Concept lattices: a representation space to structure software variability

Ra'Fat Ahmad Al-Msie'Deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, Sylvain Vauttier

2014 5th International Conference on Information and Communication Systems (ICICS)

Concept lattices: a representation space to structure software variability

R. AL-msie'deen[1], M. Huchard[1], A.-D. Seriai[1], C. Urtado[2], S. Vauttier[2] and A. Al-Khlifat[3]

[1] LIRMM / CNRS & Montpellier 2 University, Montpellier, France
{al-msiedee, huchard, seriai}@lirmm.fr
[2] LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr
[3] Al-Balqa' Applied University Salt, Jordan
{amak_n}@hotmail.com

***Abstract*—Formal Concept Analysis is a theoretical framework which structures a set of objects described by properties. Formal Concept Analysis is a classification technique that takes data sets of objects and their attributes, and extracts relations between these objects according to the attributes they share. This structure reveals and categorizes commonalities and variability in a canonical form. From this canonical form, other structures can be derived, that can be more or less complex. In this paper, we revisit two papers from the literature of the software product line domain. We point to key contributions and limits of the representation of variability by concept lattices, with illustrative examples. We present tools to implement the approach and open a discussion.**

Keywords*—*Variability, Software Product Line, Formal Concept Analysis, Concept Lattice, Feature Model.

I. INTRODUCTION

Studying variability in domain and software is a key issue of product line engineering. From this study, a designer may identify commonalities and variants of products, and be guided in migrating products into a structured software product line, or at improving its structure. Several papers have been published on that topic [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]. Two of them ([3] [7]) emphasize the use of Formal Concept Analysis [11] as a tool for understanding and extracting variability, but we think that they do not fully exploit the approach.

In this paper, we recall the main characteristics of FCA (Section II) that are valuable for variability analysis and representation (Section III). We also present simple algorithms for building concept structures, to help detect similar algorithms that may appear in the literature and to clarify the construction. We recall how concept structures have been used in [3] [7]. At last, we revisit two recent representative papers on variability in the light of concept structures (Section IV). We conclude and draw perspectives for this work in Section V.

II. CONCEPT LATTICES: A FRAMEWORK FOR EXPRESSING COMMONALITIES AND VARIABILITY

Galois lattices [12] and concept lattices [11] are core structures of a data analysis framework (Formal Concept Analysis, or FCA for short) for extracting an ordered set of concepts from a dataset, called a Formal Context, composed of objects described by attributes. This data analysis framework is currently applied to support various tasks, including information retrieval [13], data mining [14], building or maintaining class hierarchies in object-oriented software [15], software understanding [16] or ontology alignment [17]. In this section, we present the basics of FCA, and we highlight some of the properties of FCA that are useful for variability structuring.

*Definition 2.1 (**Formal Context**):* A formal context is a triple $K = (O, A, R)$ where O and A are sets (objects and attributes, respectively) and R is a binary relation, *i.e.*, $R \subseteq O \times A$.

Choosing the right objects, the right attributes and the right relation is a key modelling issue that strongly impacts the analysis. Attributes are often dependent one from another, and this dependency has to be reflected in the relation. A fine representation has to be designed when attributes are numbers. Objects (*resp.* attributes) may have to be grouped, *etc.*

In the context of software product lines, a main idea is that software products can be described by artifacts (from analysis diagrams, code, or documentation) or identified high-level features. As it is one of the approaches we want to consider in the light of FCA, we use the example from [8] to illustrate the notion of Formal Context. In this example, 8 products (bank systems) are described by *construction primitives*, corresponding to the creation of main artifacts (packages, classes, attributes and operations). Products constitute the rows of the Table I, while construction primitives constitute the columns. In the other example we will consider [9], wiki systems (rows) are described by their characteristics (columns) as shown in Table IV.

*Definition 2.2 (**Formal Concept**):* Given a formal context $K = (O, A, R)$, a formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and an attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept, $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept.

*Definition 2.3 (**Concept Specialization Order**):* Given a formal context $K = (O, A, R)$, and two formal concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ of K, the concept specialization order $\leq_s$ is defined by $C_1 = (E_1, I_1) \leq_s C_2 = (E_2, I_2)$ if and only if $E_1 \subseteq E_2$ (and equivalently $I_2 \subseteq I_1$). C_1 is called a subconcept of C_2. C_2 is called a super-concept of C_1.

978-1-4799-3023-4/14/$31.00 ©2014 IEEE

TABLE I. A FORMAL CONTEXT DESCRIBING BANK SYSTEMS BY CONSTRUCTION PRIMITIVES.

	CreatePackage(bs)	CreateClass(Bank,bs)	CreateAttribute(accounts,Bank)	CreateOperation(depositOnAccount,Bank)	CreateOperation(withdrawFromAccount,Bank)	CreateClass(Account,bs)	CreateAttribute(id,Account)	CreateAttribute(amount,Account)	CreateOperation(deposit,Account)	CreateOperation(getAmount,Account)	CreateClass(Client,bs)	CreateAttribute(name,Client)	CreateAttribute(id,Client)
Product1Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product2Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product3Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product4Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product5Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product6Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product7Bank	×	×	×	×	×	×	×	×	×	×	×	×	×
Product8Bank	×	×	×	×	×	×	×	×	×	×	×	×	×

	CreateAttribute(currency,Account)	CreateClass(Converter,bs)	CreateOperation(conv,Converter)	CreateAttribute(converter,Bank)	CreateOperation(convert,Bank)	CreateAttribute(limit,Account)	CreateOperation(withdrawWithLimit,Account)	CreateOperation(getLimit,Account)	CreateOperation(setLimit,Account)	CreateOperation(withdrawWithoutLimit,Account)	CreateClass(Consortium,bs)	CreateAttribute(cons,Bank)
Product1Bank	×	×	×	×	×	×	×	×	×		×	×
Product2Bank										×		
Product3Bank	×	×	×	×	×					×	×	×
Product4Bank	×	×	×	×	×	×	×	×	×			
Product5Bank										×	×	×
Product6Bank						×	×	×	×			
Product7Bank						×	×	×	×		×	×
Product8Bank	×	×	×	×	×					×		

For example, *Concept_9* is a subconcept of *Concept_12* (*cf.*Table II and Table III). Due to this specialization order definition, an evident property is that a subconcept owns (inherits in top-down manner) the attributes of its super-concepts, while a super-concept owns (inherits in bottom-up manner) the objects of its subconcepts. This is why, for simplicity's sake, most lattice representations show attributes (*resp.* objects) solely where they are introduced (not repeating the inherited ones). They are said to show the simplified intents and simplified extents.

TABLE II. CONCEPT_12 IN FIGURE 1 (*i.e.*, FORMAL CONCEPT EXAMPLE).

Extent	Intent
Product1Bank Product3Bank Product5Bank Product7Bank	CreatePackage(bs) CreateClass(Bank,bs) CreateAttribute(accounts,Bank) CreateOperation(depositOnAccount,Bank) CreateOperation(withdrawFromAccount,Bank) CreateClass(Account,bs) CreateAttribute(id,Account) CreateAttribute(amount,Account) CreateOperation(deposit,Account) CreateOperation(getAmount,Account) CreateClass(Client,bs) CreateAttribute(name,Client) CreateAttribute(id,Client) CreateClass(Consortium, bs) CreateAttribute(cons, bank)

TABLE III. CONCEPT_9 IN FIGURE 1 (*i.e.*, FORMAL CONCEPT EXAMPLE).

Extent	Intent
Product1Bank Product3Bank	CreatePackage(bs) CreateClass(Bank,bs) CreateOperation(withdrawFromAccount,Bank) CreateClass(Account,bs) CreateAttribute(id,Account) CreateAttribute(amount,Account) CreateOperation(deposit,Account) CreateOperation(getAmount,Account) CreateClass(Client,bs) CreateAttribute(name,Client) CreateAttribute(id,Client) CreateClass(Consortium, bs) CreateAttribute(cons, bank) CreateAttribute(currency,Account) CreateClass(Converter,bs) CreateOperation(conv,Converter) CreateAttribute(converter,Bank) CreateOperation(convert,Bank)

*Definition 2.4 (**Concept Lattice**):* Let $\mathcal{C}_K$ be the set of all concepts of a formal context K. This set of concepts provided with the specialization order $(\mathcal{C}_K, \leq_s)$ has a lattice structure, and is called the concept lattice associated with K.

Figure 1 shows the concept lattice associated with the formal context of Table I.

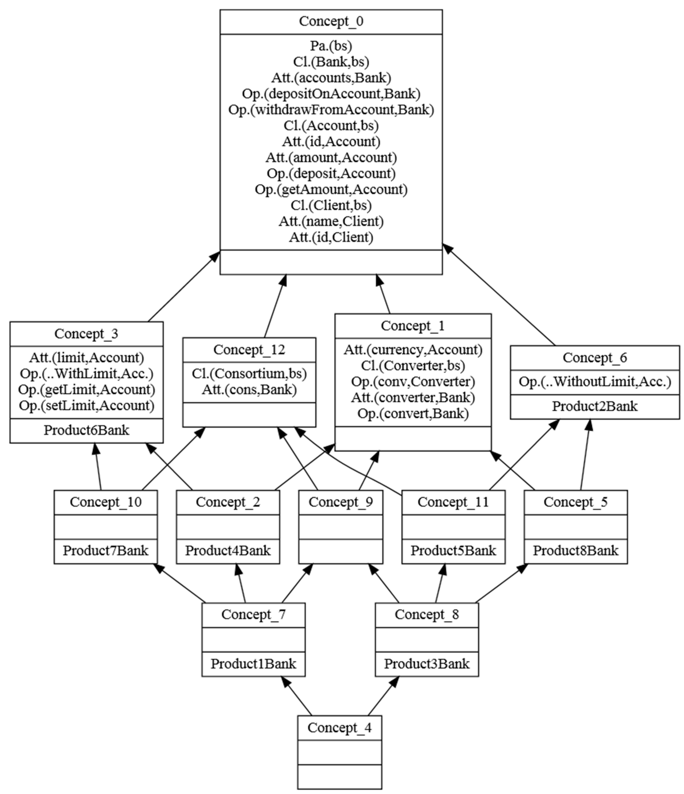

Fig. 1. The concept lattice for the formal context of Table I.

Algorithm 1 is a simple algorithm for building the Hasse diagram of a concept lattice (not recommended for implementation, but useful to understand how concepts are formed).

The reader may have noticed that, applying the simplification of extents and intents (removing inherited elements), some concepts, like *Concept_9*, are represented having empty simplified extent and intent. These concepts introduce neither objects nor attributes. In several FCA applications, they can

Algorithm 1: ComputeConceptLattice(K)

Data: K: a formal context
Result: ($\mathcal{C}_K$, $\leq_s$): the concept lattice associated with K

// compute the concepts of $\mathcal{C}_K$
$\mathcal{C}_K \leftarrow \emptyset$
foreach *i from* $|O|$ *to 1* **do**
    foreach *subset* $S \subseteq O$*, with* $|S| = i$ **do**
        compute $I_S = \{a \in A | \forall o \in S, (o, a) \in R\}$ the shared attributes
        if I_S *is not the intent of a concept already calculated in* $\mathcal{C}_K$ **then**
            $\mathcal{C}_K \leftarrow \mathcal{C}_K \cup (S, I_S)$
// establish the specialization order
Compute the transitive reduction of $\leq_s$ by comparing the concept extents in $\mathcal{C}_K$

be ignored (*e.g.,* in [15] [3] [7]).

Reversely, the term *object concept* (*resp. attribute concept*) refers to a concept which introduces at least one object (*resp.* attribute). In Figure 1, Concepts 0, 1, 3, 6, 12 are attribute concepts; Concepts 2, 3, 5, 6, 7, 8, 10, 11, are object concepts; Concepts 4 and 9 do not introduce any object or any attribute.

*Definition 2.5 (**AOC-poset**):* The AOC-poset (for Attribute-Object-Concept poset) is the sub-order of ($\mathcal{C}_K$, $\leq_s$) restricted to object-concepts and attribute-concepts.

Algorithm 2 is a simple algorithm for building the Hasse diagram of the AOC-poset. In this algorithm, we use complementary classical FCA notations: for any object set $S_o \subseteq O$, the set of shared attributes is $S'_o = \{a \in A | \forall o \in S_o, (o, a) \in R\}$, and for any attribute set $S_a \subseteq A$, the set of owners is $S'_a = \{o \in O | \forall a \in S_a, (o, a) \in R\}$. Figure 2 shows the AOC-poset for the context of Table I (Concepts 9 and 4 have been removed from the lattice; concepts have been re-numbered by the tool).

Algorithm 2: ComputeAOCposet(K)

Data: K: a formal context
Result: (AOC_K, $\leq_s$): the AOC-poset associated with K

// compute the object concepts and the attribute concepts
$AOC_K \leftarrow \emptyset$
foreach $o \in O$ **do**
    $AOC_K \leftarrow AOC_K \cup (\{o\}'', \{o\}')$ // that is, objects that share the same attributes as o, with the attributes of o
foreach $a \in A$ **do**
    $AOC_K \leftarrow AOC_K \cup (\{a\}', \{a\}'')$ //that is, objects that share the attribute a, with the attributes they share
// establish the specialization order
Compute the transitive reduction of $\leq_s$ by comparing the concept extents in AOC_K

There is a drastic difference of complexity between the two structures, because the concept lattice may have $2^{min(|O|,|A|)}$ concepts, while the number of concepts in the AOC-poset is bounded by $|O| + |A|$ [18] [19] [20] [21] [22]. Most of the algorithms for building concept lattices are cited and compared in [23]. Algorithms for building AOC-posets are introduced and compared in [24], except for the more recent one [25]. Most of the existing tools are referenced from the web page of Uta Priss [26]. For this paper, we used the eclipse eRCA platform [27].

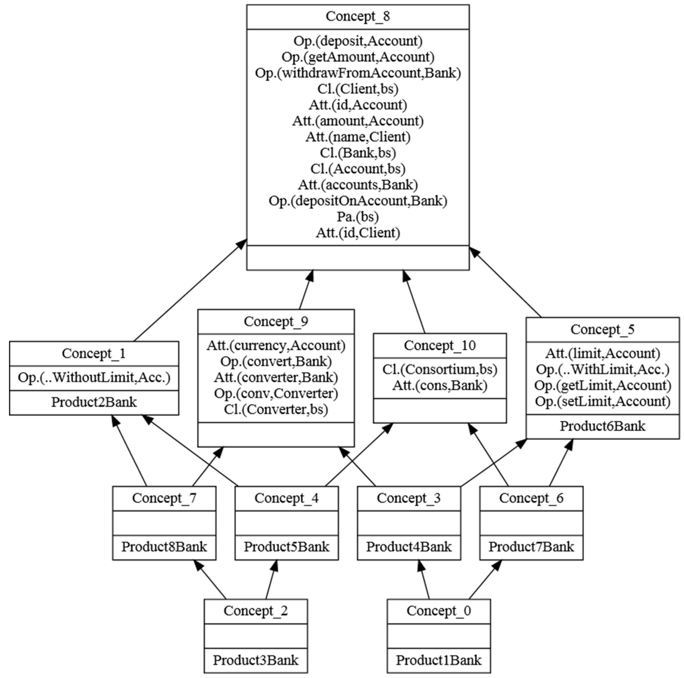

Fig. 2. The AOC-poset for the formal context of Table I.

III. Properties of the Concept Lattice and the AOC-poset with Regard to Variability

Now that definitions have been given, we can, in the tracks of [3] [7], highlight some interesting properties of concept lattices or AOC-posets with regard to variability. Note that [3] uses the concept lattice. [7] prefers to use the lattice reduced to attribute concepts (that we call the AC-poset for attribute concept poset). In these two papers, a formal context describes the products through their high-level features rather than artifacts extracted from source code or construction primitives, but their underlying principles remain the same. These structures (often referred in the following as concept structures) contain many information about both products and the way attributes are present in these products. Lessons can also be learnt about relations among attributes (for example implications), that are true for the considered products.

Commonalities are found in the top concept. They correspond to artifacts that are always present or mandatory features [3]. If some attributes appear in the bottom concept, this means they are never used in products (dead features) [3]. **Mutually exclusive features** (or artifacts) can be recognized in the concept lattice using the meet (denoted by $\sqcap$) lattice operation ([3]), or computing the greatest lower bounds in the AOC-poset. If a feature f_1 is introduced in concept C_1, a feature f_2 is introduced in concept C_2 and $C_1 \sqcap C_2 = \perp$, that is, if the bottom of the lattice is the greatest lower bound of C_1 and C_2, the two features never occur together in a product. In the lattice of Figure 1, $CreateOperation(withdrawWithLimit, Account)$ and $CreateOperation(withdrawWithoutLimit, Account)$ are introduced in *Concept_3* and *Concept_6* respectively. The two concepts meet at the bottom concept in the lattice, or never meet in the AOC-poset, meaning they are exclusive alternatives. **Required relations** can also be easily obtained in the concept structure [7]. Data mining research often uses a property of concept structures to extract implication rules: when an attribute (a feature or an artifact) a_1 is introduced in a subconcept of the concept that introduces another attribute a_2, there is an implication: $a_1 \rightarrow a_2$. This also means that if two attributes a_1 and a_2 are introduced in the same concept, $a_1 \leftrightarrow a_2$ (they are co-occurrence). In [7], authors propose an algorithm to extract all the implications from the AC-poset. For example, in the lattice of Figure 3, $storage \leftrightarrow unicode$ (from Concept_6) and $License\_GPL2 \rightarrow Language\_PHP$. To mine **implications with negations** (called exclude constraints in [9]), we use the meet of the introducers of the two involved features. For example, the meet of *Concept_11* which introduces LicenseCostFree_DifferentLicenses and *Concept_4* which introduces License_GPL is the bottom. We can thus deduce that $LicenseCostFree\_DifferentLicenses \rightarrow \neg License\_GPL$.

We enrich this set of properties already noticed by [3] [7] by a few original ones, we believe will be useful in future work using FCA for variability representation. First, the simplified intents of concepts correspond to variability blocks. We can hypothesize these blocks contain both main features and alternatives. In Figure 2, the intent of *Concept_9* effectively corresponds to a feature (conversion), but the intent of *Concept_8* acts more as a miscellanea that should be redivided using extra information, as linguistic information or code dependencies.

Then, the presence of products in concept extents and their position in the concept structure have been under-exploited although they might reveal important information on the product variants. A concept's *support* (number of objects in the extent) is often used in data-mining as one of the indicators on implication and association rules. We should consider it as an interesting measure on variability blocks. For example, the concept structure helps identify the variability blocks (and their corresponding features) that are often shared (*e.g.,* $Storage, Unicode$ shared by 7 wikis) and those that are rare (*e.g.,* $Language\_Java$ which we can find only in $Confluence$).

If the products are successive versions, analyzing concept extents may help identify evolution patterns: blocks of features that appear in a version, that disappear, that disappear then reappear, *etc.* In very large software product lines, only frequent variability blocks can be extracted, using the iceberg lattices [28], to have a simplified view. Besides, indicative relationships between products can be read directly from the concept structure. Generally speaking, we can say that the concept structure is a kind of classification of products with regard to exposed features. If two products are in the same simplified extent (*e.g.,*, in the wiki lattice in Figure 3, the extent of *Concept_14* contains both $DokuWiki$ and $PmWiki$), these products have an equivalent set of features (or artifacts). One may replace the other in some cases. If a product p_1 (*e.g.,* Product8Bank in *Concept_7*) is introduced in a subconcept of a concept that introduces a product p_2 (*e.g.,* Product2Bank in *Concept_1*), p_1 adds new features to p_2. The concept structure also reveals that a product merges the features of several other products (*e.g.,* Product3Bank merges artifacts from Product5Bank and Product8Bank). Similarity between products can be measured with a proximity measure in the concept structure like it has been done for ontologies [29], using various variants of path length calculus or set comparison operations on the intents of the involved concepts. Incompatible products can be revealed by having the meet of their introducer concepts at the bottom (or nothing).

The concept structure also produces concepts that do not correspond to existing products but to suggested *abstract products* that can be built by navigating the concept structure. For example, *Concept_12* in lattice 1. Such products could be interesting and relatively cheap to develop as they conform to the product line definition and would be alternate feature combinations that might be interesting for software products' customers.

Some patterns in the concept structure may be of interest to structure the whole software product family. For example, some lattices are disconnected into smaller pieces when removing the top and the bottom concepts. In this case, we obtain an *horizontal decomposition* of the lattice, that may help find sub-families among software products (*a.k.a.* product ranges).

This view on variability has its limits. First, it is very **dependent on the set of products** that are considered, and a kind of "closed world assumption" is made. Moreover, as noticed in [8], separating variability blocks that constitute concepts intents into features **needs extra information**. In some cases, we believe this extra information should be available early, to be included in the formal context itself, so as to be treated uniformly when extracting variability. Besides, the structure of **the feature tree is not directly in the lattice**, except if a wise encoding is used for formal contexts that embeds the feature structure (*cf.* illustrative details on the wiki example below).

IV. Revisiting some SPL reverse engineering approaches in the light of FCA

Several approaches extract features or feature models from products or domains including [1] [4] [5] [6] [2]. Our objective is not to be exhaustive and we do not pretend that concept structures are useful for or underlying all the approaches. We choose to focus on two representative approaches (where concept structures might be interesting to look at) to have enough space to go into details and open a discussion. We use the first approach to show that detected variability can sometimes be mapped into concepts of the concept structure. As for the second approach, it is used to raise the question of the potential complementarity of concept structures and feature models.

A. FCA as a framework (Ziadi et al. [8])

Authors of this paper propose an automatic approach for feature identification from source code for a set of product variants. They assume that the product variants use the same vocabulary to name packages, classes, attributes and methods.

They describe the products with construction primitives, as it has been shown in Table I. The recovered feature model contains a single mandatory feature that includes the common parts for all product variants' source code, and optional features. What is interesting is a clever algorithm for variability block (feature) identification that we rephrase hereafter using our notations.

Let $K = (O, A, R)$ be the formal context between products and construction primitives (*cf.* Table I). F is the resulting set, it will be composed of subsets of A. R_{work} is initially the relation R, R_{work} will evolve during the algorithm. Note that in the algorithm $\{a\}'$ and $\{a\}''$ are computed in R_{work}.

Algorithm 3: ComputeFeatures(R)

Data: R: a binary relation (a part of a formal context $K = (O, A, R)$)
Result: F: a set of subsets of attributes from A

F $\leftarrow \emptyset$
$\mathrm{R}_{work} \leftarrow$ R
while $R_{work} \neq \emptyset$ **do**
    mfcp $\leftarrow$ any element $a \in A$ with a maximal $|\{a\}'|$
    products $\leftarrow$ {a}' // the products that have a in R_{work}
    f $\leftarrow \{a\}''$ // the attributes shared by products having a in R_{work}
    $\mathrm{R}_{work} \leftarrow \mathrm{R}_{work} \setminus \{(x, y)\ s.t.\ x \in f\}$
    F $\leftarrow$ F $\cup \{f\}$

This algorithm efficiently builds the simplified intents of the attribute concepts (Concepts 1, 5, 8, 9, 10 in Figure 2), going top-down: the attributes are considered from the most to the less frequent. When an attribute a is considered, $\{a\}''$ is computed in R_{work} and added to the result (F). This is equivalent to computing $\{a\}''$ in R and then removing the attributes that are strictly more frequent than a (that also are inherited attributes). This is thus the simplified intent of a concept.

This is an example where variability elements can be mapped to concepts of the concept structure, and this highlights their algorithm and gives foundations of their result in the FCA framework. For the specific application the authors chose in this paper, it was not necessary to build the whole structure. Anyway, this is interesting to know that with a slight modification of their algorithm (adding the edges of the conceptual structure), they could, for example, mine knowledge about mutually exclusive construction primitives.

B. *FCA for a complementary view (Acher et al. [9])*

This other paper we would like to elaborate on, takes product descriptions as its input and synthesizes a feature model. Products are described by characteristics (language, license, *etc.*) with different patterns on values (many-valued, one-valued, *etc.*). Table IV shows how we interpreted the values in a formal context. As the authors of the paper, we think that such an interpretation has to be guided by the user. In a sense, rather than building a feature model for each product and then assembling them, the formal context gathers product descriptions.

At first sight, with a concept structure approach, we might find it difficult to have a feature tree that resembles the author's. Concept structures would not tell us what the main features and their variations are. But given that this information is included in the table itself (columns of the initial table in the paper, reproduced as columns without symbol _ in our formal context), the root and the first level of the feature model can be built. The lattice contains information to decide whether main features are mandatory (RSS, or $License$), optional ($Language$, or $Storage$), or alternatives ($LicenseCostFee$ and $Storage$). It also makes it possible to know if feature values are xor groups (the values of $License$ for example), or mutex (values of $LicenseCostFee$). Constraints can also be further deduced with specific lattice operations as we explained before. Of course, in the concept structure, as when authors assemble the feature models, we can read many things, but they remain assumptions that should be user-validated. For example, when considering two features (or feature values), introduced by two comparable concepts, we can deduce a real require constraint, but this situation can also be purely fortuitous.

Comparing theoretically the merging of feature models and what can be learned in the concept structure would really be interesting. We think that a hint in that direction might be to define, for any feature model, an equivalent (minimal if possible, but it is not necessary) representative set of products, and then to derive a concept structure from this object set. At least, we can say that merging the collection of feature models on one side and building the concept structure on the other side give us complementary views on variability.

TABLE IV. A FORMAL CONTEXT DESCRIBING WIKI SYSTEMS BY CHARACTERISTICS.

	License	License_Commercial	License_NoLimit	License_GPL	License_GPL2	Language	Language_Java	Language_Python	Language_PHP	Language_Perl
Confluence	×	×				×	×			
Pbwiki	×		×							
MoinMoin	×			×		×		×		
DokuWiki	×				×	×			×	
PmWiki	×				×	×			×	
DrupalWiki	×				×	×			×	
Twiki	×			×		×				×
MediaWiki	×			×		×			×	

	Storage	Storage_DB	Storage_Files	Storage_FilesRCS	LicenseCostFree	LicenseCostFree_US10	LicenseCostFree_DifferentLicenses	LicenseCostFree_Community	RSS	Unicode
Confluence	×	×				×			×	×
Pbwiki					×				×	
MoinMoin	×		×						×	×
DokuWiki	×		×						×	×
PmWiki	×		×						×	×
DrupalWiki	×	×					×		×	×
Twiki	×			×				×	×	×
MediaWiki	×	×							×	×

V. CONCLUSION AND PERSPECTIVES

In this paper, we focused on concept structures and the opportunities they offer for structuring variability. Concept structures can be seen a summary of known data (artifacts, features, *etc.*) for a set of products. Existing theory and algorithms

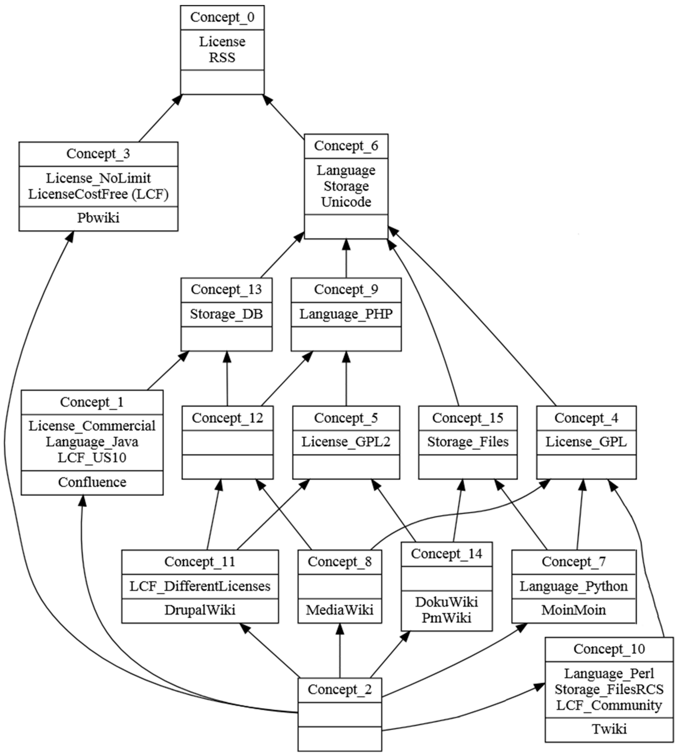

Fig. 3. The concept lattice for the formal context of Table IV.

can be used to extract many information about variability (variability blocks, constraints, relationships between products, structure of the software product line, *etc.*) and cope with the intrinsic complexity of the whole concept lattice. We revisited two representative papers on variability extraction to show their commonality and complementarity with the concept structure view.

Some theoretical questions are raised on the relations between feature models and concept structures that we would like to explore into more details. We have discussed about extracting information from the concept structure in order to build (part of) a feature model. Reversely, from a feature model, can we map to a canonical concept structure that embeds the same information? Besides, as we explained, many opportunities offered by concept structures are not applied to variability yet, thus opening paths to plenty of future works.

Acknowledgments: This work has been supported by project CUTTER ANR-10-BLAN-0219.

References

[1] Y. Yang, X. Peng, and W. Zhao, "Domain feature model recovery from multiple applications using data access semantics and formal concept analysis," in *WCRE*, 2009, pp. 215–224.

[2] S. Duszynski, "A scalable goal-oriented approach to software variability recovery," in *SPLC Workshops*, 2011, p. 42.

[3] F. Loesch and E. Ploedereder, "Restructuring variability in software product lines using concept analysis of product configurations," in *CSMR*, 2007, pp. 159–170.

[4] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, "An exploratory study of information retrieval techniques in domain analysis," in *SPLC*, 2008, pp. 67–76.

[5] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*, 2011, pp. 461–470.

[6] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *ECSA*, 2011, pp. 220–235.

[7] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Extraction of feature models from formal contexts," in *SPLC Workshops*, 2011, p. 4.

[8] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *CSMR*, 2012, pp. 417–422.

[9] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *VaMoS*, 2012, pp. 45–54.

[10] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *WCRE*, 2012, pp. 145–154.

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

[12] M. Barbut and B. Monjardet, *Ordre et Classification: Algèbre et combinatoire*. Hachette, 1970, vol. 2.

[13] C. Carpineto and G. Romano, "Exploiting the potential of concept lattices for information retrieval with credo," *j-jucs*, vol. 10, no. 8, pp. 985–1013, aug 2004.

[14] P. Valtchev, R. Missaoui, and R. Godin, "Formal concept analysis for knowledge discovery and data mining: The new challenges," in *ICFCA 2004*, ser. LNCS, vol. 2961. Springer, 2004, pp. 352–371.

[15] R. Godin and H. Mili, "Building and maintaining analysis-level class hierarchies using galois lattices," in *OOPSLA*, 1993, pp. 394–410.

[16] M. U. Bhatti, N. Anquetil, M. Huchard, and S. Ducasse, "A catalog of patterns for concept lattice interpretation in software reengineering," in *SEKE*, 2012, pp. 118–123.

[17] G. Stumme and A. Maedche, "Ontology merging for federated ontologies on the semantic web," in *FMII*, 2001, pp. 413–418.

[18] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing," in *SEKE*, 2013, pp. 244–249.

[19] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using formal concept analysis," in *ICSR*. Springer, 2013, pp. 302–307.

[20] R. Al-Msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity," in *IRI*. IEEE, 2013, pp. 586–593.

[21] R. Al-Msie'deen, A. Seriai, and M. Huchard, *Reengineering software product variants into software product line: REVPLINE approach*. LAP LAMBERT Academic Publishing, January 2014.

[22] R. AL-Msie'deen, A. D. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "An approach to recover feature models from object-oriented source code," in *Actes de la Journée Lignes de Produits 2012*, Lille, France, Novembre 2012, pp. 15–26.

[23] S. O. Kuznetsov and S. A. Obiedkov, "Comparing performance of algorithms for generating concept lattices," *J. Exp. Theor. Artif. Intell.*, vol. 14, no. 2-3, pp. 189–216, 2002.

[24] G. Arévalo, A. Berry, M. Huchard, G. Perrot, and A. Sigayret, "Performances of galois sub-hierarchy-building algorithms," in *ICFCA*, 2007, pp. 166–180.

[25] A. Berry, M. Huchard, A. Napoli, and A. Sigayret, "Hermes: an efficient algorithm for building galois sub-hierarchies," in *to appear in proceedings of Concept Lattices and Applications (CLA 2012)*, 2012.

[26] Uta Priss: http://www.upriss.org.uk/fca/fca.html, 2007.

[27] Eclipse eRCA Platform: https://code.google.com/p/erca/, 2010.

[28] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal, "Computing iceberg concept lattices with titanic," *Data Knowl. Eng.*, vol. 42, no. 2, pp. 189–222, 2002.

[29] E. G. M. Petrakis, G. Varelas, A. Hliaoutakis, and P. Raftopoulou, "X-similarity: Computing semantic similarity between concepts from different ontologies," *JDIM*, vol. 4, no. 4, pp. 233–237, 2006.