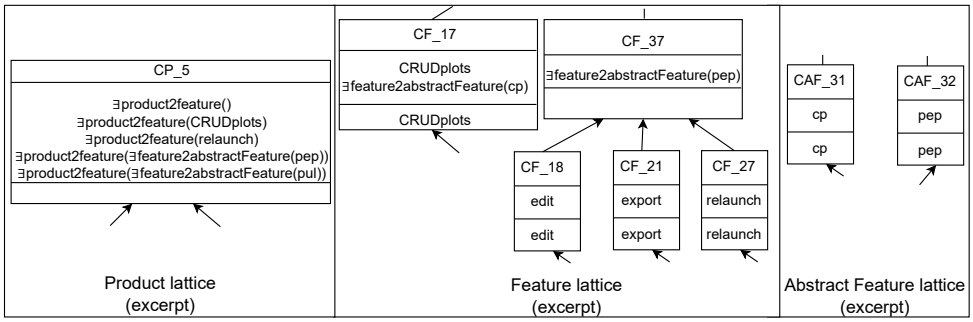

Figure 11: Excerpt of the concept lattices for products, features and abstract features produced by RCA after simplification of the relational attributes.

Table 9: Extended formal context for products during RCA running. The product context of Table 8 has been extended with the relational attributes. product2feature has been shorten into p2f. feature2abstractfeature has been shorten into f2a.

product	Vine	Almond	Orchard	∃p2f()	∃p2f(refresh)	∃p2f(...)	∃p2f(CRUDplots)	∃p2f(edit)	∃p2f(sort)	∃p2f(filter)	∃p2f(export)
Vine	x			x	x		x	x			x
Almond		x		x			x				
Orchard			x	x	x		x	x	x	x	

product	∃p2f(know)	∃p2f(manage)	∃p2f(choose)	∃p2f(view)	∃p2f(CRUDfarmer)	∃p2f(relaunch)	∃p2f(∃f2a(pep))	∃p2f(∃f2a(pul))
Vine	x		x	x	x	x	x	x
Almond		x		x		x	x	x
Orchard	x				x	x	x	x

follows:

(I1R) <3> => ∃product2feature(), ∃product2feature(CRUDplots), ∃product2feature(relaunch), ∃product2feature(∃feature2abstractFeature(pep)), ∃product2feature(∃feature2abstractFeature(pul))

Compared to the initial form of this implication (I1), in the FCA setting, the relational attributes indicate in addition that all products have:

- at least a feature (by ∃product2feature())
- at least a subfeature of pep (by ∃product2feature(∃feature2abstract-Feature(pep)))
- at least a subfeature of pul (by ∃product2feature(∃feature2abstract-Feature(pul)))

As a consequence, pep and pul are mandatory. In addition, in the feature model, they are candidate to be parent of a group.

To even improve readability of the implications, the relational path, i.e. the succession of <scaling operator><relation> is removed in the relational attributes. Fo example ∃product2feature(∃feature2abstractFeature(pul))) is simplified to only keep *pul*. This path was needed for the construction, but in the absence of name ambiguity, it can be deduced. The implication (I1R) then becomes: (I1RS) <3> => ⊤, CRUDplots,relaunch,pep,pul. ⊤ is the symbol that represents the top concept of any lattice.

3.3.4. Guidelines to enhance a feature model applied to our example

In this section, we describe the whole relational context family that we build from the dataset. Then we draw guidelines to interpret different types of implications extracted thanks to RCA, and how they are used to enhance the feature model. For rules with a maximal support and rules with null support, the interpretation of rules in terms of constraints in the feature model is automated. For rules of intermediate support, interpretation is left to an expert, using the guidelines.

The whole dataset. In the whole setting, following the models of Figures 3 and 5, we build one formal context for each element of interest: products, user stories, roles, features, abstract features, merges, changes and filenames. In these contexts, each object has a single attribute corresponding to the name of the element (we thus have diagonal formal contexts). We also build a relational context for each relationship: Products2UserStories, UserStories2Roles, UserStories2Roles, Features2AbstractFeatures, UserStories2Merges, Merges2Changes and Changes2Filenames. For example, we build the relational context between features and the abstract features, shown in Table 4. We see in this table that the abstract feature *plot edit parameters* is associated with its (concrete) associated features, e.g. *can edit the parameters of a plot current season* and *can export observation data for a plot*.

Interpretation of the various sorts of implications. To enhance the feature model, we exploit generated implications from the product context. We obtain a list of implications representing the relations between objects among the products, the user stories, the roles, the features, the abstract features, the merges, the changes and the filenames. We interpret chosen implications with specific shape (and their support), to extract the corresponding constraints for our feature model: constraints that correspond to the tree structure that can reveal mandatory features, and cross-tree constraints. In

the example of Fig. 12, the constraint $AF_{A2} \Rightarrow F_2$ is a cross-tree constraint which implies that all the configurations containing AF_{A2} must contain F_2. The constraint $F_1 \Rightarrow AF_{A1}$ is a refinement constraint, represented by a tree edge, which means that when the feature F_1 is contained in a configuration, its abstract feature AF_{A1} is also contained into it (this tree edge was already present after the NLP step). Finally, the constraint $AF_{A2} \Rightarrow F_3$ is an additional constraint implying that if the abstract feature AF_{A2} is part of a configuration, then the feature F_3 is also part of it, indicating that F_3 is a mandatory feature. The constraint $F_3 \Leftrightarrow F_4$ describes the relation between features that are always together in the same configurations (i.e. co-occurrent).

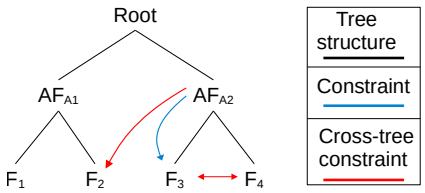

Figure 12: Feature tree with constraints.

The implications involving features and abstract features are interpreted as follows:

- If $n = 0$: No product has all the features in the premise. From the rule: $A \wedge B \implies C$ with a 0 support, we create the cross-tree constraint $\neg (A \wedge B)$.

- if n is the number of products, the premise is empty and the features in the conclusion are mandatory. In this case, when a feature under an abstract feature AF is mandatory then by construction of the abstract features, AF is mandatory.

- if $n > 0$: we deduce that the premise implies the conclusion. This implication can be translated into a cross-tree constraint. Some parts of it can also be turned into a mandatory annotation and removed from the cross-tree constraint. For example, if we have the constraint $AF1 \implies F2 \wedge F3$ with AF1 an abstract feature, F2 and F3 two features, and with F3 a child of AF1 in the tree structure, then F3 is a mandatory feature child of AF1, and the cross-tree constraint is $AF1 \implies F2$. When the premise is a conjunction, the underlying

constraint is added to the feature model in the form of a cross-tree constraint.

In what follows, we apply simplifications consisting in keeping only one feature among co-occurring ones, removing implied ones, removing attributes of the bottom concept and erasing relations.

From the premise of one of the simplified implications with support 0: `can sort my plots in the list, can export observation data for a plot, can manage my irrigations and recommendations in my favorite unit`, we can deduce the mutual exclusion constraint showed below the feature model of Fig. 13.

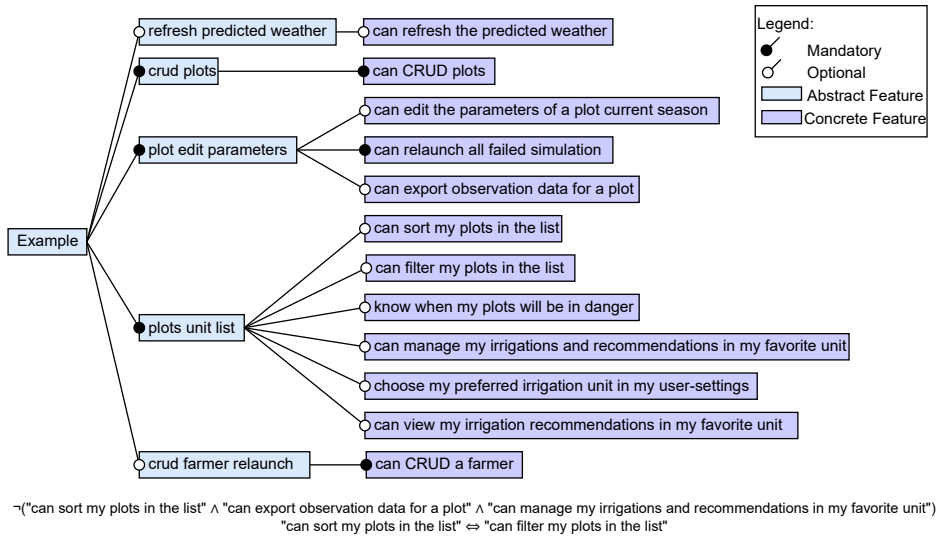

Figure 13: Feature Model Example.

From the simplified implication I1RS with support 3 (empty premise) " $=> \top$, CRUDplots, relaunch, pep, pul", we deduce that these features are mandatory.

We also interpret implications linking the source code (filenames) to the features. Consider the simplified implication: "administrator, farmer, mrs:1-2, plot unit list, plot edit parameters, src/plotList, src/plot, src/weather, src/farmer, edited_file, new_file ⇒ As a farmer with a plot I can sort my plots in the list, As a farmer with a plot I can filter my plots in the list, (...)". We can deduce, that, together with some conditions, actions on several files

(plotList, plot, weather, farmer), both features `sort` and `filter` are present, thus consolidating the fact that they are co-occurring, and that this is not accidental (due to the low number of products we examine).

Lastly, some information about roles can be extracted from some implications. For example with the following simplified implication, we can deduce that under some conditions, when refresh is present (and some merge are made that we omit for simplicity), then the role `farmerwithplot` is present: "<2> As a farmer I can refresh the predicted weather, mrs:1-2, plot unit list, plot edit parameters, src/plot, src/weather, src/farmer, edited_file, new_file $\Rightarrow$ farmerwithplot". This supports the idea that *refresh*, despite the fact that this is not mentioned in the user story, may be connected to role farmerwithplot, and propose it as an annotation of the feature model is suitable.

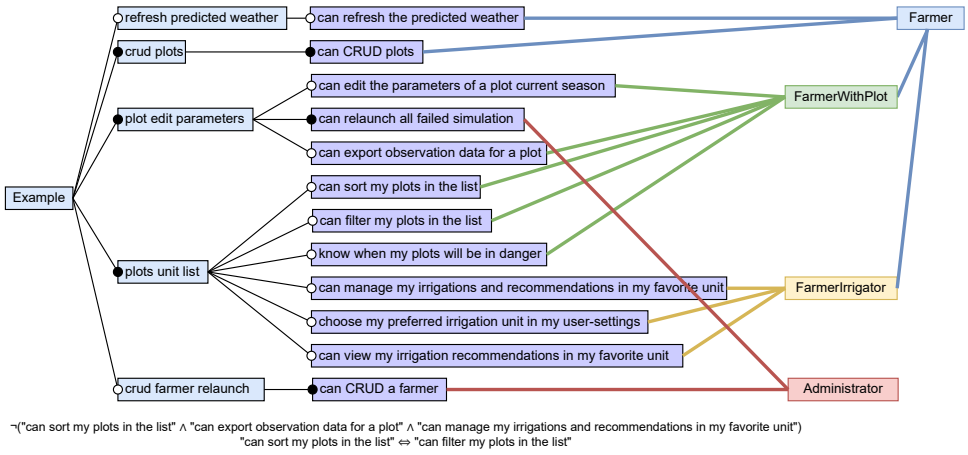

Figure 14: Enhanced Feature Model (with additional colored roles and links).

3.3.5. Summary of the application of RCA to enhance the feature tree and obtain the feature model annotated with roles

The RCA part of the process is summarized in Fig.15. Step RCA 1 takes as input various information from the code base to build the Relational Context Family (RCF). Step RCA 2 is simply the RCA process, which outcomes are extended formal contexts and concept lattices (Sect. 3.3.2). The extended formal contexts contain relational attributes that refer to the concepts in the lattices. At Step RCA 3, the extended formal contexts are used to compute a

base of implications (also Sect. 3.3.2). The implications are simplified (Step RCA 4) to obtain readable and actionable forms (Sect. 3.3.3). They are used to complete the feature tree to obtain a feature model, with various graphical or textual constraints (Step RCA 5) as described in Sect. 3.3.4. Finally, the connection between roles and features may be used to produce a feature model annotated by roles (Step RCA 6), as also described in Sect. 3.3.4.

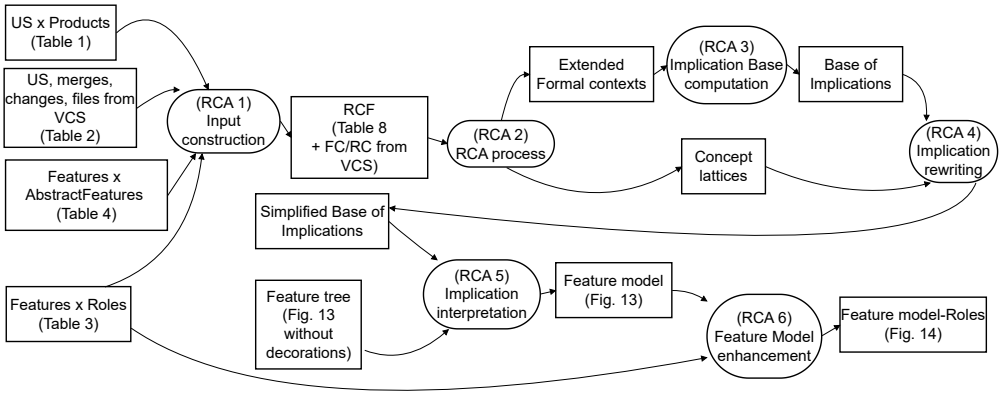

Figure 15: Summary of the RCA part.

3.4. Tool

We have automated the proposed process using several tools. To obtain our process inputs, we first use the *GitLab API*[1] to retrieve user stories and source code. For natural language processing, we use *Gensim* [20] with *word2vec* [11] and *k-means* [12]. Relational concept analysis and the Duquenne-Guigues base of implications were performed using *FCA4J*[2] [21] and the proper premises base algorithm described in [22]. FCA4J was used as a Java JAR with multiple options and comes with online documentation. In our dataset with multiple formal and relational contexts, the execution time was about one minute to perform all the computations. Our final models are represented in *XML* format and are loaded in *FeatureIDE* [10].

The artifact [23] used in this article is available online here:
https://gite.lirmm.fr/tgeorges/Bridging-the-Gap-between-User-Stories-

[1]https://docs.gitlab.com/ee/api/rest/index.html
[2]https://www.lirmm.fr/fca4j/

and-Feature-Models-by-Leveraging-Version-Control-Platform

4. Case study

In order to evaluate the proposed process, we conducted a case study with our industrial partner. We defined a protocol to compare feature models synthesized with our process with those manually created by our industrial partner. Currently, our industrial partner did not fully adopt SPLE. Thereby we do not have concrete feature models manually defined by the IT team. We organized a workshop where an expert from the IT team was asked to build his own feature model starting from a set of features. Then for the comparison, we confronted the two feature models by interacting with this expert. The closest the synthesized feature model is from the expert's one the more it is accurate. When a feature model is accurate it becomes useful for the IT team.

For the case study we used the user stories and the code repository from the GitLab instance of our industrial partner. The dataset includes 127 user stories, 6 products, 60 merges and a thousand of files. We identified 15 roles and arranged 127 features into 42 clusters. For the validation with the expert, we took a sample with 60 features in 10 clusters, in order to make the validation human-feasible. To introduce the workshop, a first feature model is created for a smaller example with only 12 features.

The expert is a software engineer who actively participates in the development of the products. He has 10 years of experience, 8 of which in the company. An introduction to the feature models, how to build them and what the rules are was given before the experiment started and we spent the necessary time answering questions and showing examples.

4.1. Protocol

We followed the procedure described below.

Step (1), we asked the expert to construct *his* feature model with a set of features; *Step (2)*, we presented to the expert *our* feature model, and we asked him to rearrange it if necessary; *Step (3)*, we asked the expert to rearrange *his* feature model, if necessary.

In our validation, we avoided any bias introduced by our presence in the workshop, by first letting our interviewee building the feature model without help or hints. First we give the expert the features and abstract features, and we asked him to produce a feature model; Then we give to the expert

our feature model and we asked him to rearrange it; Finally we give back to the expert the feature model built at the first time and we asked if he wants to edit it.

Based on the feature model created by the expert, there were three possible outcomes depending on the similarity between his results and ours. *Case 1:* the expert's feature model presents only clusters *similar* to those of our feature model, this may mean that our feature model is relevant to the company. The relevance depends on the number of similarities. *Case 2:* the expert's feature model has clusters that are similar to those in our feature model and it combines features from different clusters, so our feature model partially represents the system. *Case 3:* the expert's feature model has no clusters that are similar to those in our feature model. The expert reorganizes features from different clusters.

At each step, we engaged in a discussion about the process, during which we asked the expert to assess their level of confidence in the feature model they had developed, using a rating scale of 1 to 5. A score of 1 indicated little confidence in the model, while a score of 5 meant complete confidence. This discussion also helped to determine the similarity of clusters between those of the expert and ours. We also timed the duration required for each step. The first step, which involved sorting and categorizing features manually, proved to be the most time-consuming. On the other hand, fixing the automated model and then the manual model took less time. This difference was due to the better understanding of the features gained during the first step, as well as the preliminary reflection carried out. We asked the expert to evaluate the difficulty of each step on a scale of 1 to 5. The expert indicated that the first step was the most complex, while the subsequent steps were simpler. In order to compare the results of the expert with ours, we keep the count of the number of clusters. The measurements are presented in Table 10 for the example with 12 features and in Table 11 for the real case with 60 features.

Table 10: Metrics of the workshop on the example with 12 features

Example	Time in minutes	Difficulty (1-5)	#Clusters	Confidence (1-5)
Manual	10	3	6	3
Automated	5	2	5	3
Manual	5	2	6	4

Finally, we recorded the expert's remarks and comments for each step. The expert emphasized that the first step was the most demanding in terms

Table 11: Metrics of the workshop on the real case with 60 features

Actual	Time in minutes	Difficulty (1-5)	#Clusters	Confidence (1-5)
Manual	30	4	8	3
Automated	10	2	7	3
Manual	15	2	7	4

of time and complexity. It allowed them to familiarize themselves with the features, but it was also tedious. However, the second step shed new light on the features and introduced an organizational perspective. This enriched the final step, which involved correcting the initially created manual model, improving both the categories and the level of confidence.

4.2. Analysis

The expert knows perfectly the user stories and did not make any comment on the roles or features identified from these user stories. This confirms that the first steps of our process did not have any negative impact while processing the initial dataset.

The first step was the longest, the expert found hard the creation of the first clusters. The trust on the expert feature models during the first step was 3/5. In the second step for the comparison with our synthesized feature models, the expert grade is 3/5. The expert noticed functional differences with his work. In particular, he made the observation that our feature model is closer to the implementation and the feature model built by hand is more abstract.

In the third step, the expert edited his feature model and gave a trust score of 4/5. This means that our synthesized feature model, even if it did not provide him a complete solution, helped the expert to refine his model and obtain a more accurate point of view on the whole system. The expert said that: "*the synthesized feature model is a time saver and it is easier to correct his feature model than building it manually from scratch*". The expert did not find inconsistency in the synthesized feature model. The expert said that: "*the identified cross-tree constraints are too many, with a lot of false positives*".

In the third step, the expert noticed some mistakes in his model that have to be corrected. For example, the expert refined one cluster by dividing it in two clusters. At the end, the feature model had more specific clusters, which correspond to more focused functional points. The expert also renamed some clusters using names given by our model.

On the example, the expert constructed 4 clusters in *Step 1* and 5 clusters in *Step 3*. During Step 2, the expert suggested to add two clusters and merge one in our feature model.

On the larger sample, the expert made 10 clusters in *Step 1*. The expert added 2 clusters for a total of 11 clusters in our model during *Step 2*. Then in *Step 3* the expert renamed the clusters and rearranged the features.

4.3. Discussion

Main findings

We designed a method based on the user stories of existing products, relying on a VCS platform to guide a migration to a product line approach. Our output is actually a particular point of view of the analyzed system: a feature model. It represents the current system based on the features identified from the user stories. We leverage natural language processing, clustering and relational concept analysis to automatically obtain relevant and well-organized models. The current output is already usable and useful for our industrial partner. All our planned enrichment to the process will improve the overall quality. One of the lessons learnt during the workshop with the expert is that analyzing and correcting an automatically generated feature model is easier and faster than building it manually. Indeed an expert can, with low effort, analyze the generated information, while building it from scratch is laborious and time-expensive.

Several improvements could be considered to enhance the approach: The feature names could be abstracted to more general concepts or characteristics to align them with standard feature definitions. This would improve the readability of the feature model and enable the formation of more meaningful groups. The inclusion of multiple industry experts in the validation process would increase the robustness of the case study and mitigate bias introduced by a single expert point of view. Comparing our approach with existing automated feature model synthesis methods would provide another perspective on the evaluation. Our process generated a large number of cross-tree constraints, some of which were redundant or irrelevant. As our approach is semi-automatic, we asked the expert to verify and reduce these constraints.

Threats to Validity

One threat is about the generalization of the approach and the results of the case study. Indeed, the case study was conducted with our industrial partner, and the tool has been designed built on the technology used by

this partner. In particular, user stories are described within GitLab issues. However, we expect our approach to be applicable to a large spectrum of projects.

The main requirement is the meaning of the user stories, they should represent a feature relevant for a user. If the user and the feature can be identified within a user story, then our approach can be applied. Even with a set of user stories not formatted like ours, a pre-processing step is possible to clean them.

The dataset we used to validate our approach contains 127 user stories, 6 products and 127 features. We assume that this is a relevant set of features that can be considered generic enough to show the feasibility. In terms of applicability, our approach showed good scalability during testing. The notebook runs in less than one hour (including clustering and RCA). Overall, our approach should remain feasible for real-world scenarios with minor domain and dataset adaptation.

Defining user stories as requirements and managing them together with source code through merges in a VCS are a common practice in nowadays software projects. This workflow is popular in companies adopting agile methods in their software projects. Our approach is thus applicable for any software using user stories to describe them and using merges handled by a VCS to implement their features. Even with a different technical implementation of such process, for example with user stories externalized from the VCS, the approach remains applicable, with a different tooling.

The validation of our approach is made on existing products of our industrial partner. During the workshop with the expert, we have compared two ways of producing the feature model: from scratch and adapting a feature model generated thanks to our approach. From this case study, we conclude that our generated feature model is relevant and useful. However, one threat to validity resides in the fact that we have provided the expert with a single automatically-generated feature model, using our approach. So on the one hand it may happen that any generated information from the user stories, without using our approach, may be a good help to build the feature model. On the other hand, we did not evaluate independently the benefits of each part of our process. We thus plan to organize a second workshop with experts, in order to provide them with different generated artifacts, and to compare the relevance of the different parts of our approach. In this case study, we wanted to focus in particular on the evaluation of the output of the process. Since the expert was already familiar with the products and had

prior knowledge of feature modeling, this should be considered when generalizing the methodology. Replicating the case study with a person who does not know the dataset may lead to other results. The parameters of natural language processing methods are usually dependent on the used dataset. We tried several variations of models and parameters. In the end, the pipeline word2vec for vectorisation, elbow method and then clustering with k-means were the most efficient in our case. In another dataset, the parameters should be adjusted and the models tuned for better results.

5. Related Work

Agile SPL principles. It is natural to think of combining Agile approaches and software product line engineering, as both promote reusability. Diaz et al. did a systematic literature review on Agile product line engineering has been presented [24]. da Silva et al. conducted a systematic mapping study on Agile software product lines [25]. Hanssen proposed enabling factors for Agile software product line engineering [26]. He highlighted the different natures of both approaches, SPL being "pro-active and plan-driven development", while Agile development promotes "re-active and change-driven development". Among the enablers, we can find this advice: keep continuity between product development and core assets; and simplify requirement management. By proposing a reengineering process which considers a chain of connected artifacts (from user stories to source code file names), our approach aims to ensure part of this continuity. Founding the process on user stories contributes to keeping requirements in the spirit of Agile development.

Version Control Systems and SPL. Version Control Systems are a common practice to manage sequential revisions of a software, but they hardly manage the system variants. Authors of [27] review the existing strategies that have been proposed in *Variation control systems*. One recent proposal is ECSEST (Extraction and Composition for Systems Evolving in Space and Time), which aims to support variants composition while managing feature revisions [28]. In our work, we do not address specifically the problem of maintaining revisions of various variants and knowledge on feature location. We aim at exploiting the connection between user stories and the code changes to identify and maintain a variability model. Dintzner et. al proposed a co-evolution of features and code [29]. Their tool (a FEVER extension) relies on the commits in a VCS, and maintains a mapping between

features and assets. The proposal is applied to the variability model of the Linux kernel which is expressed in the Kconfig language. A difference with our work is that we use the chain of mappings that goes from user stories to files to synthesize a feature model. We also use NLP techniques to identify features and RCA to identify logical constraints.

Natural Language Processing applied to requirements in SPLs. Bakar et al. conducted a systematic literature review on approaches that extract features from natural language requirements in software product lines [30]. Niu et al. deal with textual documents to identify functional requirement profiles (FRPs) for building a variability model written in OVM (Object Variability Model) [31]. The FRPs are "user-visible system functionalities" written as pairs composed with a verb followed by its direct object. Authors use various techniques to identify indexing units: looking for verbs, stemming, part-of-speech tagging, removing stop words, and identifying linguistic affinities in the form of 2-words units. In a second paper, authors examined the interaction between quality requirements and functional requirements through formal concept analysis [32]. To this end, they build a table where objects are scenarios and attributes are the requirements. The concept lattice then highlights and organizes the requirement interactions. Another approach (FENL) is presented by Bakar et al. [33]. The authors use LSA (Latent Semantic Analysis) followed by clustering algorithms (including k-means) to identify features from software reviews. In our work, we analyze user stories, and we use a word2Vec pre-trained model for the vectorization, and k-means for the clustering, after having determined the number of clusters with Elbow. Mefteh et al. [34] more specifically use multilingual functional requirements, that are short sentences such as "The system should block and remove spyware". These requirements are analyzed to extract features, using various techniques. In our NLP process we used the user story format for the decomposition and separated the role and the feature. Mefteh et al. use functional requirements only from the system point of view. For text representation/vectorization, they use TFIDF and add synonyms coming from a thesaurus, while we use a word2Vec pre-trained model.

General approaches for feature model synthesis. A lot of approaches have been proposed for building feature models. We review some representative proposals. Efficient algorithms for feature graph extraction from CNF and DNF are presented by She et al. [35]. Linsbauer et al. designed an approach

based on a genetic algorithm [36]. The issue of extracting the feature models from large collections of partial product descriptions is addressed by Davril et al. [37]. A method combining an interaction with the practitioner to parameterize the extraction and an automatic algorithm merging product descriptions to compute the feature hierarchy and then add variability is described by Acher et al. [38]. Synthesis of attributed feature models is studied by Becan et al. [39], and they explore the usage of ontologies in the process [40]. Mefteh et al. [41] built feature models using requirements expressed through use case diagrams, scenarios and functional requirements.

In contrast to these works, we focused in our approach on software products described by their user stories and VCS information. We are thereby targeting software development projects adopting modern, agile and centralized project management around a VCS platform. We also leveraged different techniques, like NLP, clustering and FCA/RCA.

Approaches leveraging Formal Concept Analysis. Formal Concept Analysis (FCA) has been explored in several approaches concerned by logical relationships extraction or feature model synthesis. Traditional usages with feature models are identified by Loesch et al. [42]. Ryssel et al. propose an approach for extracting feature diagrams and cross-tree constraints from formal contexts using Formal Concept Analysis [43]. Their approach builds a concept hierarchy and a base of implications. [44] develops an alternative approach for feature model synthesis which structures the tree with internal nodes representing logical operators (e.g. an *or* node) rather than abstract features. In [34], after a step where features are extracted from multilingual functional requirements, Formal Concept Analysis and several heuristics are applied to synthesize a feature model.

A synthesis of the correspondences between conceptual structures and feature models is presented in [15]. It leads to characterize *equivalence classes* of feature models, by their embedding into a conceptual structure, which is canonical.

Relational Concept Analysis (RCA) has also been leveraged in the literature for extracting logical relationships in the context of software product lines. A feature variability model is extracted by Hlad et al. [45] as a first step of an approach that addresses the problem of mapping features and feature interactions to their corresponding code artifacts. RCA is used to build this mapping and to derive logical formulas on features that annotate the source code. Carbonnel et al. [46] use RCA to analyze the variability in a set of

interconnected products, e.g. accounting software (first product family) described by the used database management system (second product family), the programming languages they are written in (third product family), and so on. Tracks to synthesize feature models with references [47] are drawn.

Our positioning. In our work, we take advantage of many of these works. The use of ontologies by Becan et al. [40] and Czarnecki et al. [48] were a source of inspiration. We use the correspondences identified by many researchers along the years between conceptual structures and feature models, and in general the correspondences between conceptual structures and the logical variability structuring of an SPL. Compared to existing work using FCA, we consider the connection of many different artifacts, that range from user stories to impacted files, going through issues, merge operations and changes. Exploiting user stories requires considering roles and features associated to the roles, which also is a specificity of our approach.

6. Conclusion

In this paper we presented an automated process for synthesizing feature models from existing software products built in agile projects. By leveraging natural language processing, vectorization, clustering, and formal and relational concept analysis, we were able to identify and refine concrete and abstract features, as well as establish traceability relations between user stories and source code. Our process is based on natural language processing to identify roles and features. We also cluster the features in abstract features. Then we leverage relational concept analysis to complete an initial feature model with additional constraints. This approach offers a comprehensive way to extract relevant feature models from existing products that can be easily modified by experts to produce the actual feature model of the product line.

As a perspective of this work, we plan to improve our approach by conducting a more deeper analysis of the source code. With the study of the code base at different granularity levels, from high-level architecture to individual lines of code, we want to find patterns, dependencies between features. We want to integrate Abstract Syntax Trees (AST) to identify variability at a finer granularity. AST will allow us to parse the code into syntax elements to understand how different parts of the code interact, and to study when two user stories have to implement their features in the same set of files and other source code artifacts.

Besides this, we plan to integrate the ontology in different ways in our process to improve it by guiding the clustering with the ontological relationships. The ontology will also help us to give more meaningful names to the abstract features. Expected improvements are on the identification of abstract features and the refinement of their hierarchical organization. In addition, we plan to analyze the implications, especially those with null support that often correspond to accidental exclusions. We also would like to combine the use of the ontology and the concept lattice generated by relational concept analysis to identify relevant *or* groups and alternatives.

References

[1] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering - Foundations, Principles, and Techniques, Springer, 2005.

[2] C. W. Krueger, Easing the transition to software mass customization, in: F. van der Linden (Ed.), Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers, Vol. 2290 of Lecture Notes in Computer Science, Springer, 2001, pp. 282–293. doi:10.1007/3-540-47833-7_25.
URL https://doi.org/10.1007/3-540-47833-7_25

[3] F. van der Linden, Software product families in europe: The esaps & café projects, IEEE Softw. 19 (4) (2002) 41–49. doi:10.1109/MS.2002.1020286.
URL https://doi.org/10.1109/MS.2002.1020286

[4] M. Abbas, R. Jongeling, C. Lindskog, E. P. Enoiu, M. Saadatmand, D. Sundmark, Product line adoption in industry: an experience report from the railway domain, in: R. E. Lopez-Herrejon (Ed.), SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A, ACM, 2020, pp. 3:1–3:11. doi:10.1145/3382025.3414953.
URL https://doi.org/10.1145/3382025.3414953

[5] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, J. Softw. Evol. Process. 25 (1) (2013) 53–95. doi:10.1002/smr.567.
URL https://doi.org/10.1002/smr.567

[6] D. Spinellis, Version control systems, IEEE Softw. 22 (5) (2005) 108–109.

[7] M. Cohn, User Stories Applied: For Agile Software Development, Addison Wesley Longman Publishing Co., Inc., USA, 2004.

[8] B. Ganter, R. Wille, Formal Concept Analysis - Mathematical Foundations, Springer, 1999.

[9] M. Rouane-Hacene, M. Huchard, A. Napoli, P. Valtchev, Relational concept analysis: mining concept lattices from multi-relational data, Ann. Math. Artif. Intell. 67 (1) (2013) 81–108.

[10] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, Featureide: An extensible framework for feature-oriented software development, Sci. Comput. Program. 79 (2014) 70–85.

[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, 2013, pp. 3111–3119.

[12] H.-H. Bock, Clustering Methods: A History of k-Means Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 161–172.

[13] J. L. Guigues, V. Duquenne, Familles minimales d'implications informatives résultant d'un tableau de données binaires, Mathématiques et sciences humaines 95 (1986) 5–18.
URL http://www.numdam.org/item/MSH_1986__95__5_0/

[14] J. Galasso, M. Huchard, Extending boolean variability relationship extraction to multi-valued software descriptions, in: R. E. Lopez-Herrejon, J. Martinez, W. K. G. Assunção, T. Ziadi, M. Acher, S. Vergilio (Eds.), Handbook of Re-Engineering Software Intensive Systems into Software Product Lines, Springer International Publishing, 2023, pp. 143–173. doi:10.1007/978-3-031-11686-5_6.
URL https://doi.org/10.1007/978-3-031-11686-5_6

[15] J. Carbonnel, M. Huchard, C. Nebut, Modelling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions, Journal of Systems and Software 152 (2019) 1–23.

[16] K. Bertet, C. Demko, J.-F. Viaud, C. Guérin, Lattices, closures systems and implication bases: A survey of structural aspects and algorithms, Theoretical Computer Science 743 (2018) 93–109. doi:https://doi.org/10.1016/j.tcs.2016.11.021.
URL https://www.sciencedirect.com/science/article/pii/S0304397516306806

[17] J. Baixeries, V. Codocedo, M. Kaytoue, A. Napoli, Three views on dependency covers from an FCA perspective, in: D. Dürrschnabel, D. López-Rodríguez (Eds.), Formal Concept Analysis - 17th International Conference, ICFCA 2023, Kassel, Germany, July 17-21, 2023, Proceedings, Vol. 13934 of Lecture Notes in Computer Science, Springer, 2023, pp. 78–94. doi:10.1007/978-3-031-35949-1_6.
URL https://doi.org/10.1007/978-3-031-35949-1_6

[18] J. Carbonnel, K. Bertet, M. Huchard, C. Nebut, FCA for software product line representation: Mixing configuration and feature relationships in a unique canonical representation, Discret. Appl. Math. 273 (2020) 43–64. doi:10.1016/J.DAM.2019.06.008.
URL https://doi.org/10.1016/j.dam.2019.06.008

[19] M. Wajnberg, Analyse relationnelle de concepts : une méthode polyvalente pour l'extraction de connaissance, Thèse de doctorat, Université du Québec à Montréal, Université de Lorraine (Nov. 2020).
URL https://hal.science/tel-03042085

[20] R. Rehurek, P. Sojka, Gensim–python framework for vector space modelling, NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic 3 (2) (2011).

[21] A. Gutierrez, M. Huchard, P. Martin, FCA4J: A java library for relational concept analysis and formal concept analysis, in: Proceedings of the Sixteenth International Conference on Concept Lattices and Their Applications (CLA 2022) Tallinn, Estonia, June 20-22, 2022., Tallinn, Estonia, June 20-22, 2022, Vol. 3308 of CEUR Workshop Proceedings, CEUR-WS.org, 2022, pp. 207–212.

[22] A. Bazin, A depth-first search algorithm for computing pseudo-closed sets, Discret. Appl. Math. 249 (2018) 28–35.

[23] T. Georges, M. Huchard, C. Tibermacine, C. Nebut, M. Konig, Bridging the gap between user-stories and feature models by leveraging version control platforms (Aug. 2025). doi:10.5281/zenodo.16799821.
URL https://doi.org/10.5281/zenodo.16799821

[24] J. Díaz, J. Pérez, P. P. Alarcón, J. Garbajosa, Agile product line engineering - a systematic literature review, Softw. Pract. Exp. 41 (8) (2011) 921–941.

[25] I. F. da Silva, P. A. da Mota Silveira Neto, P. O'Leary, E. S. de Almeida, S. R. de Lemos Meira, Agile software product lines: a systematic mapping study, Softw. Pract. Exp. 41 (8) (2011) 899–920.

[26] G. K. Hanssen, Agile software product line engineering: enabling factors, Softw. Pract. Exp. 41 (8) (2011) 883–897.

[27] L. Linsbauer, F. Schwägerl, T. Berger, P. Grünbacher, Concepts of variation control systems, Journal of Systems and Software 171 (2021) 110796.

[28] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, S. Fischer, R. E. Lopez-Herrejon, A. Egyed, Evolving software system families in space and time with feature revisions, Empir. Softw. Eng. 27 (5) (2022) 112.

[29] N. Dintzner, A. van Deursen, M. Pinzger, FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems, Empir. Softw. Eng. 23 (2) (2018) 905–952.

[30] N. H. Bakar, Z. M. Kasirun, N. Salleh, Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review, Journal of Systems and Software 106 (2015) 132–149.

[31] N. Niu, S. M. Easterbrook, Extracting and modeling product line functional requirements, in: 16th IEEE International Requirements Engineering Conference, RE 2008, 8-12 September 2008, Barcelona, Catalunya, Spain, IEEE Computer Society, 2008, pp. 155–164.

[32] N. Niu, S. M. Easterbrook, Concept analysis for product line requirements, in: K. J. Sullivan, A. Moreira, C. Schwanninger, J. Gray (Eds.), Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009, ACM, 2009, pp. 137–148.

[33] N. H. Bakar, Z. M. Kasirun, N. Salleh, H. A. Jalab, Extracting features from online software reviews to aid requirements reuse, Applied Soft Computing 49 (2016) 1297–1315.

[34] M. Mefteh, N. Bouassida, H. Ben-Abdallah, Mining feature models from functional requirements, Comput. J. 59 (12) (2016) 1784–1804.

[35] S. She, U. Ryssel, N. Andersen, A. Wasowski, K. Czarnecki, Efficient synthesis of feature models, Inf. Softw. Technol. 56 (9) (2014) 1122–1143.

[36] L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Feature model synthesis with genetic programming, in: C. Le Goues, S. Yoo (Eds.), Search-Based Software Engineering, Springer International Publishing, Cham, 2014, pp. 153–167.

[37] J. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, P. Heymans, Feature model extraction from large collections of informal product descriptions, in: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, ACM, 2013, pp. 290–300.

[38] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, P. Lahire, On extracting feature models from product descriptions, in: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings, ACM, 2012, pp. 45–54.

[39] G. Bécan, R. Behjati, A. Gotlieb, M. Acher, Synthesis of attributed feature models from product descriptions, in: D. C. Schmidt (Ed.), Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, ACM, 2015, pp. 1–10.

[40] G. Bécan, M. Acher, B. Baudry, S. B. Nasr, Breathing ontological knowledge into feature model synthesis: an empirical study, Empir. Softw. Eng. 21 (4) (2016) 1794–1841.

[41] M. Mefteh, N. Bouassida, H. Ben-Abdallah, Feature model synthesis from language-independent functional descriptions, in: 2018 IEEE 16th International Conference on Software Engineering Research, Management and Applications (SERA), 2018, pp. 151–158. doi:10.1109/SERA.2018.8477223.

[42] F. Loesch, E. Ploedereder, Restructuring variability in software product lines using concept analysis of product configurations, in: R. L. Krikhaar, C. Verhoef, G. A. D. Lucca (Eds.), 11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands, IEEE Computer Society, 2007, pp. 159–170.

[43] U. Ryssel, J. Ploennigs, K. Kabitzsch, Extraction of feature models from formal contexts, in: I. Schaefer, I. John, K. Schmid (Eds.), Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2), ACM, 2011, p. 4.

[44] R. Al-Msie'deen, M. Huchard, A. Seriai, C. Urtado, S. Vauttier, Reverse engineering feature models from software configurations using formal concept analysis, in: K. Bertet, S. Rudolph (Eds.), Proceedings of the Eleventh International Conference on Concept Lattices and Their Applications, Košice, Slovakia, October 7-10, 2014, Vol. 1252 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 95–106.

[45] H. Nicolas, B. Lemoine, M. Huchard, A.-D. Seriai, Leveraging relational concept analysis for automated feature location in software product lines, in: GPCE 2021 - 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Association for Computing Machinery (ACM SIGPLAN), Chicago, United States, 2021, pp. 170–183.

[46] J. Carbonnel, M. Huchard, C. Nebut, Exploring the variability of interconnected product families with relational concept analysis, in: Pro-

ceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019, ACM, 2019, pp. 90:1–90:8.

[47] K. Czarnecki, S. Helsen, U. W. Eisenecker, Staged configuration through specialization and multilevel configuration of feature models, Softw. Process. Improv. Pract. 10 (2) (2005) 143–169.

[48] K. Czarnecki, C. H. P. Kim, K. T. Kalleberg, Feature models are views on ontologies, in: Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings, IEEE Computer Society, 2006, pp. 41–51.