Towards Complex Product Line Variability Modelling: Mining Relationships from Non-Boolean Descriptions

Jessie Carbonnel, Marianne Huchard, Clémentine Nebut

Towards Complex Product Line Variability Modelling: Mining Relationships from Non-Boolean Descriptions

Jessie Carbonnel[a], Marianne Huchard[a], Clémentine Nebut[a]

[a]*LIRMM, University of Montpellier and CNRS*
161 rue Ada, 34095 Montpellier, France

Abstract

Software product line engineering relies on systematic reuse and mass customisation to reduce the development time and cost of a software system family. The extractive adoption of a product line requires to extract variability information from the description of a collection of existing software systems to model their variability. With the increasing complexity of software systems, software product line engineering faces new challenges including variability extraction and modelling. Extensions of existing boolean variability models, such as multi-valued attributes or UML-like cardinalities, were proposed to enhance their expressiveness and support variability modelling in complex product lines. In this paper, we propose an approach to extract complex variability information, i.e., involving features as well as multi-valued attributes and cardinalities, in the form of logical relationships. This approach is based on Formal Concept Analysis and Pattern Structures, two mathematical frameworks for knowledge discovery that bring theoretical foundations to complex variability extraction algorithms. We present an application on product comparison matrices representing complex descriptions of software system families. We show that our method does not suffer from scalability issues and extracts all pertinent relationships, but that it also extracts numerous accidental relationships that need to be filtered.

Keywords: Complex Software Product Line, Reverse Engineering, Variability Modelling, Extended Feature Models, Formal Concept Analysis, Pattern Structures

1. Introduction

Software Product Line Engineering (SPLE) [55] is an approach based on systematic reuse and mass customisation that aims at reducing the development time and cost of a set of similar software systems. The core of this approach is based on the development of a generic software architecture, on which variable and reusable software artefacts can be plugged depending on given requirements. In this way, several different yet similar software systems, also called software variants or a software family, can be automatically derived. The generic architecture, the reusable artefacts and the set of software systems that can be derived form the Software Product Line (SPL). In this process, variability modelling is a central task that documents common and variable artefacts, along with the way they can be combined to constitute a valid software system. Representing artefacts by features, a feature being a distinguishable characteristic of one or several software systems, is the most commonly used variability modelling approach, where feature models (FMs) [34] are the *de facto* standard models. FMs organise a set of features in a hierarchy representing several levels of details, and express constraints between these features to depict their possible compatibility in a software system. Basic FMs, also called boolean FMs, only represent boolean features, i.e., characteristics that can be present or not in a software system. However, the growing importance of ultra large-scale systems and systems-of-systems affects traditional software product line approaches, and gives new challenges in the domain of *complex software product lines* [33]. Limitations regarding expressiveness of boolean FMs have been addressed, and extensions to overstep them have

Email addresses: jcarbonnel@lirmm.fr (Jessie Carbonnel), huchard@lirmm.fr (Marianne Huchard), nebut@lirmm.fr (Clémentine Nebut)

been proposed, e.g., feature cardinalities [19], group cardinalities [18, 57], or multi-valued attributes [19, 7, 9]. Aside from these extended FMs, other types of variability models try to tackle the problem of representing complex SPLs, such as orthogonal variability models [55] or the common variability language [32], with separation of concerns and by using references to connect several variability models.

The survey made by Berger et al. in 2013 [11] shows that a significant part of companies opts for an *extractive* (or *bottom up*) adoption of SPL. This means that they perform a migration from individually developed software variants, possibly without reuse effort, to an SPL approach [38]. Migrating from an existing collection of software variants to an SPL is an arduous task that implies to design a variability model based on descriptions of the existing software family [37]. Numerous papers addressing automated or semi-automated synthesis of FMs from software variant descriptions can be found in the literature [1, 58, 21, 2, 30, 31, 22, 44, 42, 48]. However, these papers only focus on boolean FM synthesis. In this paper, we address the problem of extracting complex variability information from software variant descriptions, as a part of the process of reverse engineering extended FMs. We focus on two kinds of complex variability information that were introduced through two FM extensions: *UML-like cardinalities* and *multi-valued attributes*. To the best of our knowledge, only Becan et al. [8] work on extracting complex variability models (in the form of FMs extended with attributes) from software system descriptions, by taking into account boolean features as well as multi-valued attributes. Here, we propose a method to extract logical relationships that can then be used to build an FM including complex variability; however, the synthesis of extended FMs is not studied in this paper.

The proposed extraction approach is based on Formal Concept Analysis (FCA) [26], a mathematical framework for data analysis, information management and knowledge representation, which is widely used for knowledge discovery in data [54]. From a set of objects described by binary attributes, FCA organises the objects depending on the attributes they share in a structure called a concept lattice. Concept lattices naturally highlight constraints between the binary attributes and therefore support the extraction of logical relationships that are true for the considered set of objects. FCA has numerous applications in software engineering [62, 43, 52, 64, 63], including boolean FM extraction from software system descriptions [58, 2, 17]. However, none of those existing FCA-based approaches deals with complex variability. In previous work [17], we studied the parallel between boolean FMs and FCA structures, and proposed a sound and complete FCA-based method to extract boolean variability information and support boolean FM synthesis. In this paper, we investigate an FCA extension to broaden the previous extraction method and take into account complex variability information. More specifically, we study Pattern Structures [25], an extension of traditional FCA which allows to consider more complex data than binary attributes to describe the objects, as for instance multi-valued attributes. We design a method that uses FCA and Pattern Structures to extract logical relationships involving features, but also cardinalities and multi-valued attributes. The method proposed here is able to take into account complex software system descriptions, i.e., descriptions that are not restricted to boolean characteristics. As complex descriptions, we study Product Comparison Matrices (PCMs), a formalism displaying products of a same family against both boolean and multi-valued characteristics in a tabular way. We used three existing datasets already studied in the product line community: PCMs from Wikipedia [59], `Robocode` [46] and `JHipster` [29]. Our approach offers a mathematical framework for complex variability information extraction based on a unique and canonical structure. The types of logical relationships that can be extracted with our approach include the ones extracted by Becan et al. [8] (i.e., feature groups and complex binary implications), as well as complex co-occurrences and mutex: these relationships correspond to the logical semantics of extended FMs, as we will show in this paper. This work is a step towards the global objective of facilitating the transition of complex software variants that have been individually developed, to software reuse and mass-customisation approaches. This paper extends previous work presented in [16] by defining and evaluating the complex variability extraction approach.

The remainder of this paper is organised as follows. Logical relationships present in boolean FMs and their extensions for complex variability modelling are presented in Section 2, where we identify the kinds of logical relationships they represent. In Section 3, we present the theoretical bases of Formal Concept Analysis and Pattern Structures. We expose and illustrate our extraction approach in Section 4, and evaluate its applicability and usefulness in Section 5. Related work is discussed in Section 6, and Section 7 concludes the paper.

2. Identifying Logical Relationships in Feature Models and two of their Extensions

In this section, we study the logical semantics in boolean feature models and in two of their prevalent extensions which have been proposed to document and manage more complex variability information.

2.1. Logical Relationships in Boolean Feature Models

Boolean Feature Models (FMs) [34] are a family of graphical languages enabling to define the scope of a product line in terms of features (i.e., distinguishable characteristics or behaviours) and constraints between these features. Figure 1 presents a boolean FM about web browsers providing accessibility features.

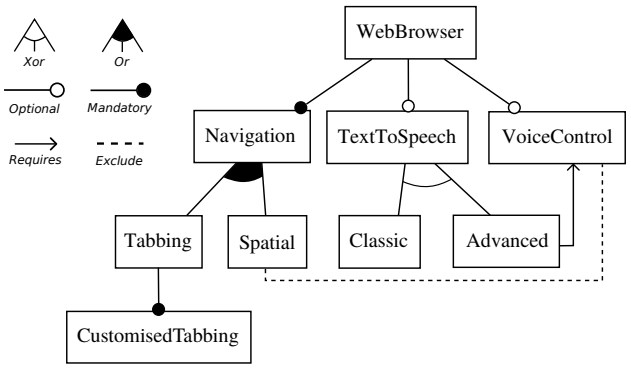


Figure 1: Boolean FM about web browsers

A boolean FM represents a finite set of features in a hierarchy (called a *feature tree*) expressing child-parent (refinement) relationships. Boolean FMs express constraints to guide the user into selecting a subset of the presented features. Constraints can be represented graphically by decorating the edges of the feature tree: these constraints show how the selection of a feature may affect the selection of its child features. A black disc forces the selection of the child feature when the parent feature is selected (*mandatory relationship*), whereas a white circle indicates that the child feature can be optionally selected (*optional relationship*). Several child features can be grouped, a group being depicted by an arc indicating the number of features which can be selected: a black-filled arc states that at least one child feature of the group has to be selected (*or-group*), and a non-filled arc shows that exactly one child feature of the group has to be selected (*xor-group*). Finally, additional constraints that cannot be expressed on the feature tree edges can be added. They are typically *requires* and *exclude* constraints, and are called *cross-tree constraints*. The boolean FM of Figure 1 states that: all web browsers support at least one navigation strategy amongst tabbing navigation (navigating between focusable elements using the tabular key) and spatial navigation (navigating between focusable elements using the arrow keys). If the web browser allows tabbing navigation, then the user has to customise it (i.e., indicate which elements should have the focus). A text to speech feature may be optionally proposed, which can be either classic or advanced. A web browser may also provide a voice control feature. The advanced text to speech requires voice control, but voice control is incompatible with spatial navigation. A subset of features satisfying all the FM constraints is called a valid configuration of the FM. The set of all valid configurations of an FM is called the scope of the product line (i.e., the description of all derivable products) or its *configuration semantics*.

FMs also give knowledge about the modelled domain and the interactions of some of its concepts: this knowledge is called the *ontological semantics* of the FM [61]. From a strictly logical point of view, moving the feature `CustomisedTabbing` of Figure 1 under the root feature `WebBrowser` as an optional child feature, and adding two "requires" cross-tree constraints between the two features `CustomisedTabbing` and `Tabbing` leads to an equivalent FM (i.e., representing the same set of valid configurations). But the resulting refinement relationship (`CustomisedTabbing` refines `WebBrowser` rather than `Tabbing`) is different: a part of the ontological semantics of the original FM is thus lost by this transformation.

Extracting FM relationships that convey correct ontological knowledge from a set of variant descriptions without using external ontologies or relying on an expert intervention is quite infeasible. Besides, the parallel between boolean FMs and propositional logic has been widely studied [45, 21, 10]; writing boolean FMs in the form of propositional formulas allows to represent the *logical semantics* of the FM constraints and to depict feature compatibility through logical relationships. Extracting logical relationships from a set of variant descriptions is easier and less error-prone than extracting ontological FM relationships. In fact, several different FM ontological semantics may correspond to a single FM logical semantics, and it is difficult to automatically detect the most meaningful one. For these reasons, we focus in this paper on the extraction of variability information in the form of logical relationships. Assessing the

ontological knowledge that may correspond to the extracted logical semantics by relying on experts and ontological resources is left as future work.

Table 1: Logical semantics of boolean FM constraints; p represents a parent feature, c, c_i a child feature of p, and f_i any feature

FM constraints	**Logical semantics**
child-parent	$c \Rightarrow p$
optional	none
mandatory	$p \Rightarrow c$
or-group	$p \Rightarrow \{c_1 \vee ... \vee c_n\}$
xor-group	$p \Rightarrow \{c_1 \oplus ... \oplus c_n\}$
requires	$f_1 \Rightarrow f_2$
exclude	$f_1 \Rightarrow \neg f_2$

Table 1 shows the logical semantics of boolean FMs as presented in [10, 21, 53]. Therefore, we can identify in boolean FMs the following logical relationships between features: binary implications $f_1 \Rightarrow f_2$ (from child-parent, mandatory and requires relationships), mutex $f_1 \Rightarrow \neg f_2$ (from exclude constraints), the particular case of double implications that we call co-occurrences $f_1 \Leftrightarrow f_2$ (from double requires relationships), or-groups and xor-groups.

2.2. Logical Relationships in Extended Feature Models

In what follows, we consider the extended FM of Figure 2. The FM represents web browsers such as the FM in Figure 1, but with some additional information. The feature `VoiceControl` now possesses an attribute `Version` of type integer that defines the version number of the used voice control software. The cardinality of the feature `CustomisedTabbing` states that a web browser may define several tabbing navigation strategies. The two group cardinalities constrain the number of features that can be selected in the corresponding group. The constraint `Advanced` $\Rightarrow$ `VoiceControl::Version` ≥ 2 is a *requires* constraint involving a feature and an attribute value.

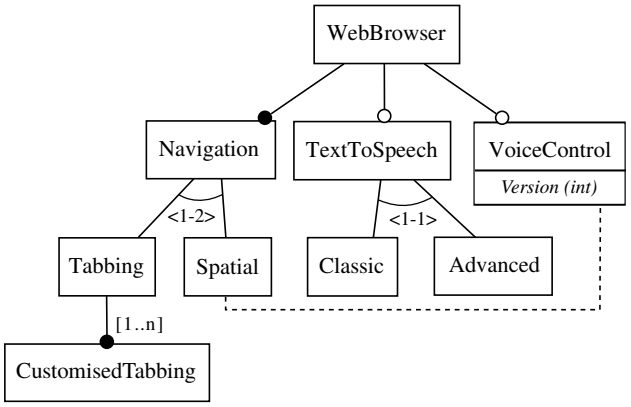


Figure 2: Extended feature model with a feature attribute, a feature cardinality and two group cardinalities

2.2.1. Attributes

An extension of boolean FMs proposes to add multi-valued attributes. An attribute possesses a type (e.g., integer, string, enumeration) and is associated with one feature of the FM. This extension enables to model more detailed information without complexifying the FM [19]. In fact, in an "all-feature view" of the FM, each attribute value would be represented as one feature. In the cases where the possible values are too numerous (e.g., numerical values), the number of features would be too important and the FM unintelligible. For instance, introducing an attribute `Version` of type integer in the feature `VoiceControl` reduces the number of features necessary to represent this information, as shown in Figure 3.

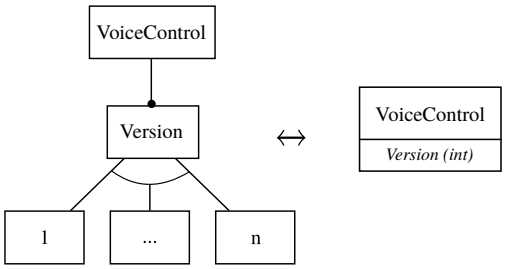


Figure 3: Representing numerous values with features (left-hand side) versus with an attribute (right-hand side)

Introducing attributes and their values in FMs allows to express more complex variability information, i.e., *requires* and *exclude* constraints between features and/or attribute values. In our example of Figure 2, the constraint `Advanced` $\Rightarrow$ `VoiceControl::Version` $\geq$ `2` involves a feature and an attribute value. As stated by their name, feature-groups and feature tree are only defined over the set of features, so they do not involve attributes.

To sum up, the variability information induced by attributes thus corresponds to the following logical relationships: binary implications, co-occurrences and mutex between a feature and an attribute value, or between two attribute values. We call these additional logical relationships *augmented variability information*.

2.2.2. Cardinalities

Another extension of FMs introduces UML-like cardinalities on features and on feature-groups [20].

Feature-group cardinalities depict the minimum and the maximum number of sub-features that can be selected in groups (denoted $\langle min - max \rangle$). Therefore, boolean FM group notations for xor-groups and or-groups do not stand any more: xor-groups are defined by a cardinality $\langle 1 - 1 \rangle$, while or-groups by a cardinality $\langle 1 - n \rangle$. This can be written in propositional logic by representing each combination of sub-features of the group that is allowed by the cardinality. Let p be a feature, and $\{f_1, f_2, f_3\}$ be a feature-group with p as a parent and associated with the cardinality $\langle 2 - 3 \rangle$. The logical relationship representing this group is:

$$p \rightarrow ((f_1 \wedge f_2 \wedge \neg f_3) \vee (f_1 \wedge f_3 \wedge \neg f_2) \vee (f_2 \wedge f_3 \wedge \neg f_1) \vee (f_1 \wedge f_2 \wedge f_3))$$

Feature cardinalities define the minimum and the maximum number of occurrences of a given feature in a valid configuration (denoted $[min..max]$). Except for the graphical notation that can be different, feature cardinalities can be seen as multi-valued attributes: in the example of Figure 2, the cardinality of `CustomisedTabbing` could be represented by an attribute `Occurrence` of type integer. Hence, feature cardinalities could also be represented by features, as we have seen before with attribute values. Note that even though the representation is different, the information remains the same. Figure 4 depicts the different (yet equivalent) ways to represent a feature cardinality, which subsumes Figure 3.

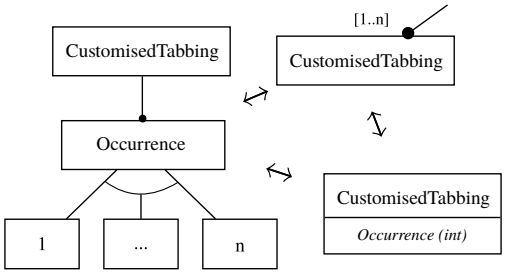


Figure 4: Different ways to represent a feature cardinality

Group-cardinalities allow to consider that all feature-groups are the same kind of variability information: not as xor-/or-groups, but as feature-groups associated with a cardinality. Thus, concerning feature groups, this FM extension

does not add any new kind of variability information compared to the one found in boolean FMs, but generalises two existing ones. Moreover, as feature cardinalities can be seen as feature attributes of type integer, the kinds of variability information induced by feature cardinalities are the same as the ones induced by attributes, i.e., *augmented variability information*.

3. Formal Concept Analysis, Pattern Structures, and Variability

In this section, we present the basics of Formal Concept Analysis theory (Section 3.1), its connections with variability extraction (Section 3.2), and Pattern Structures (Section 3.3), one of the FCA extensions which allows to extract logical relationships between more complex data types than binary attributes (e.g., multi-valued attributes, numerical values).

3.1. Formal Concept Analysis

Formal Concept Analysis (FCA) [26] is a mathematical framework for data analysis, information management and knowledge representation. From a set of objects that are described by a set of binary attributes, the application of FCA provides a classification of the set of objects depending on the attributes they share. An overview of the FCA process is illustrated in Figure 5 and is detailed in what follows.

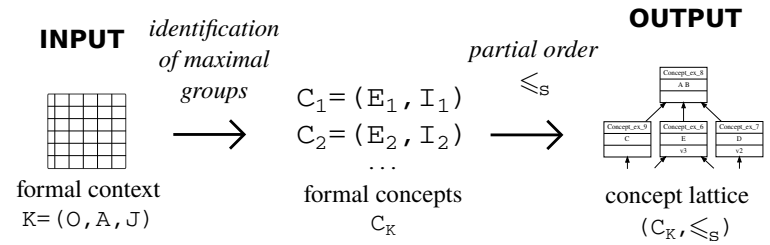


Figure 5: FCA process

As input, FCA takes a *formal context* $K = (O, A, J)$, where O is the set of objects, A is the set of binary attributes and $J \subseteq O \times A$ is a binary relationship stating "which objects possess which binary attributes". A formal context can be represented by a table $O \times A$, where a cross in the cell (o, a) states that the object o possesses the binary attribute a. Table 2 presents an excerpt of a formal context where the objects (lines) represent variants of web browsers, and the binary attributes (columns) represent 9 accessibility features characterising these web browsers. Note that this formal context displays the configuration semantics of the boolean FM from Figure 1.

The application of FCA on a formal context extracts a set of *formal concepts*, where a formal concept represents a maximal set of objects sharing a maximal set of binary attributes. More formally, a formal concept is a pair $C = (E, I)$ where $E = \{o \in O \mid \forall a \in I, (o, a) \in J\}$ is called the concept's extent and $I = \{a \in A \mid \forall o \in E, (o, a) \in J\}$ is called the concept's intent. There is no other object than the ones in E that share all the attributes of I, and there is no other attribute than the ones in I that are shared by all the objects of E. For example, in Table 2, we highlighted the formal concept representing the maximal group of objects (`v2`, `v3`, `v4` and `v5`) and the maximal group of attributes (`WebBrowser`, `Navigation` and `Spatial`) they share. Provided a permutation of the rows and columns, a formal concept can be seen as a maximal rectangle of crosses in the table.

The set C_K of formal concepts of a formal context K can be partially ordered by the set-inclusion order (denoted $\leq_s$) on the concepts' extents (or equivalently the set-containment on the concepts' intents). It is formally written as follows: given two concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$, $C_1 \leq_s C_2$ if and only if $E_1 \subseteq E_2$ (or equivalently $I_1 \supseteq I_2$). For example, let C_1= ({`v4`, `v5`}, {`WebBrowser`, `Navigation`, `Spatial`, `TextToSpeech`, `Classic`}) and C_2 = ({`v2`, `v3`, `v4`, `v5`}, {`WebBrowser`, `Nagivation`, `Spatial`}), we have: $C_1 \leq_s C_2$. This order may be seen as a specialisation relation between the concepts: C_1 is called a sub-concept of C_2, and C_2 a super-concept of C_1. The set of all concepts C_K of a formal context, provided with the order $\leq_s$ forms a lattice structure $(C_K, \leq_s)$ called a *concept lattice*. Figure 6 (left-hand side) presents the concept lattice obtained from Table 2. Concept lattices are canonical, i.e., there exists a unique concept lattice that can be extracted from a given formal context.

Table 2: Formal context with 9 objects representing web browsers and described by 9 binary attributes representing accessibility features. In light gray, we highlight the presence of a concept, composed of (1) an extent, which is a maximal group of objects (`v2`, `v3`, `v4` and `v5`), and of (2) an intent, which is the maximal group of attributes (`WebBrowser`, `Navigation` and `Spatial`) these objects share.

	WebBrowser	**Navigation**	**Tabbing**	**CustomisedTabbing**	**Spatial**	**TextToSpeech**	**Classic**	**Advanced**	**VoiceControl**
v1	×	×	×	×					
v2	×	×			×				
v3	×	×	×	×	×				
v4	×	×			×	×	×		
v5	×	×	×	×	×	×	×		
v6	×	×	×	×		×	×		
v7	×	×	×	×		×	×		×
v8	×	×	×	×		×		×	×
v9	×	×	×	×					×

A naive algorithm [28] to build the concept lattice of a context considers all the subsets of O, and, for any subset $E \in 2^O$ computes I the set of attributes shared by all objects from E, keeps the largest such (E, I) subsets, and then organises these (E, I) pairs by set-inclusion. The number of concepts may reach $2^{min(|O|,|A|)}$ in the worst case, when the concept lattice is isomorphic to the O subset lattice or to the A subset lattice. But fortunately, in real applications, it rarely happens and the efficient algorithms [41, 4, 36] do not follow the naive schema. There is a large literature on efficient lattice construction: a survey and a comparison of the performances of the main algorithms have been proposed by Kuznetsov and Obiedkov in [40]. In addition, one can find on the website of Uta Priss[1] a list of links towards existing tools and libraries implementing these algorithms.

In Figure 6, a concept is represented by a three-part box: the top part shows the concept's name, which is unique in the structure, the middle part displays the concept's intent, and the bottom part the concept's extent. An arrow between two boxes represents the specialisation relation (i.e., partial order) from a concept (arrow's source) to one of its super-concepts (arrow's target). Right-hand side of Figure 6 represents the same concept lattice as in the left-hand side, except this time the concepts are presented in an optimised way by displaying each object and each binary attribute only once in the structure. An object (resp. a binary attribute) is introduced in the lowest (resp. greatest) concept of the concept lattice possessing it. Therefore, a concept inherits all the objects of its sub-concepts, and all the binary attributes of its super-concepts.

An *attribute-concept* is a concept introducing at least an attribute, which means that one of its attributes does not belong to any of its super-concepts; this is the case in Figure 6, for *Concept_10* introducing `Classic`, and for *Concept_13* introducing `Tabbing` and `CustomisedTabbing`. Dually, an *object-concept* is a concept introducing at least an object, which means that one of its objects does not belong to any of its sub-concepts; in Figure 6, *Concept_8*, which introduces `v6` and *Concept_6*, which introduces `v3` are both object-concepts. Some concepts are both object-concept and attribute-concept, such as *Concept_11*, which introduces `Spatial` and `v2`. *Plain-concepts* introduce neither attributes nor objects, as for instance *Concept_4*.

In some applications, taking into account plain-concepts is unnecessary; this is often the case when FCA is used to organise the elements of the initial dataset in a hierarchy [27], and not for clustering. To lighten the produced conceptual structure, one can choose to only construct the concept lattice's sub-hierarchies restricted to attribute- and/or object-concepts. For instance, an interesting sub-hierarchy when dealing with attribute relationships is the one that is restricted to attribute-concepts, called an *attribute-concept partially ordered set*, or AC-poset for short. This structure keeps specialisation order between attributes, along with the extents of the formal concepts introducing

[1] `http://www.upriss.org.uk/fca/fcasoftware.html`

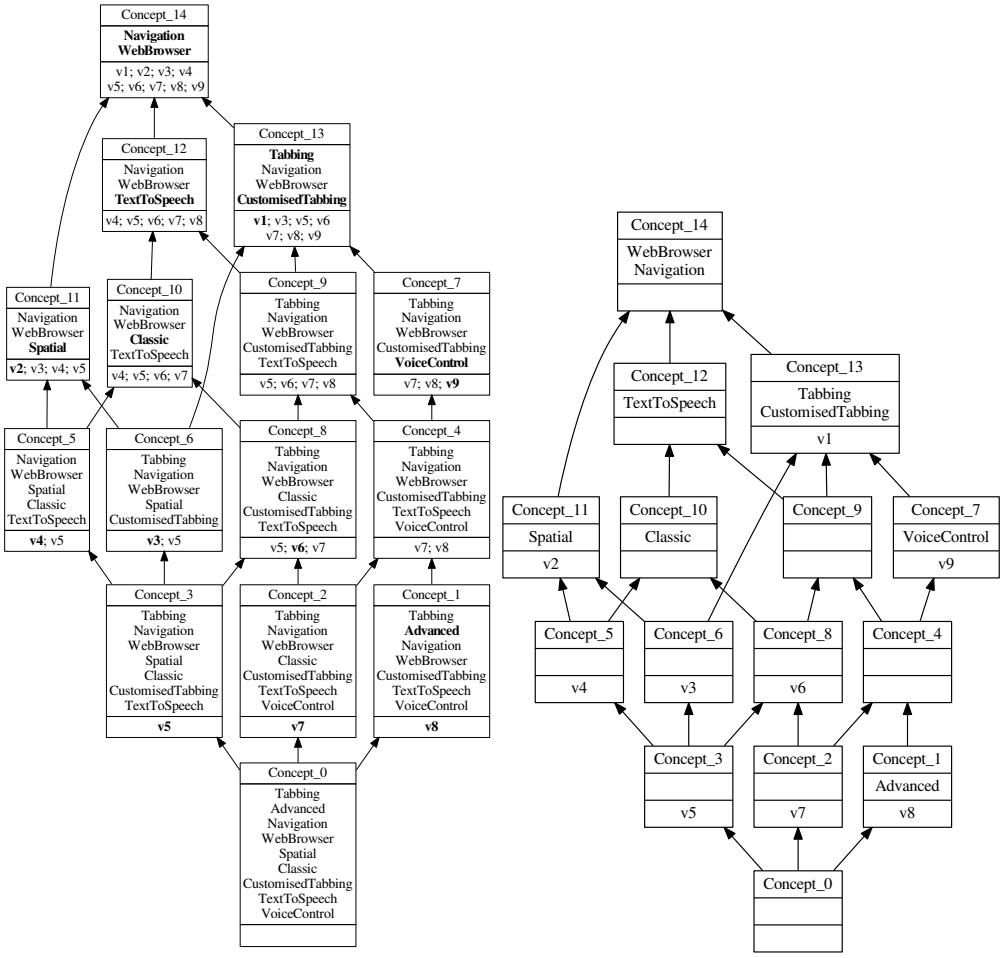


Figure 6: Concept lattice associated with the formal context of Table 2. Left-hand side: full description of concepts' extents and intents. The introduced objects and attributes are presented in bold face. Right-hand side: simplified description of concepts' extents and intents, showing only the introduced objects and attributes.

them. The AC-poset associated with the formal context of Table 2 is presented in Figure 7, where only the 7 attribute-concepts of Figure 6 (*Concept_1*, *7*, *10*, *11*, *12*, *13* and *14*) have been retained.

The AC-poset can be constructed from the concept lattice (by removing all concepts that do not introduce any attribute), but this is not an efficient option, the number of concepts being bounded by $|A|$. A naive algorithm (given in [15]) to build the AC-poset first builds the attribute-concepts and then organises these concepts by set-inclusion. To build all the attribute-concepts, for each attribute a, one has to compute the objects that possess a (which gives the extent of the concept introducing a) and then the attributes which are shared by all these objects (which gives the intent of the concept introducing a). This procedure may produce concept duplicates that have to be removed. Efficient algorithms can be derived from those that build the AOC-posets, which contain both attribute-concepts and object-concepts and are surveyed in [12]. `Galicia`[2], `RCAExplore`[3] and `AOC-poset Builder`[4] are three available tools implementing those algorithms.

[2] `http://www-labs.iro.umontreal.ca/~galicia/`

[3] `http://dataqual.engees.unistra.fr/logiciels/rcaExplore`

[4] `http://www.lirmm.fr/AOC-poset-Builder/`

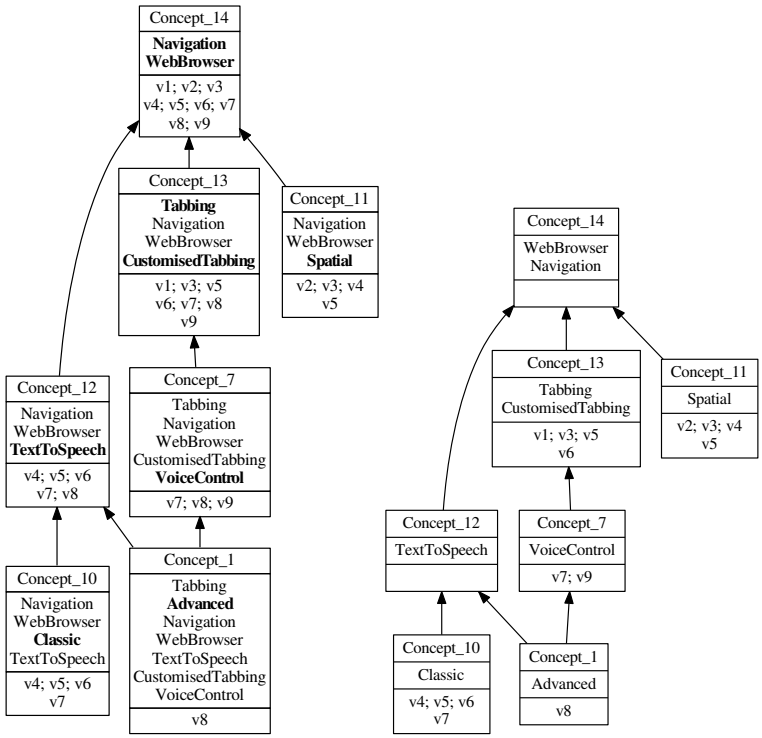


Figure 7: AC-poset associated with the formal context of Table 2. Left-hand side: full description of concepts' extents and intents. The introduced attributes are presented in bold face. Right-hand side: simplified description of concepts' extents and intents, showing only the introduced attributes.

3.2. Formal Concept Analysis and Variability

The way the objects (i.e., similar software systems) and the binary attributes (i.e., software characteristics) are organised in the concept lattice highlights information regarding their variability. More specifically, one can extract by the means of FCA all logical relationships involving the binary attributes, such as binary implications, co-occurrences, mutual exclusions (mutex) and groups (in the sense of the feature groups one can find in boolean FMs). The extraction algorithms rely on the hierarchy of concepts introducing binary attributes; thus, AC-posets may be used instead of concept lattices for this application.

Binary implications can be deduced from the transitive reduction of the partial order between concepts in the AC-poset: if an attribute a_1 is introduced in a sub-concept of another concept introducing the attribute a_2, then $a_1 \Rightarrow a_2$ holds [2]. For instance, in Figure 7, one can see that `Classic` $\Rightarrow$ `TextToSpeech` because *Concept_10* is a sub-concept of *Concept_12*. The tools `java-lattices`[5] and `ConExp`[6] are two popular tools permitting to extract binary implications by the means of FCA. Co-occurrences can be recognized as double binary implications, or when two features are introduced in a same attribute-concept. For instance, `Tabbing` and `CustomisedTabbing` are two co-occurring features because they are introduced in the same *Concept_13*. This type of variability information can be extracted with a graph-search in the AC-poset, or can be deduced from symmetric binary implications extracted with the aforementioned tools. For computing the feature groups, we consider all the sets of pairwise non comparable attribute-concepts (called "Antichains" in the graph theory terminology) by growing size. For each such antichain S, we compute P the lowest attribute-concepts that are greater than all elements of S. Then for each element p of P (such element is candidate to be a parent of the group), we check if the configurations that have an element s of S cover exactly the configurations that have p. If the configuration sets associated with each s form a partition, the set of attribute-concepts S is a xor-group with parent p, otherwise, when they do not form a partition, and S does not

[5] `http://thegalactic.github.io/java-lattices/`
[6] `http://conexp.sourceforge.net/`

include a smaller group, S is an or-group with parent p. We provide a detailed algorithm to extract these groups in [15]. In [15] we also provide an algorithm to compute sets of features that cannot appear together (*nat* for "not all together") by reconsidering all the antichains. The *nat* with cardinal 2 are mutex. An antichain is a *nat* if the features have no configuration in common, it does not contain a xor-group, is not included in a xor-group and, either its size is 2, or it does not contain any smaller *nat*. Ryssel et al. [58], AL-Msie'deen et al. [2] and Shatnawi et al. [60] also provide algorithms to extract mutex from FCA structures. To the best of our knowledge, there is no available tool permitting to extract mutex and feature groups by means of FCA; we implemented these algorithms in our tool `CLEF` presented in Section 5 and available on GitHub.

The presented extraction method is sound and complete according to the general FCA theorems [26]: it allows to extract all the logical relationships (among the four types presented before) that are true for the considered set of objects. Figure 8 presents the grammar of the variability information that can be extracted (except that we do not consider the *nat*, but only the mutex, which correspond to more usual FM cross-tree constraints). We simplify the logical relationships representing feature-groups by introducing the notation $(p, \{f_1, ..., f_n\}, \langle min - max \rangle)$ to be used instead of the one aforementioned in Section 2.2.2.

variability info.	:=	*relationship**
relationship	:=	*implication* | *co-occurrence* | *mutex* | *group*
implication	:=	*feature* '⇒' *feature*
co-occurrence	:=	*feature* '⇔' *feature*
mutex	:=	*feature* '⇒' '¬' *feature*
group	:=	'(' *feature* ',' '{' *feature_set* '}' ',' *cardinality* ')'
feature_set	:=	*feature* | *feature_set* ',' *feature*
feature	:=	**feature_name**
cardinality	:=	'⟨' **nb_min** '-' **nb_max** '⟩'

Figure 8: Grammar of variability information that can be extracted with traditional FCA

3.3. Pattern Structures

Pattern structures [25] have been proposed as a generalisation of FCA to describe a set of objects O with data types that are more complex than binary attributes. The FCA process generalised for pattern structures is presented in Figure 9.

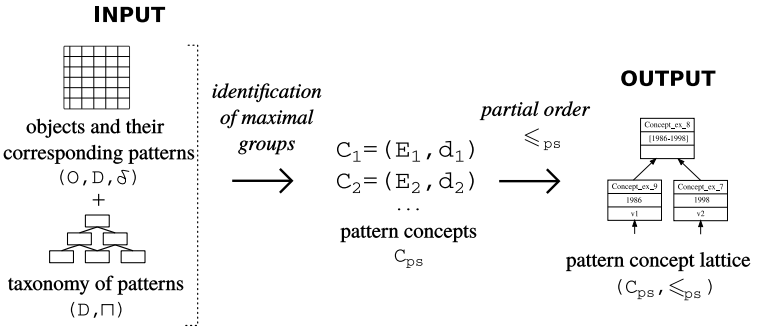


Figure 9: Pattern structure process

In this approach, each object is characterised by a *pattern* (also called a *description*) taken from a set of patterns (denoted D) having the same type. A set of patterns can be of any type of data on which one can establish *similarities*. The similarity of two patterns $d_1, d_2 \in D$ is given by a *similarity operator* (denoted $\sqcap$) that returns the most specific pattern of D representing the similarity of d_1 and d_2, or in other words the most specific generalisation of d_1 and d_2. For

instance, in the software engineering domain, a set of patterns $D_{languages}$ could depict programming languages, and one can define the similarity of the two patterns *Java* and *C++* by a third pattern *Java* $\sqcap$ *C++* = *Object Oriented Language*. Another example may be to define the similarity of two integer intervals by the smallest interval containing them: [1986, 1986] $\sqcap$ [1998, 1998] = [1986-1998]. In the following, we will denote intervals containing a single integer with this integer: e.g. [1986, 1986] will be denoted as 1986. A similarity operator is associated to a subsumption relation $\sqsubseteq$ which allows to partially order the set of patterns D by specialisation in a hierarchy, as in a taxonomy:

$$a \sqsubseteq b \iff a \sqcap b = a, \qquad \forall a, b \in D$$

Please note that, contrary to standard notation, $a \sqsubseteq b$ is read "a is subsumed by b" in this context, and therefore a is more general than b. In our examples, *Object Oriented Language* $\sqsubseteq$ *C++*, and [1986-1998] $\sqsubseteq$ 1998. The taxonomy of patterns $(D, \sqcap)$ is a meet semi-lattice, i.e., a hierarchical structure in which each pair of elements possesses an upper-bound. A special element, denoted "*", represents a dissimilarity value to be able to specify that some patterns have no similarity. Figure 10 presents the taxonomy of patterns taken from the column `FirstRelease` of Table 4 and organised according to the interval similarity operator defined before. An arrow of the meet semi-lattice states that the source subsumes the target. The set of objects O, the taxonomy of patterns $(D, \sqcap)$ and the mapping $\delta : O \rightarrow D$ that associates each object $o \in O$ with a pattern $d \in D$ form a pattern structure. For instance, the set of products of Table 4, the taxonomy of patterns given at Figure 10, and the column `FirstRelease` of Table 4 associating each product to a value in Figure 10, form a pattern structure.

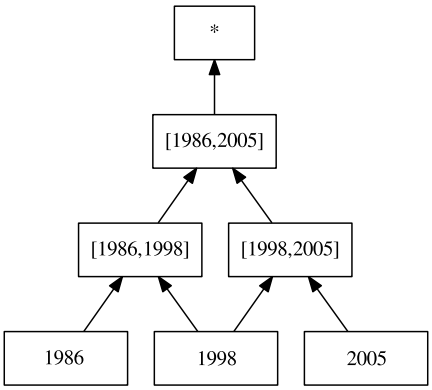


Figure 10: Automatically built taxonomy for the values of attribute `FirstRelease`

Given a pattern structure $PS = (O, (D, \sqcap), \delta)$, a set of *pattern concepts* can be extracted, where a pattern concept represents a maximal set of objects $O' \subseteq O$ described by the most specific pattern $d \in D$ characterising all the objects of O'. The set of all pattern concepts extracted from a pattern structure can be partially ordered by the specialisation relation $\leq_{ps}$ as follows: given two pattern concepts $C_1 = (O_1, d_1)$ and $C_2 = (O_2, d_2)$, $C_1 \leq_{ps} C_2$ if and only if $O_1 \subseteq O_2$ and $d_2 \sqsubseteq d_1$. For instance, in Table 4 ({`CVS`, `CVSNT`}, [1986-1998]) is a pattern concept because there is no other more specific pattern than [1986-1998] in Figure 10 that corresponds to both `CVS` and `CVSNT`, and there are no other products than these two that are first released between 1986 and 1998. The set of all pattern concepts of *PS* provided with the partial order $\leq_{ps}$ forms a *pattern concept lattice*. In a pattern concept lattice, the set of objects is structured depending on patterns of potentially complex type, and their similarities. Traditional FCA algorithms to extract variability information may be applied on pattern concept lattices in the same way as concept lattices. To the best of our knowledge, `Latviz` [3] is the only publicly available tool allowing to compute pattern structures which is for now restricted to intervals.

Patterns can be of atomic types (e.g., dates, numerical values, literals), but it is possible to combine several pattern types (from different pattern sets) in a *vector of patterns* [35]. A vector of patterns is of the form $\langle d_1, d_2, \ldots, d_n \rangle$, where $d_i, i \in \{1, 2, \ldots, n\}$ is a pattern of the compound taxonomy $(D_i, \sqcap_i)$. The similarity between two vectors of patterns (denoted as $\sqcap_{pv}$, where *pv* means *pattern vector*) can be obtained by computing the similarity/generalisation of patterns with the same rank in the vectors:

$$\langle d_{k1}, d_{k2}, ..., d_{kn}\rangle \sqcap_{pv} \langle d_{j1}, d_{j2}, ..., d_{jn}\rangle$$
$$= \langle d_{k1} \sqcap_1 d_{j1}, d_{k2} \sqcap_2 d_{j2}, ..., d_{kn} \sqcap_n d_{jn}\rangle.$$

Therefore, in this framework, a set of vectors of patterns can be handled in the same way as a set of patterns of atomic type. Detailed examples of pattern vectors and pattern concept lattices are given in the next section.

We estimate that pattern structures can broaden the scope of variability information that can be extracted from software variant descriptions. In fact, vectors of patterns and similarity operators are good candidates to depict complex software variant descriptions, composed of more than boolean features, as multi-valued attributes and cardinalities. As pattern structures rely on traditional FCA, all the variability information extracted with FCA can also be extracted with pattern structure generalisation, i.e., implications, mutex and co-occurrences between features. If the pattern structure represents multi-valued attributes in addition to boolean features, it may also extract augmented variability information. The grammar of the variability information that can be extracted using pattern structures is presented in Figure 11. As for traditional FCA, the extraction approach is sound and complete.

variability info.	:=	*relationship**
relationship	:=	*implication* | *co-occurrence* | *mutex* | *group*
implication	:=	*augmented element* '⇒' *augmented element*
co-occurrence	:=	*augmented element* '⇔' *augmented element*
mutex	:=	*augmented element* '⇒' '¬' *augmented element*
group	:=	'(' *feature* ',' '{' *feature_set* '}' ',' *cardinality* ')'
augmented element	:=	*feature* | *attribute*
feature_set	:=	*feature* | *feature_set* ',' *feature*
attribute	:=	**attribute_name** '=' **value**
feature	:=	**feature_name**
cardinality	:=	'⟨' **nb_min** '-' **nb_max** '⟩'

Figure 11: Grammar of the augmented variability information that can be extracted with pattern structures

In what follows, we propose a method to 1) compose vectors of patterns from multi-valued matrices representing complex software variant descriptions, and 2) extract complex variability information from them using pattern structures and FCA extraction algorithms.

4. Extracting Complex Variability

In this section, we present and illustrate our method for extracting complex variability information from multi-valued descriptions based on FCA and Pattern Structures. Section 4.1 presents the illustrative example. An overview of the method is given in Section 4.2. The Sections 4.3, 4.4 and 4.5 detail the three steps of the proposed method. Finally, Section 4.6 proposes a way to reduce the redundancy in the extracted information.

4.1. Illustrative example

We use *Product Comparison Matrices* (PCMs) [59] as software variant descriptions. PCMs are multi-valued matrices which depict a set of products depending on a set of characteristics, and allow a user to easily compare similar products from a same family. PCMs may gather in their cells heterogeneous data, including boolean attributes (usually "yes" and "no" values that can be considered as features) and multi-valued attributes. Thus, they are interesting candidates for the extraction of augmented variability information. PCMs may come from various sources, e.g., websites (as for instance Wikipedia), automatic generation, manually built by designers or developers. The main

drawback of using PCMs lies in the fact that they are not formalised: in most cases they need to be cleaned to be in a given format and to be automatically processed easily [59, 51]. For instance, Table 3 presents an excerpt of a Wikipedia PCM depicting 5 version control software systems depending on 6 characteristics.[7] We can see that the four first columns (from `ClientServer` to `Lock`) represent boolean attributes in different ways: the column `ClientServer` depicts only "yes" values, the column `Distributed` depicts "yes" and "no" values, and the two next columns display crosses. The column `FirstRelease` gives information understandable for a human, but difficult to process by a machine. Also, the value of this column for the element `ClearCase` does not give the good information. Finally, the last column `ProgrammingLanguage` can display several values in one cell, but the separators (e.g., ",", "/") and the way to write a same value (e.g., "only C", "c", "C") may differ. In order to ease the automated processing of this kind of PCM, we clean them by 1) harmonising the cells values (e.g., same value separator in each cell, same way to write a value to represent an information) and 2) replacing each missing/ambiguous value by the element "$*$". For instance, we choose here to represent boolean characteristic values by crosses. The multi-valued attribute `FirstRelease` should represent only the year of the first public release of the software system. We choose to separate values by semicolons to avoid issues with `.CSV` format. Table 4 presents the cleaned version of Table 3.

Table 3: Excerpt of a Wikipedia PCM about version control software systems

Software	ClientServer	Distributed	Merge	Lock	FirstRelease	ProgrammingLanguage
Git		yes	×		Started in April 2005	C / shell scripts / Perl
CVS	yes	no	×		First publicly released July 3, 1986	only C
ClearCase	yes	no	×	×	most recent version is 9.0	c, java and perl
GnuArch		yes	×		Initial release March 26, 2005	C and shell scripts
CVSNT	yes	no	×	×	First publicly released 1998	C++

Table 4: Cleaned version of the PCM from Table 3 following a chosen format. Values of `FirstRelease` are intervals of integer (representing years). We recall that intervals containing a single integer (e.g. [1986, 1986]) are denoted as this single integer (e.g. 1986)

Software	ClientServer	Distributed	Merge	Lock	FirstRelease	ProgrammingLanguage
Git		×	×		2005	C;shell scripts;perl
CVS	×		×		1986	C
ClearCase	×		×	×	*	C;java;perl
GnuArch		×	×		2005	C;shell scripts
CVSNT	×		×	×	1998	C++

Note that in this section, the term "attribute" has a different meaning than in FCA; to stay consistent with variability modelling and FM terminology, the term "attribute" here refers to the term used to name "multi-valued characteristics" (as the ones extending boolean FMs), and the term "features" indicates "boolean characteristics" (as the ones used in boolean FMs).

4.2. Method overview

We have defined a 3-step process to extract augmented variability information from multi-valued matrices as depicted in Figure 12. As input, it takes a PCM (variant descriptions), and as output, it gives a file documenting the extracted variability information in the form of logical relationships.

The first step of this process consists in defining the composition of the vectors of patterns, that will represent each software variant displayed in the PCM. This means that one has first to identify and define the sets of patterns $D_1, D_2, ..., D_n$ that will compose the pattern vectors. If the PCM is correctly cleaned so as to follow a given structured format, this step may be fully automated. Then, one has to define a similarity taxonomy for each identified composite pattern set of the vectors. Some taxonomies may be defined automatically depending on the type of patterns and the chosen similarity operator: this will be addressed next. Therefore, this step may be fully automated in some cases, and otherwise semi-automated. The second step is to build the pattern concept lattice based on the pattern vectors

[7] Original PCM: `https://en.wikipedia.org/wiki/Comparison_of_version_control_software`, last accessed in April 2018

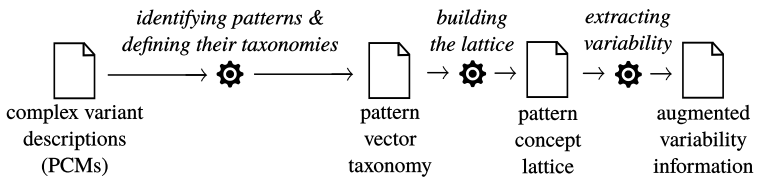


Figure 12: Process to extract augmented variability information from PCMs

and the defined taxonomies. It can be fully automated. The third and last step consists in extracting the variability information from the pattern concept lattice. In the following sections, we describe and illustrate each step based on the PCM example of Table 4.

4.3. Identifying patterns and defining their taxonomies

In this section, we assume that input PCMs are complete (no empty cells) and consistent (the same information is represented by the same attribute value). If it is not the case, the PCMs have to be manually cleaned to respect these requirements.

In what follows, we propose a way to build a pattern vector taxonomy denoted $(D_{pv}, \sqcap_{pv})$, structuring a PCM information by similarity and on which FCA can be applied. This step first consists in representing each product of the PCM by a vector of patterns corresponding to the product description. A PCM tabularly represents a collection of values of different characteristics for each product it documents. Thus, one can intuitively make the connection with vectors combining the characteristic values. However, in a pattern structure, the set of values of each characteristic needs to be organised in a taxonomy representing similarities between these values. Therefore, before being able to build the pattern vector taxonomy based on product descriptions, we have to define a taxonomy for the values of each characteristic of the input PCM. Values represented by a PCM may come from two types of data: features (PCM boolean characteristics) and multi-valued attributes (PCM multi-valued characteristics). We study these two types and define how to build taxonomies from them.

Note that boolean features may be seen and manipulated as multi-valued attributes with a $\{true, false\}$ value domain. We do not process features as attributes for two reasons. Firstly, features have a different significance compared to attributes in product line variability modelling, and thus it is important to maintain the distinction. Secondly, the boolean nature of features allows a more efficient and centralised process than multi-valued attributes, as we will show hereafter.

4.3.1. Processing boolean characteristics representing features

Defining a taxonomy for each boolean characteristic not only offers no new information, but also complexifies both the pattern vectors and the final lattice structure by adding useless taxonomies. Traditional FCA provides a solution to avoid redundancy and complexification of the processed data by building a canonical concept lattice in which the attributes are organised by specialisation, as in a taxonomy. Therefore, for each product of the PCM we consider the set of features that it owns (instead of each feature individually) in order to 1) simplify the pattern vectors and the final conceptual structure by considering feature sets as patterns, and 2) automatically create a (unique) taxonomy by means of FCA. Figure 13 (left-hand side) represents the concept lattice associated with the formal context formed by the first 4 columns of Table 4. Figure 13 (right-hand side) represents the taxonomy extracted from this lattice.

As a consequence, the set of features of a PCM will be represented by a pattern set denoted D_f. The similarity operator defined by the set-intersection $\cap$ allows to automatically define the taxonomy $(D_f, \cap)$.

4.3.2. Processing multi-valued characteristics representing attributes

To use the values $a_1, a_2, \ldots, a_n$ of an attribute a in a pattern vector, they have to be organised in a taxonomy, i.e., each value of a has to represent a pattern in D_a, and there must exist at least one similarity operator $\sqcap_a$ over D_a. It is unlikely that all values of the PCM attribute a represent all the necessary values to build the taxonomy $(D_a, \sqcap_a)$. For instance, if values a_1 and a_2 are present in the PCM, D_a contains both a_1 and a_2, but also $a_3 = a_1 \sqcap_a a_2$. However, a_3 may not be a value found in the PCM. In this case, it is thus necessary to complete D_a with the missing values.

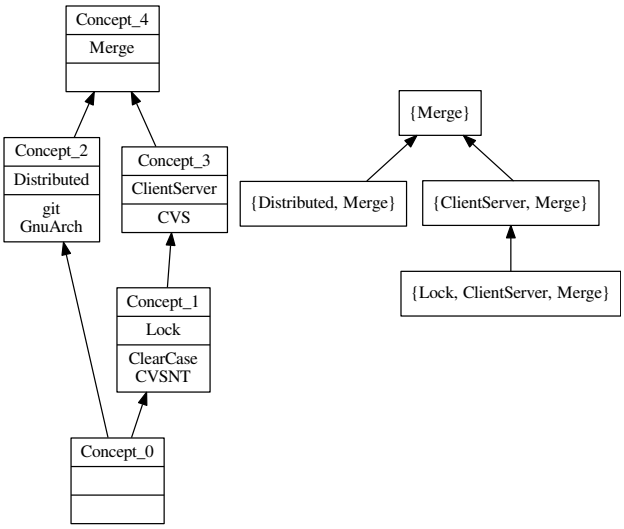


Figure 13: (left-hand side) concept lattice associated with the feature sets of Table 4; (right-hand side) induced taxonomy

Depending on the value type, and the chosen similarity operator, this completion may be automated. In fact, if a formula can be defined to compute the similarity of two values of $(D_a, \sqcap_a)$, the taxonomy construction may be automated from the values of a found in the PCM. This is the case for instance with intervals of integers; let us define the similarity of two intervals $\sqcap_{inter}$ by:

$$[a_1, b_1] \sqcap_{inter} [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)]$$

Figure 10 presents a taxonomy built automatically for the values of the attribute `FirstRelease` of Table 4 with the similarity operator $\sqcap_{inter}$. Note that if the taxonomy does not need to be built integrally, it may be done on demand.

When the attribute has literal values, it is more difficult to automatically complete the taxonomy. Several cases are possible for this kind of attributes. A multi-valued attribute of a PCM may give several values for a single product; this is the case for `ProgrammingLanguage`. If some of its values are shared by several products (e.g., the value `C` or `shell scripts` in our example), one can once again use FCA to automatically build a taxonomy based on the set-intersection of the set of values. To build this taxonomy, we consider that each literal value of the attribute is an FCA binary attribute, and we build the corresponding formal context. For the attribute `ProgrammingLanguage`, we obtain the context of Table 5. The corresponding concept lattice and the extracted taxonomy are presented in Figure 14.

	C	C++	shell scripts	perl	java
Git	×		×	×	
CVS	×				
ClearCase	×			×	×
GnuArch	×		×		
CVSNT		×			

Table 5: Formal context corresponding to the values of the attribute `ProgrammingLanguage`

In the case where the attribute values are unique for each product of the PCM, the automatically built taxonomy states that all elements are incomparable. Even if this solution allows the attribute to be processed by the framework, we loose the benefits of using pattern structures.

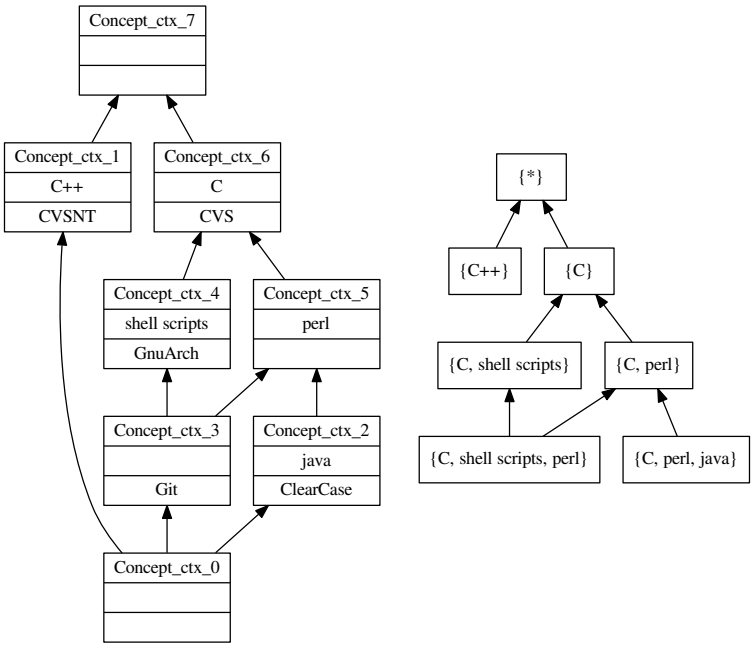


Figure 14: (left-hand side) Concept lattice associated with Table 5 and (right-hand side) induced taxonomy

In all cases, if an external taxonomy does not exist, an expert may build it manually. There are many ontologies that are available on the web, as the Protege Ontology Library,[8] that may be used as taxonomies.

4.3.3. Composing pattern vectors

In the following, we explain how the previously obtained taxonomies of feature set and multi-valued attributes are combined to form pattern vectors, and the pattern vector taxonomy.

Definition 4.1 (PCMs' pattern vectors).
Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of multi-valued attributes of a PCM, and $(D_1, \sqcap_1), (D_2, \sqcap_2), \ldots, (D_n, \sqcap_n)$ their associated taxonomies. Let $(D_f, \cap)$ be the taxonomy associated with the feature sets of the PCM. Then, each product of the PCM can be described by a pattern vector of the form:

$$\langle d_1, d_2, \ldots, d_n, d_f \rangle,$$
$$\text{with } d_i \in (D_i, \sqcap_i), \forall i \in \{1, 2, \ldots, n\}, \text{ and } d_f \in (D_f, \cap)$$

The PCM of Table 4 possesses 2 attributes and 4 features. Thus, the pattern vectors for the PCM of Table 4 are composed of 3 elements: the first element represents the value of the attribute `FirstRelease`, the second represents the value of the attribute `ProgrammingLanguage` and the third represents a subset of the 4 features. We selected similarity operators that can automatically build the taxonomies: $\sqcap_{inter}$ for `FirstRelease` and its values of type integer, and $\cap$ for `ProgrammingLanguage` and its literal values. The first two products are thus described by the following pattern vectors:

$$\text{Git} = \langle 2015, \{C, perl, shell\ scripts\}, \{Distributed, Merge\} \rangle$$
$$\text{CVS} = \langle 1986, \{C\}, \{ClientServer, Merge\} \rangle$$

We can now automatically define the similarity (or generalisation) of two **p**attern **v**ectors, denoted as $\sqcap_{pv}$:

[8] `http://protegewiki.stanford.edu/`

$$\text{Git} \sqcap_{pv} \text{CVS} =$$
$$\langle 2015 \sqcap_{inter} 1986, \{C, perl, shell\ script\} \cap \{C\}, \{Distributed, Merge\} \cap \{ClientServer, Merge\}\rangle =$$
$$\langle [1986, 2015], \{C\}, \{Merge\}\rangle$$

4.4. Building the pattern concept lattice

To the best of our knowledge, there is no publicly available tools allowing to build pattern concept lattices based on pattern structures representing taxonomies of pattern vectors (*pattern vector structures* for short). However, it is possible to use the tools defined for traditional FCA in our case thanks to a method called *binary scaling*. It generally consists in the transformation of a multi-valued context (e.g., a PCM) in a formal context in order to be processed by means of FCA. For this, characteristic values produce a binary attribute in the output formal context. In our case, we do not seek to apply binary scaling on a multi-valued matrix, but on a pattern vector structure. We define binary scaling on taxonomies of pattern vectors:

Definition 4.2 (Binary scaling).
Let $P_s = (O, (D_{pv}, \sqcap_{pv}), \delta)$*be a vector pattern structure, where vectors of* $(D_{pv}, \sqcap_{pv})$ *are composed of values of n taxonomies* $\{(D_1, \sqcap_1), (D_2, \sqcap_2), \ldots, (D_n, \sqcap_n)\}$.

The binary scaling of P_s *produces a formal context* $K = (O_s, A_s, J_s)$ *such that:*

- $O_s = O$
- $\mathcal{A}_s = \bigcup_{i=1}^{n} D_i$
- $\mathcal{J}_s = \{(o, d_1) | \exists d_2 \in \delta(o), d_1, d_2 \in D_i, i \in \{1, 2, \ldots, n\}, d_1 \sqsubseteq_i d_2\}$

In other words, each value of each pattern taxonomy will correspond to a binary attribute of the produced formal context. The traditional FCA algorithms presented in Section 3 to build concept lattices and AC-poset are thus applicable as they are for pattern structures.

Figure 15 presents the pattern AC-poset associated with the formal context obtained after the binary scaling of the pattern vector structure built in the previous subsection.

4.5. Extracting complex variability information

As for traditional FCA, the pattern AC-poset, as well as the pattern concept lattice, allows to extract several types of variability information in the form of logical relationships. Among them, we find the 4 types of logical relationships necessary to represent *augmented variability information*. Here again, the traditional FCA algorithms to extract variability information are applicable thanks to the binary scaling.

In what follows, we outline how to read these types of relationships in pattern concept lattices and pattern AC-posets.

Binary implications: As each concept inherits all patterns of its super-concepts, implications can be extracted between patterns (i.e., features and/or attribute values) from the partial order given by the structure. Following the arrows of Figure 15, we can see that attribute value `FirstRelease:[1986-1998]` (*Concept_5*) implies the feature `ClientServer` (*Concept_8*). This can be interpreted by "all software variants first released between 1986 and 1998 support a client-server repository model". We still observe "boolean" implications: for instance, *Concept_4* and *Concept_8* bear the implication `Lock` ⇒ `ClientServer`.

Co-occurrences: When several patterns are introduced in the same concept, that means that they always occur together in all the variants. Thus, co-occurring patterns can be extracted. For instance, *Concept_6* highlights the following relationship: `Distributed` ⇔ `ProgrammingLanguage:shell scripts`. This can be interpreted by "all software variants supporting a distributed repository model use shell scripts and conversely".

Mutex: When the extents of two concepts introducing patterns have an empty intersection, it means that these two patterns are never present together in any software variant. In other words, they are mutually exclusive in the considered set. In Figure 15, *Concept_2* and *Concept_6*, respectively introducing `ProgrammingLanguage:java` and

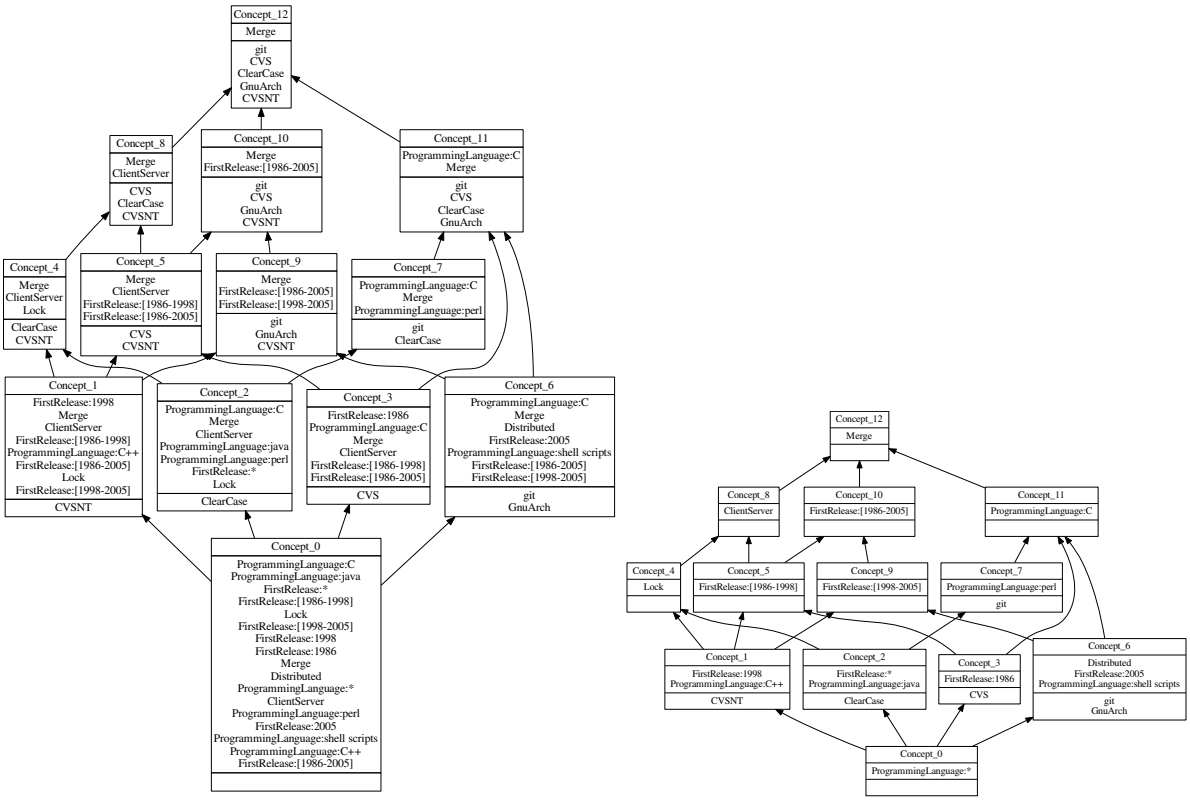


Figure 15: Pattern AC-poset structuring the software variants from the PCM of Table 4

`ProgrammingLanguage:shell scripts`, do not have any variant in common in their extent. Thus, `ProgrammingLanguage:java` $\Rightarrow \neg$ `ProgrammingLanguage:shell scripts`. This can be interpreted by "none of the software variants developed in Java uses shell scripts, and conversely".

Feature-groups: A group has a parent-feature, and it expresses the fact that if the parent-feature is present in a software variant, then at least one of the features from the group is also present in this software variant. This information can be extracted from traditional conceptual structures without using pattern structures, as it concerns only features [58, 15]. The precise group cardinality may be extracted by analysing the descriptions of the software variants present in the extension of the concept introducing the parent feature of the group.

It is noteworthy that other types of relationships that may also represent variability information can also be found, i.e., non-binary implications [58] and groups of exclusive features (*not all together*) [15]. But they do not correspond to the ones found in extended feature models, so they are not studied in this paper. However, they may be useful for instance in recommendation systems, or other types of variability models.

4.6. Redundancy elimination

Pattern structures and pattern taxonomies allows to eliminate some redundancy in the extracted relationships. We propose heuristics to reduce the number of extracted relationships without loosing information.

Binary implications between two values of the same attribute are not taken into account, as they provide the same information as the attribute taxonomy. For instance, the binary implication `FirstRelease:1986` $\Rightarrow$ `FirstRelease:[1986-1998]` (*Concept_3* and *Concept_5* in Figure 15) is removed. Also if the same element in premise implies different values of the same attribute, only the implications with the most specific conclusions are kept. In fact, the implications involving more general values may be inferred from them using the taxonomies. For instance, we have the two implications `ProgrammingLanguage:C++` $\Rightarrow$ `FirstRelease:1998` (*Concept_1*) and `ProgrammingLanguage:C++` $\Rightarrow$ `FirstRelease:[1986-1998]` (*Concept_1* and *Concept_5*). We only keep the first one,

i.e., the most specific one, as the second implication may be inferred from the first. Even though the specific implication may also be inferred from the more general one, we may also infer incorrect implications, as for instance `ProgrammingLanguage:C++` ⇒ `FirstRelease:1986`. This is due to the fact that the premise may imply a pattern but not all its more specific patterns in the taxonomy.

Mutex may also present redundancies. Mutual exclusions between two values of the same attribute do not provide any additional information because they may be found in the attribute taxonomy. Similarly to implications, if the same element is mutually exclusive with different values of the same attribute, we may remove some of them that may be inferred from the ones we kept. This time, it is the most general values that are kept. This is due to the fact that if an element is mutually exclusive with a pattern, it is also mutually exclusive with all its sub-patterns of the taxonomy. For instance, the feature `Distributed` is mutually exclusive with attribute values `FirstRelease:1998`, `FirstRelease:1986` and `FirstRelease: [1986-1998]`, so we only keep `Distributed` ⇒ ¬ `FirstRelease:[1986- 1998]`.

We can avoid redundancy in co-occurrences by removing the ones that can be inferred from at least two other co-occurrences. For instance, we can keep from *Concept_6* `Distributed` ⇔ `FirstRelease:2005`, `ProgrammingLanguage:shell script` ⇔ `Distributed` and remove `ProgrammingLanguage:shell script` ⇔ `FirstRelease:2005` Finally, for all types of relationships, we never extract relationships with a frequency equals to 0, i.e., true for the considered set of variants but which never actually occur in any variant.

5. Evaluation

In what follows, we evaluate our method through three research questions assessing both its usability and usefulness:

RQ1 (applicability). Is this approach technically applicable on existing available datasets? FCA and Pattern Structures are well known to exponentially grow with the size of the input data. To answer this question, we evaluate the size of the pattern conceptual structures (AC-poset and concept lattice) obtained when applying the method exposed in Section 4 on a selection of existing datasets representing descriptions of software variants. We define the size of pattern conceptual structures by their numbers of pattern concepts and edges representing the partial order. If these structures are too large to be easily computed, stored and managed, then our method will face some difficulties regarding applicability.

RQ2 (order of magnitude). What is the order of magnitude of the number of extracted relationships with our method? Is the redundancy elimination efficient? FCA-based relationship extraction is sound and complete; the relationships that are true for the considered input dataset may be quite numerous, and even more when considering attribute value taxonomies. Thus, we seek to obtain an order of magnitude of the number of relationships extracted from selected available datasets, and evaluate how this number is reduced with the proposed redundancy elimination. We analyse the characteristics of the dataset that may influence these orders of magnitude.

RQ3 (pertinence). Is the resulting variability information consistent? All information that is consistent for a domain expert will be present in the result of this method, modulo the correctness of the product descriptions. Yet, a certain amount of extracted variability information may be "accidental" and not pertinent, i.e., true for the considered dataset but not for the domain. This is generally due to the fact that datasets are not representative of all the software variants, as they only show a subset of possible software descriptions. Thus, to answer this question, we evaluate which percentage of extracted variability information is pertinent, and which percentage represents "accidental" ones.

5.1. Data

To answer the previous questions, we selected three existing available datasets representing descriptions of families of software variants, which have already been studied in the software product line literature: PCMs of Wikipedia [59], `Robocode` [46] and `JHipster` [29]. The goal of our selection was to work on disparate datasets to broaden the scope of our evaluation. The selected datasets are of variable sizes (from 8 to 2000 products) and possess a heterogeneous distribution of multi-valued attribute types. Each software family of these datasets is presented in the form of a PCM

in a `.CSV` file. Some of these PCMs are complete and correct, others suffer from missing and/or possible incorrect values. The cleaned versions[9] used in this evaluation are detailed in what follows.

Product Comparison Matrices of Wikipedia [59]. We first worked on 30 PCMs about software systems, taken from the software comparison category of Wikipedia.[10] These PCMs have been extracted and processed using the Java API of `OpenCompare`,[11] a dedicated tool to extract, edit, exploit and export PCMs from several websites (including Wikipedia). Before performing any automated processing, we manually cleaned each PCM as explained in Section 4.1.

Table 6 gathers information about these PCMs that can be considered system families of small sizes (23 products on average), and with potentially incomplete and/or incorrect values, as they do not follow any format and can be manually edited by any user.

Table 6: Information about the 30 PCMs from Wikipedia used for the evaluation

	minimum	maximum	mean
#products	8	75	23
#boolean characteristics	0	17	5
#literal characteristics	0	4	2
#numerical characteristics	0	4	1
#cells	36	1650	212

Robocode [46]. `Robocode`[12] is a programming game where developers have to implement the behaviour of their own bots in order to fight bots implemented by other developers. Each developer develops their bot variants on top of an API which provides a basic behaviour. Commonalities can be found in each variant, e.g., in the implementation choices or the adopted fighting strategies. The `Robocode` game is thus a variability-rich family of software systems which is considered a pertinent dataset to study variability extraction. Tabular descriptions of existing bot variants have been gathered online.[13] In total, 306 bot variants are described depending on 4 multi-valued attributes (documenting the bot movement and targeting strategies, the fighting type they are designed for, and the license of the source code), and a boolean feature stating if the source code is available. It is a PCM of medium size with incomplete values. Incorrect values may be present but should be rarer than in PCMs of Wikipedia, as the information is extracted from a wiki completed by the developers themselves.

JHipster [29]. `JHipster`[14] is a web application generator, that questions a developer concerning its technology preferences (e.g., database type, testing framework) to build a functional variant corresponding to its choices. In [29], the authors analyse the `JHipster` questionnaire to deduce the descriptions of all variants that may be generated; these descriptions are available on a Github repository.[15] More than 26000 variants have been documented against 7 boolean features and 17 multi-valued attributes; it is thus considered a large size software family. Also, as the descriptions where generated automatically from existing variants, they are complete and correct. To test if our method scales on large datasets, we split these descriptions in 3 PCMs of 500, 1000 and 2000 variant descriptions.

5.2. Methodology

We have developed a tool in Java called `CLEF`[16] (for Complex variabiLity Extraction with Fca) to perform all the steps that may be automated to answer these research questions. It is composed of 3 packages:

[9] `https://github.com/jcarbonnel/CLEF/tree/master/CLEF/data`
[10] `https://en.wikipedia.org/wiki/Category:Software_comparisons`, last accessed in April 2018
[11] `https://github.com/OpenCompare`
[12] `https://robocode.sourceforge.io/`
[13] `https://github.com/but4reuse/RobocodeSPL_teaching`
[14] `https://www.jhipster.tech/`
[15] `https://github.com/xdevroey/jhipster-dataset/tree/master/v3.6.1`
[16] `https://github.com/jcarbonnel/CLEF`

- `multivaluedcontext`, which contains classes to handle multi-valued product descriptions. The class `MultivaluedContext` contains a list of objects (i.e., the products) and a list of `Characteristics`. A concrete characteristic can be a `BinaryAttribute` (feature) or a `MultivaluedAttribute`. A multi-valued context imports descriptions from a `.CSV` file and identifies automatically boolean features and multi-valued attributes, along with their types (literal, integer or double).

- `similarities`, which contains classes to build taxonomies from multi-valued attributes. We implemented the processes for building taxonomies presented in Section 4.3, which extend `AbstractSimilarities`. If one wants to use another process to build a taxonomy, it is easy to add new classes to implement it.

- `relationshipextraction`, which contains classes implementing the methodology presented in Section 4.5 to extract binary implications, co-occurrences and mutex from the AC-poset. The implemented algorithms are the ones previously mentioned in Section 3 that can be found in [15] and [2].

As there is no publicly available tools to compute pattern concept lattices and pattern AC-posets yet, we applied binary scaling on the datasets and used `RCAExplore`, an FCA tool that computes conceptual structures from formal contexts (binary matrices). We implemented the binary scaling presented in Def. 4.2 in `CLEF` to obtain a formal context in `.RCFT` format, that can be processed by `RCAExplore`. For this, each attribute of the matrix is associated with a taxonomy corresponding to its type and that is built automatically. In our evaluation, attributes having numerical values are associated with the $\sqcap_{inter}$ similarity operator that produces intervals of values, and attributes having literal values are associated with the $\cap$ similarity operator. It is important to note that the FCA structures associated with the obtained scaled formal contexts allow to extract the same information as in a pattern conceptual structure, and contain the same number of formal concepts and edges.

RQ1. Using `CLEF`, we applied the two first steps of Section 4 on the selected datasets to obtain an order of magnitude of the size of their associated pattern conceptual structures. Once a `MultivaluedContext` is initialised from a `.CSV` file, the method `computeLattice()` applies the binary scaling, saves the obtained formal context in an `.RCFT` file, and calls `RCAExplore` to compute the associated AC-poset. The files representing the resulting conceptual structures are stored in `data/dataset_name/FCA/`, and can be analysed to obtain their size.

RQ2. We applied the third step of Section 4 by using our tool to automatically extract the logical relationships representing augmented variability information from the previously obtained AC-posets. As feature groups involve exclusively features, and their extraction by means of FCA has been already assessed [58, 15], we do not study them in this evaluation, in order to focus on augmented variability extraction. Implemented extraction algorithms have been optimised to extract the relationships without the redundancy identified in Section 4.6.

Table 7 presents an excerpt of extracted logical relationships from the `Robocode` PCM.

Table 7: Examples of extracted relationships from the `Robocode` PCM

Examples of binary implications:
* targeting: {reduced linear targeting} ⇒ fighting: {melee}
* movement: {provocative movement} ⇒ targeting: {circular targeting, segmented mean}
* movement: {musashi trick} ⇒ fighting: {one on one}
Example of co-occurrence:
* movement: {stationary} ⇔ targeting: {fire at enemys bullet}
Examples of mutex:
* fighting: {melee} ⇒ ¬ movement: {aggressive movement}
* fighting: {one on one} ⇒ ¬ targeting: {corner targeting}
* targeting: {dynamic clustering} ⇒ ¬ movement: {danger prediction}

RQ3. To evaluate the percentage of extracted accidental relationships, we seek to detect the ones involving elements that have no influence on each other and thus representing relationships that are true for the considered set of products but meaningless regarding the domain. To do so, we selected 3 PCMs and we defined a "graph of influence" between their characteristics, where an edge between two characteristics states that we considered that they influence each other. We selected PCMs having characteristics that we are sure to fully understand to ensure the validity of the graph. For instance, we considered that an attribute representing the latest stable release and another attribute representing the programming language have no influence on each other. A contrario, we considered that an attribute representing the database backend and another representing the server operating system may influence each other. However, other experts may obtain slightly different graphs, or choose to ignore the influence between certain characteristics they are not interested in. The different point of views between experts may change the results of this study. So, to limit this threat to validity, the graphs used in this evaluation have been validated by three domain experts.

We selected two Wikipedia PCMs, one about accounting softwares (`Comparison_of_accounting_software_2`) and the other one about CRM systems (`Comparison_of_CRM_systems_0`). These two graphs are presented in Figure 16. We also selected the `Robocode` PCM; its graph of influence is represented in Figure 17

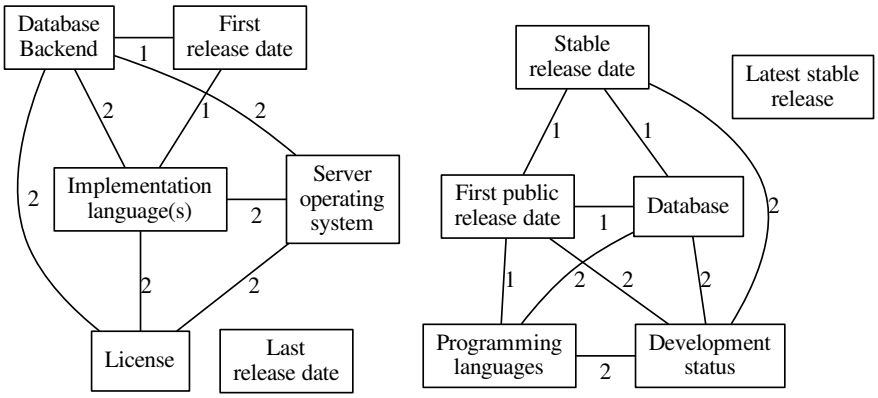


Figure 16: Graph of influence of the two selected PCMs of Wikipedia

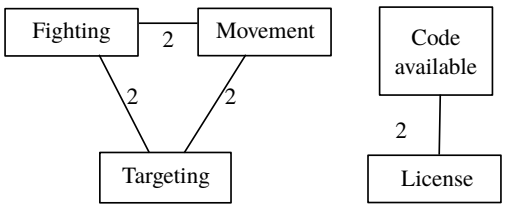
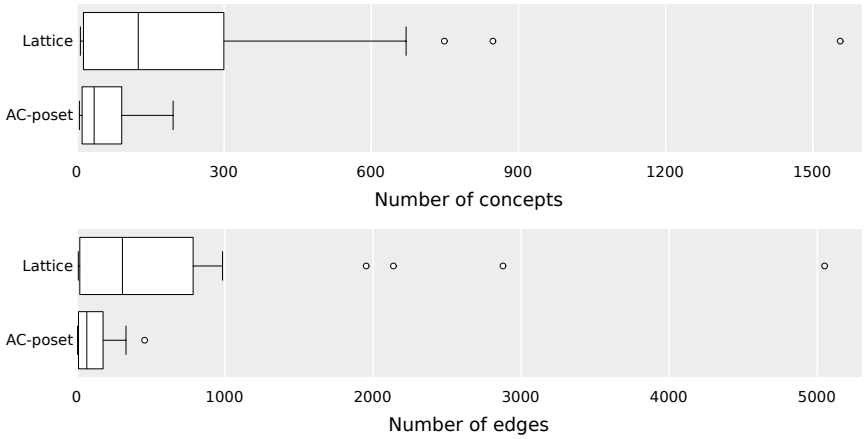


Figure 17: Graph of influence of the `Robocode` PCM

Then, we automatically separate the set of relationships in two groups depending on the graphs: the pertinent relationships and the accidental ones. We consider the set of reduced relationships obtained with the aforementioned heuristics on redundancy elimination. When the groups have been established, we can finally compute the percentage of accidental relationships obtained with our method.

5.3. Result analysis

RQ1. The sizes of pattern concept lattices and pattern AC-posets associated with the 30 PCMs of Wikipedia are presented in Figure 18. It shows the dispersion indicators (minimum (min), maximum (max), first, second and third quartile (q1, q2 and q3) and the mean) of both the number of concepts (# Concepts) and the number of edges (# Edges).



	min	q1	q2	mean	q3	max
# Concepts						
Lattices	7	13	125	243	300	1556
AC-posets	5	11	35	54	91	196
# Edges						
Lattices	9	19	307	707	785	5049
AC-posets	4	11	67	109	176	457

Figure 18: Size of pattern concept lattices and AC-posets for the 30 studied PCMs of Wikipedia

Despite that the binary scaling of each element of the taxonomies may produce large formal contexts, the pattern conceptual structures of each Wikipedia's PCM are easily computable, storable and manageable. The computation of conceptual structures using `RCAExplore` on a computer *Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 16GiB RAM* was instantaneous. Output `.DOT` files [23] storing the conceptual structures are manually browsable (i.e., they are not too convoluted), and their size did not limit their processing during the following experiments. It is noteworthy that the Wikipedia PCMs we have selected to conduct this experiment can be considered as large sized PCMs. In fact, based on the results of Sannier et al. in [59] who have performed quantitative analysis over 165 PCMs from Wikipedia, on average, a PCM possesses 178,5 cells. Table 6 shows that most of our selected PCMs are larger than the average ones in terms of number of cells, with an average of 212. Our method is thus technically applicable on Wikipedia's PCMs.

Table 8 shows the size of the conceptual structures associated with the PCM of `Robocode` and PCMs representing excerpts of 500, 1000 and 2000 products of the `JHipster` dataset.

Table 8: Size of the conceptual structures for `Robocode` and excerpts of `JHipster`

#Concepts:	Lattice	AC-poset
`Robocode`	827	183
`JHipster 500`	65K	295
`JHipster 1000`	141K	548
`JHipster 2000`	269K	1048
#Edges:	**Lattice**	**AC-poset**
`Robocode`	2397	403
`JHipster 500`	352K	2535
`JHipster 1000`	827K	4984
`JHipster 2000`	1.6M	9949

We clearly see in these larger datasets the gain of the use of AC-posets in terms of size of the structures to manage. Even though the number of products in `Robocode` is 4 time larger than the largest Wikipedia's PCM, the size of their associated conceptual structures is quite similar. This is due to the fact that `Robocode` has a few number of attributes/features (5 in total). In comparison, conceptual structures associated with the excerpts of `JHipster` have a very large size: 269727 concepts for the largest concept lattice. This is due to the large number of characteristics documenting the products. However, the AC-posets remain easily computable, and are built by `RCAExplore` in a few seconds. Their processing by `CLEF` to extract variability information is also reasonable, as it takes 4 seconds in the worst case to analyse the AC-poset and to compute all variability relationships (without redundant ones). It appeared during the experimentations that the implemented process to build the taxonomies was the most time consuming. Computing integer taxonomy when there are a lot of different initial values results in a colossal number of computed interval values. Using a threshold to limit the number of handled intervals seems to be a good compromise to avoid slowing down the overall process; this is left as future work. To sum up, even with product descriptions of consequent sizes, using AC-posets to support complex variability extraction is applicable.

RQ2. The dispersion indicators of the numbers of logical relationships extracted from each PCM of Wikipedia are listed in Table 9 depending on the type of relationships, and if they are reduced (i.e., after eliminating the redundancy) or not. Binary implications are analysed regarding both their transitive closure (TC) and their transitive reduction (TR). Figure 19 presents the comparison of the average number of mutex and implications before and after redundancy elimination.

Table 9: Dispersion indicators for the number of extracted relationships of PCMs of Wikipedia

	min	q1	q2	mean	q3	max
Mutex	0	24	300	1126	2121	5967
Red. Mutex	0	5	50	227	250	1615
Impl. (TC)	6	22	365	1054	1503	6050
Red. Impl. (TC)	5	13	69	122	167	734
Impl. (TR)	4	13	105	149	213	563
Red. Impl. (TR)	4	10	39	76	95	388
Co-occurrences	0	0	1	9	9	98

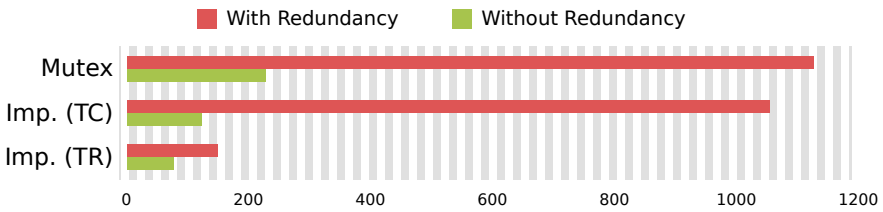


Figure 19: Comparison of the means of reduced and not reduced extracted implications and mutex

Table 10 presents the extracted complex logical relationships for `Robocode` and the 3 excerpts of `JHipster` (denoted `JH. 500`, `JH. 1000` and `JH. 2000`).

For PCMs of Wikipedia, the redundancy elimination allows to drastically reduce the number of extracted relationships: on average, 20% of the mutex, 12% of the binary implication transitive closure and 51% of the binary implication transitive reduction are retained. This is due to the fact that a product in PCMs of Wikipedia is often described by numerical attributes and/or several values for a same literal attribute. Therefore, the redundancy elimination using taxonomies built on attribute values is efficient in this case.

Concerning the `Robocode` PCM, the redundancy elimination is useful: about 19% of the mutex, 31% of the binary implication transitive closure and 57% of the binary implication transitive reduction are retained. `Robocode` does not possess any numerical attribute, but the presence of several values for a given literal attribute and a given product

Table 10: Size of the conceptual structures for Robocode and the 3 excerpts of JHipster

	Robocode	JH. 500	JH. 1000	JH. 2000
Mutex	17663	40965	144532	538668
Red. Mutex	3457	9802	19738	39124
Impl. (TC)	887	5399	10303	20214
Red. Impl. (TC)	277	5389	10303	20214
Impl. (TR)	487	2794	5383	10663
Red. Impl. (TR)	276	2789	5383	10663
Co-occurrences	11	26	23	23

allows to extract relationships between groups of values and thus to reduce their number without losing information. For instance, the two following binary implications can be extracted from the Robocode PCM:

$$\textit{movement} : \textit{perpendicular} \Rightarrow \textit{targeting} : \textit{head on targeting}$$
$$\textit{movement} : \textit{perpendicular} \Rightarrow \textit{targeting} : \textit{linear targeting}$$

Thanks to the built taxonomy on the attribute targeting, the binary implication

$$\textit{movement} : \textit{perpendicular} \Rightarrow \textit{targeting} : \{\textit{head on targeting}, \textit{linear targeting}\}$$

is also extracted and can replace the two previous ones.

Redundancy elimination for mutex relationships in the 3 JHpister PCMs allows to retain 24%, 14% and 7% of the relationships. We notice that, in this case, the redundancy elimination does not reduce the extracted binary implications. This is due to the fact that these datasets do not possess any numerical attribute, and that each product has exactly one value per literal attribute. In this particular case, literal attribute values cannot be grouped to form relationships between more general values that can replace others. Moreover, as each product is associated to exactly one value per attribute, there is no binary implication between values of the same attribute. For these two reasons, redundancy elimination based on taxonomies does not work for binary implications in these datasets. However, it is useful for mutex relationships as, here again, values of the same literal attribute never appear together in any product of the JHipster PCMs and thus produce a large number of mutex. As we do not keep mutex between values of the same attribute, redundancy elimination highly reduces their number.

RQ3. The percentages of pertinent extracted relationships from the 3 selected PCMs are presented in Figure 20. We only considered the transitive closure of the implication set, as the pertinence of the implications is assessed on each implication separately, and not by considering the whole set.

An example of accidental extracted relationship from the Wikipedia PCM about accounting softwares is: latest stable release:4.07.10 ⇒ programming language:php. In fact, we considered in the graph of influence of Figure 16 that the attribute representing the latest stable release and the attribute representing the programming language had no influence on each other.

An overall observation shows that relationships annotated as "not pertinent" represent a significant part of extracted relationships. About 65% of extracted relationships from the PCM about accounting softwares, and 33% in the PCM about CRM systems are annotated as not pertinent. The large part of accidental relationships is due to the fact that there are not enough variant descriptions in a dataset to document all possible interactions between the dataset characteristics. This is not only the case for PCMs of Wikipedia: about 39% of relationships extracted from the Robocode PCM are considered not pertinent. This denotes the necessity to complete the proposed approach with tools and methods to detect pertinent relationships among the extracted ones.

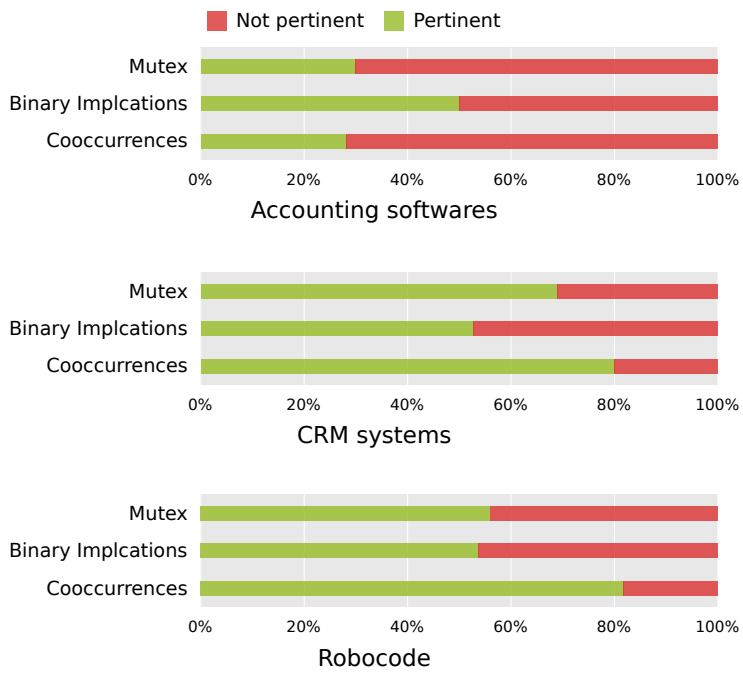


Figure 20: Percentages of pertinent extracted relationships from the 3 selected PCMs according to the graphs of influence of Figure 16 and Figure 17

5.4. Threats to validity

An external threat to validity is that our quantitative and qualitative evaluations are based on only 3 selected datasets, that may be not representative enough of product descriptions in general. The fact that each PCM has to be manually cleaned before applying our method and performing the experimentations has limited the number of studied PCMs. However, recent research has been conducted on automated extraction and analysis of PCMs [51, 50]. This kind of work could ease the process to obtain cleaned PCMs and allow to make our evaluation on a more significant number of PCMs. Also, PCMs from Wikipedia could be not representative enough of the ones one can find elsewhere, and thus PCMs from other websites need to be evaluated. This is left as future work.

Moreover, a construct threat lies on the heuristic we used to detect "accidental" relationships among the extracted ones. It is possible that some of the relationships considered consistent may be not useful to model the domain variability, or that some of them may be more valuable than others. This clearly denotes the necessity to deepen the analysis of extracted variability relationships to complement our approach and, in the end, to develop efficient tools to assist practitioners in variability modelling from product descriptions.

A last construct threat is the validity of the developed algorithms used to implement our evaluations. To limit this threat, we manually reviewed the results of each step, first on small examples, and then on some of the selected PCMs.

6. Related Work

Lots of works can be found in the literature about extraction of variability information from product descriptions (not necessarily software systems), mostly to synthesise basic feature models as introduced in the FODA report [34]. Basic feature models (also called boolean feature models) graphically represent variability of a product family by organising features in a hierarchy, and adding information as feature groups, mutex, or feature implications. Building a feature model from product descriptions is a way to represent variability information characterising the initial set of product variants.

Boolean feature model synthesis and feature constraint discovery. Haslinger et al. propose in [30] a dedicated algorithm to build a feature model from a collection of feature sets (i.e., representing valid configurations), the latter being presented in a similar form to formal contexts. They extend their method in [31] to be able to add cross-tree constraints to their synthesised feature tree. In [1], Acher et al. propose their own dedicated algorithm to extract feature models. Their approach and ours have in common the fact that they start from PCMs. The BUT4Reuse tool [48], which has been proposed to foster bottom up SPL adoption, hosts two approaches that focus on feature constraint discovery and manipulation: (1) From the graph decomposition of several model variants (including cardinalities) and the way an identified feature spans in this graph decomposition, the MoVa2PL approach [47] extracts model element dependencies as well as Requires and Mutual exclusion constraints between features; (2) A visualisation in concentric zones of the way a feature is connected to the other features in terms of hard (requires, excludes) or soft (encourages, discourages) constraints is proposed and evaluated in the FRoGs approach [49]. Davril et al. [22] address the problem of feature model synthesis when the set of product variants is not formally documented. They first mine relevant features from informal documents and produce a product-by-feature matrix close to PCMs. They extract all valid implications and use text-mining techniques and co-occurrences between features to help identify meaningful implications to build a relevant model. Text-based techniques could be integrated in a future approach to detect relevant variability relationships to complement our method.

Search-based techniques are used to extract feature models from product descriptions, such as in [42] where genetic programming is used to synthesise a population of feature models representing the initial collection of feature sets, or in [44] where two objective functions evaluate an evolutionary algorithm, hill climbing and a random search to synthesise feature models. These search-based approaches produce FMs representing feature sets close to the input ones, but do not assess meaningfulness of the represented variability information. In comparison, extracting meaningful variability information from descriptions by relying on knowledge discovery techniques is the main goal of our work.

FCA and its associated conceptual structures have also been used in this domain. Ryssel et al. [58] propose an approach to extract variability information from formal contexts, which strongly inspired our algorithms of variability extraction from concept lattices and AOC-posets. Loesch and Ploedereder [43] use FCA to extract variability information from the set of valid configurations of a feature model to help its restructuring. Finally, Al-Msie'Deen et al. [2] propose algorithms to extract different kinds of feature groups from AOC-posets constructed from software variant configurations. The authors seek to extract logical relationships rather than focusing on their ontological semantics. In previous work [17] we deepened the part played by FCA in boolean variability representation. For this, we studied how boolean variability information is represented in FCA structures and boolean FMs, and we established a matching between the logical relationships expressed by these two formalisms. We regrouped existing approaches of the literature which focus on extracting the logical semantics of boolean FMs from FCA structures, and we formulated a sound and complete extraction method. We showed how to use this method along with FCA structures to extract a diagrammatic representation of equivalence classes of boolean FMs to help designers in boolean FM synthesis. In this paper, we use the results of our previous work as a basis to extend the extraction method to complex variability information by using an FCA extension called Pattern Structures. Here, we show how FCA extensions enable to take into account structured datasets (ontology/taxonomy) to deal with complex variability information, and how it can be used to reduce the number of extracted relationships.

All these works have in common the fact that they extract variability in the form of boolean feature models or feature constraints, and from product descriptions that only document binary features. In comparison, our approach also takes into account multi-valued attributes and allows to extract more complex variability information e.g., implications, co-occurrences and mutex between a feature and an attribute value or between two attribute values. However, we do not propose a method to construct feature models from the extracted variability information. But the extracted variability information, in the form of logical relationships, is independent from the final graphical representation, and can assist the synthesis of feature models, as well as models in other formalisms (e.g., OVMs, CVL models).

Another approach to extract variability uses natural language processing. For example, by analysing product descriptions in the form of commercial texts, [24] identifies candidate features and classifies them into two categories: commonalities and variabilities. The features are then entered into a graphical tool and a feature model can be manually built. In our approach, the found relationships between features are more precise than just commonalities and variabilities, but on the other hand we do not focus on feature identification and the input of our approach is not

text but structured product descriptions.

Complex variability extraction. To our knowledge, the work of Becan et al. [8] is the closest to ours, and the only one assessing extraction of variability information involving both features and attributes. They propose dedicated algorithms to extract attributed feature models which are an extension of boolean feature models where features can possess valued attributes. They start from configuration matrices, just as ours. For this, they first compute feature groups, implications between features and mutually exclusive features. Then, they extract all possible complex implications involving at least an attribute value. In our approach, we propose, in addition, to compute mutex between features and/or attribute values, and to document co-occurrences between features and/or attributes. Besides, whereas we extract logical relationships from a canonical conceptual structure, they rely on several different structures (binary implication graph, mutex-graph). We do not address yet the problem of synthesising a complex variability model as attributed feature models; we prefer to focus on extracting logical relationships independently from any graphical representation, to later be able to build different kinds of variability models. The novelty of our approach is two-fold. First, using FCA and Pattern Structures gives a unique mathematical framework to formalise extraction algorithms of the different logical relationships depicting variability. Then, our approach is generic, as Pattern Structures allow to take into account any kind of data on which a similarity operator can be defined. It would be interesting to experimentally compare the obtained relationships of the same nature with the two approaches (from Becan et al., and ours), as well as the corresponding computation time; this is left as future work.

Pattern Structures. Pattern Structures have been used for knowledge discovery on complex types of datasets. Ganter and Kuznetsov first use Pattern Structures to organise graphs representing molecules [26]. In [56], Reynaud et al. use FCA and Pattern Structures for RDF triples classification in DBpedia datasets. The authors show how to process the obtained structures to discover relevant information. In the same domain, Barbant et al. [13] use Pattern Structures to represent ontologies and discover knowledge in the web of data. Kaytoue et al. [35] use pattern structures to mine gene expression data. They use vectors of patterns composed of intervals. In comparison, we used pattern vectors that may be composed of different types of patterns and not only intervals. In [14], Buzmakov et al. mine meaningful sequential patterns from sequential data using FCA and Pattern Structures. Baixeries et al. [6] use Pattern Structures to compute functional dependencies from databases, and show how Pattern Structures can characterise other types of dependencies as degenerated multivalued dependencies. In another work [5], the authors propose a way to compute similarity dependencies between the data values. These works can help to filter meaningful relationships, or to define new relationships to extract that can be useful for practitioners in the field of variability analysis and representation. To the best of our knowledge, `Latviz` [3] is the only publicly available tool allowing to define pattern structures and it is limited to intervals.

7. Conclusion

In this paper, we studied two feature model extensions (multi-valued attributes and cardinalities) to cope with complex product line variability modelling. We presented Formal Concept Analysis as a mathematical framework that brings theoretical foundations to the extraction of variability information from product descriptions. We also presented pattern structures, an FCA extension that allows to take into account more complex data, and allows us to consider not only the software variants' features, but also multi-valued attributes. We linked variability information that can be extracted by these two frameworks with the one found in the two considered FM extensions. We proposed an approach to extract complex variability information using these frameworks, as a part of extended FM synthesis from product descriptions (that is not studied in this paper). We used Product Comparison Matrices, a formalism that depicts a set of similar products and their characteristics in a tabular form, as descriptions of software families. We detailed the three steps of our extraction approach with an application on Wikipedia PCMs, and we illustrated each step with a toy example. We performed quantitative and qualitative evaluations of our method on 3 selected datasets to assess its applicability and usefulness. Our experiments showed that our approach does not suffer from scalability issues when applied on significant existing datasets, but that it needs to be completed by methods to efficiently separate meaningful relationships from accidental ones. This work is a first step toward a more generic approach to assist designers and practitioners into developing complex variability models from families of existing software systems.

In future work, we plan to study existing techniques to lower the number of considered extracted relationships by deepening the separation of the meaningful ones from the accidental ones. Some tracks of reflection are text-based techniques, and Formal Concept Analysis and Pattern Structures metrics [39]. These techniques could also be used for evaluating the relevance of each relationship and provide a ranking to assist a practitioner to select the ones they need. Another future work will be to consider the extraction of other kinds of variability information that could be useful to synthesise complex variability models from software descriptions. Particularly, relationships between several independent but connected software families are to be studied, allowing applications in the field of multiple software product lines. We deem that Relational Concept Analysis, another extension of Formal Concept Analysis allowing to extract relationships between several datasets, is a good candidate for this task. Finally, we only consider the first step of variability model synthesis from product descriptions, i.e., variability information extraction. In the future, we will address the second step which consists into building a complex variability model based on the obtained relationships.

References

[1] Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12). pp. 45–54. ACM (2012)

[2] Al-Msie'deen, R., Huchard, M., Seriai, A., Urtado, C., Vauttier, S.: Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In: Proceedings of the 11th International Conference on Concept Lattices and Their Applications (CLA'14). pp. 95–106 (2014)

[3] Alam, M., Le, T.N.N., Napoli, A.: LatViz: A New Practical Tool for Performing Interactive Exploration over Concept Lattices. In: Proceedings of the 13th International Conference on Concept Lattices and Their Applications (CLA'16). pp. 9–20 (2016)

[4] Andrews, S.: In-close, a fast algorithm for computing formal concepts. In: Supplementary Proceedings of 17th International Conference on Conceptual Structures (ICCS'09). pp. 1–14 (2009)

[5] Baixeries, J., Kaytoue, M., Napoli, A.: Computing similarity dependencies with pattern structures. In: Proceedings of the 10th International Conference on Concept Lattices and Their Applications (CLA'13). pp. 33–44. CEUR-WS.org (2013)

[6] Baixeries, J., Kaytoue, M., Napoli, A.: Characterizing functional dependencies in formal concept analysis with pattern structures. Annals of Mathematics and Artificial Intelligence 72(1-2), 129–149 (2014)

[7] Batory, D.S.: Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05). pp. 7–20. Springer (2005)

[8] Bécan, G., Behjati, R., Gotlieb, A., Acher, M.: Synthesis of attributed feature models from product descriptions. In: Proceedings of the 19th International Conference on Software Product Line (SPLC'15). pp. 1–10. ACM (2015)

[9] Benavides, D., Martín-Arroyo, P.T., Cortés, A.R.: Automated reasoning on feature models. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05). pp. 491–503. Lecture Notes in Computer Science, Springer (2005)

[10] Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615–636 (2010)

[11] Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wasowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13). pp. 7:1–7:8. ACM (2013)

[12] Berry, A., Gutierrez, A., Huchard, M., Napoli, A., Sigayret, A.: Hermes: a simple and efficient algorithm for building the AOC-poset of a binary relation. Annals of Mathematics and Artificial Intelligence 72(1-2), 45–71 (2014)

[13] Brabant, Q., Couceiro, M., Napoli, A., Reynaud, J.: From Meaningful Orderings in the Web of Data to Multi-level Pattern Structures. In: Proceedings of 23rd International Symposium on Foundations for Intelligent Systems (ISMIS'17). pp. 622–631. Lecture Notes in Computer Science, Springer (2017)

[14] Buzmakov, A., Egho, E., Jay, N., Kuznetsov, S.O., Napoli, A., Raïssi, C.: On mining complex sequential data by means of FCA and pattern structures. International Journal of General Systems 45(2), 135–159 (2016)

[15] Carbonnel, J., Huchard, M., Nebut, C.: Analyzing Variability in Product Families through Canonical Feature Diagrams. In: Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17). pp. 185–190 (2017)

[16] Carbonnel, J., Huchard, M., Nebut, C.: Towards the extraction of variability information to assist variability modelling of complex product lines. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'18). pp. 113–120. ACM (2018)

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

[18] Czarnecki, K.: Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models. Ph.D. thesis, Technische Universitat Illmenau, Germany (1999)

[19] Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative Programming for Embedded Software: An Industrial Experience Report. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02). pp. 156–172. Lecture Notes in Computer Science, Springer (2002)

[20] Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged Configuration Using Feature Models. In: Proceedings of the 3rd International Conference on Software Product Lines (SPLC'04). pp. 266–283. Lecture Notes in Computer Science, Springer (2004)

[21] Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: Proceedings of the 11th International Conference on Software Product Lines (SPLC'07). pp. 23–34. IEEE Computer Society (2007)

[22] Davril, J., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., Heymans, P.: Feature model extraction from large collections of informal product descriptions. In: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13). pp. 290–300. ACM (2013)
[23] Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz - open source graph drawing tools. In: International Symposium on Graph Drawing. pp. 483–484. Lecture Notes in Computer Science, Springer (2001)
[24] Ferrari, A., Spagnolo, G.O., Gnesi, S., Dell'Orletta, F.: Cmt and fde: Tools to bridge the gap between natural language documents and feature diagrams. In: Proceedings of the 19th International Conference on Software Product Line (SPLC '15). pp. 402–410. ACM (2015)
[25] Ganter, B., Kuznetsov, S.O.: Pattern Structures and Their Projections. In: Proceedings of the 9th International Conference on Conceptual Structures (ICCS'01). pp. 129–142. Lecture Notes in Computer Science, Springer (2001)
[26] Ganter, B., Wille, R.: Formal concept analysis - mathematical foundations. Springer (1999)
[27] Godin, R., Mili, H.: Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In: Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93). pp. 394–410. ACM (1993)
[28] Guénoche, A.: Construction du treillis de Galois d'une relation binaire. Mathématiques et Sciences Humaines 109, 41–53 (1990)
[29] Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Heymans, P.: Yo variability! JHipster: a playground for web-apps analyses. In: Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'17). pp. 44–51. ACM (2017)
[30] Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Reverse Engineering Feature Models from Programs' Feature Sets. In: Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11). pp. 308–312. IEEE Computer Society (2011)
[31] Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: On Extracting Feature Models from Sets of Valid Feature Combinations. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'13). pp. 53–67. Lecture Notes in Computer Science, Springer (2013)
[32] Haugen, O., Wasowski, A., Czarnecki, K.: CVL: Common Variability Language. In: Proceedings of the 17th International Software Product Line Conference (SPLC '13). pp. 277–277. ACM (2013)
[33] Holl, G., Grünbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. Information & Software Technology 54(8), 828–852 (2012)
[34] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021 (1990)
[35] Kaytoue, M., Kuznetsov, S.O., Napoli, A., Duplessis, S.: Mining gene expression data with pattern structures in formal concept analysis. Information Sciences 181(10), 1989–2001 (2011)
[36] Krajca, P., Outrata, J., Vychodil, V.: Advances in algorithms based on cbo. In: Proceedings of the 7th International Conference on Concept Lattices and Their Applications (CLA'10). pp. 325–337 (2010)
[37] Krueger, C.W.: Easing the transition to software mass customization. In: Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE'01). pp. 282–293. Lecture Notes in Computer Science, Springer (2001)
[38] Krueger, C.W.: Practical strategies and techniques for adopting software product lines. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR'02). pp. 349–350. Lecture Notes in Computer Science, Springer (2002)
[39] Kuznetsov, S.O., Makhalova, T.P.: On interestingness measures of formal concepts. Information Sciences 442-443, 202–219 (2018)
[40] Kuznetsov, S.O., Obiedkov, S.A.: Comparing performance of algorithms for generating concept lattices. Journal of Experimental and Theoretical Artificial Intelligence 14(2-3), 189–216 (2002)
[41] Lindig, C.: Fast concept analysis. Working with Conceptual Structures-Contributions to the 8th International Conference on Conceptual Structures 2000, 152–161 (2000)
[42] Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Feature Model Synthesis with Genetic Programming. In: Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE'14). pp. 153–167. Lecture Notes in Computer Science, Springer (2014)
[43] Loesch, F., Ploedereder, E.: Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems (CSMR'07). pp. 159–170. IEEE Computer Society (2007)
[44] Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. Journal of Systems and Software 103, 353–369 (2015)
[45] Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Proceedings of the 2nd International Conference on Software Product Lines (SPLC'02). pp. 176–187. Lecture Notes in Computer Science, Springer (2002)
[46] Martinez, J., Tërnava, X., Ziadi, T.: Software product line extraction from variability-rich systems: the robocode case study. In: Proceeedings of the 22nd International Conference on Systems and Software Product Line (SPLC'18). pp. 132–142. ACM (2018)
[47] Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Automating the extraction of model-based software product lines from model variants (T). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15). pp. 396–406. IEEE Computer Society (2015)
[48] Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up technologies for reuse: automated extractive adoption of software product lines. In: Proceedings of the 39th International Conference on Software Engineering, (ICSE'17), Companion Volume. pp. 67–70. IEEE Computer Society (2017)
[49] Martinez, J., Ziadi, T., Mazo, R., Bissyandé, T.F., Klein, J., Traon, Y.L.: Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In: Proceedings of the Second IEEE Working Conference on Software Visualization (VISSOFT'14). pp. 50–59. IEEE Computer Society (2014)
[50] Nasr, S.B., Bécan, G., Acher, M., Filho, J.B.F., Baudry, B., Sannier, N., Davril, J.: MatrixMiner: a red pill to architect informal product descriptions in the matrix. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15). pp. 982–985. ACM (2015)
[51] Nasr, S.B., Bécan, G., Acher, M., Filho, J.B.F., Sannier, N., Baudry, B., Davril, J.: Automated extraction of product comparison matrices from informal product descriptions. Journal of Systems and Software 124, 82–103 (2017)

[52] Niu, N., Easterbrook, S.M.: Concept analysis for product line requirements. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09). pp. 137–148. ACM (2009)
[53] Pan, J.Z., Staab, S., Aßmann, U., Ebert, J., Zhao, Y.: Ontology-driven software development. Springer Science & Business Media (2012)
[54] Poelmans, J., Elzinga, P., Viaene, S., Dedene, G.: Formal concept analysis in knowledge discovery: a survey. In: Proceedings of the 8th International Conference on Conceptual Structures (ICCS'10). pp. 139–153. Lecture Notes in Computer Science, Springer (2010)
[55] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer Science & Business Media (2005)
[56] Reynaud, J., Alam, M., Toussaint, Y., Napoli, A.: A Proposal for Classifying the Content of the Web of Data Based on FCA and Pattern Structures. In: Proceedings of 23rd International Symposium on Foundations for Intelligent Systems (ISMIS'17). pp. 684–694. Lecture Notes in Computer Science, Springer (2017)
[57] Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with uml multiplicities. In: Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT'02) (2002)
[58] Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of feature models from formal contexts. In: Workshop Proceedings (Volume 2) of the 15th International Conference on Software Product Lines (SPLC'11). pp. 4:1–4:8. IEEE Computer Society (2011)
[59] Sannier, N., Acher, M., Baudry, B.: From comparison matrix to Variability Model: The Wikipedia case study. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE'13). pp. 580–585. IEEE (2013)
[60] Shatnawi, A., Seriai, A., Sahraoui, H.A.: Recovering software product line architecture of a family of object-oriented product variants. Journal of Systems and Software 131, 325–346 (2017)
[61] She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Proceedings of the 33rd International Conference on Software Engineering, (ICSE'11). pp. 461–470. ACM (2011)
[62] Snelting, G.: Software reengineering based on concept lattices. In: Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00). pp. 3–10. IEEE Computer Society (2000)
[63] Tilley, T., Cole, R., Becker, P., Eklund, P.W.: A Survey of Formal Concept Analysis Support for Software Engineering Activities. In: Formal Concept Analysis, Foundations and Applications. Lecture Notes in Computer Science, vol. 3626, pp. 250–271. Springer (2005)
[64] Xue, Y., Xing, Z., Jarzabek, S.: Feature Location in a Collection of Product Variants. In: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12). pp. 145–154. IEEE Computer Society (2012)