FCA for Software Product Lines Representation: Mixing Configuration and Feature Relationships in a Unique Canonical Representation

Jessie Carbonnel, Karell Bertet, Marianne Huchard, Clémentine Nebut

▶ To cite this version:

Jessie Carbonnel, Karell Bertet, Marianne Huchard, Clémentine Nebut. FCA for Software Product Lines Representation: Mixing Configuration and Feature Relationships in a Unique Canonical Representation. CLA: Concept Lattices and their Applications, HSE, Moscow Russia, Jul 2016, Moscow, Russia. pp.109-122. lirmm-01354971

HAL Id: lirmm-01354971

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01354971

Submitted on 21 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FCA for Software Product Lines Representation: Mixing Configuration and Feature Relationships in a Unique Canonical Representation

Jessie Carbonnel[1], Karell Bertet[2], Marianne Huchard[1] and Clémentine Nebut[1]

[1] LIRMM, CNRS and Université de Montpellier, France
[2] L3i, Université de La Rochelle, France
{jcarbonnel,huchard,nebut}@lirmm.fr,bertet@univ-lr.fr

Abstract. Software Product Line Engineering (SPLE) is a software engineering domain in which families of similar softwares (called products) are built reusing common artifacts. This requires to analyze commonalities and variabilities, for example to detect which parts are common to several products and which parts differ from one product to another. Such software characteristics that may be present or not in a product are called features. Several approaches in the literature exist to organize features and product configurations in terms of features. In this paper we review those approaches and show that concept lattices are a relevant structure to organize features and product configurations. We also address scaling issues related to formal context computation in the domain of SPLE.

Keywords: Software Product Lines, Feature Model, Formal Concept Analysis, Concept Lattice

1 Introduction

Software Product Line Engineering (SPLE) focuses on the reuse of common software pieces to reduce the building and maintenance cost of similar software systems (called products). An important step of this methodology consists in analyzing and modeling variability, i.e. mainly extracting "features", a feature being a discriminating characteristic common to several products or specific to a product. A product configuration is then a set of these features. Different formalisms are used in SPLE to organize features and product configurations. Some of these formalisms focus on features, while others represent product configurations. Some are canonical, while others are not, and depend on the designer point of view.

In this paper, we review the main used formalisms and we show that concept lattices might be a relevant (canonical) structure for representing variability, while highlighting information on relationships between product configurations, and between product configurations and features, that other formalisms hardly represent. Besides explaining what is the contribution of concept lattices to variability representation, we propose a solution to address some scaling issues of

concept lattices in this domain. Actually, scaling issues can occur at two levels when computing: (1) the formal context, and (2) the concept lattice. Here, we focus on scaling issues related to formal context computation; we investigate implicative systems on attributes as closure operators to build a feature closed sets lattice without building the formal context. We show that implicative systems are another representation of the variability that can be useful for designers.

The remaining of the paper is organized as follows. Section 2 presents the various formalisms found in the literature to capture the variability of a software product line. Section 3 shows that concept lattices are an interesting formalism to analyse variability, and presents related work concerning the use of formal concept analysis for product lines. Section 4 explains how using implicative systems allows to face scaling issues related to formal context computation for variability management.

2 Existing Formalisms for variability representation

To capture and describe the variability of a software product line, almost all approaches in the literature use feature-oriented representations [11, 12, 20]. Features describe and discriminate the products. As an example, features for an e-commerce website may include displaying a *catalog*, proposing to fill a *basket* of products, or offering a *payment_ method*. In our context, we consider a feature set F. A product configuration (or simply a configuration) is a subset of F.

Feature models (FMs) are graphical representations that include a decorated feature tree and a set of textual cross-tree constraints which complements information given in the tree. The vertices of the tree are the features (from F), while the edges (in $F \times F$) correspond to refinement or sub-feature (part of) relationships in the domain. Edges can be decorated by a symbol meaning that if the parent feature is selected, the child feature can be selected or not (*optional*). Another symbol indicates that if the parent feature is selected, the child feature is necessarily selected (*mandatory*). Groups of edges rooted in a feature represent: *xor-groups* (if the parent feature is selected, exactly one feature has to be selected in the group), and *or-groups* (if the parent feature is selected, at least one feature has to be selected in the group). Fig. 1 shows a simple FM for e-commerce websites.

Such a software necessarily includes a *catalog* for proposing products, and this catalog is displayed using a *grid* or (exclusive) using a *list*. Optionnally, a *basket* functionality is proposed. A *payment_ method* may also be optionally proposed. Two payment methods are proposed: *credit_ card* or (inclusive) *check*. A cross-tree constraint, written below the tree, indicates that if a basket is proposed, a payment method is also proposed (and reciprocally).

A variability representation conveys *ontological* information (*ontological* semantics): the edges of the feature tree and the groups correspond to domain knowledge, e.g. the group *grid, list* indicates a semantic refinement of catalog; the edge *(e_ commerce, catalog)* indicates that catalog is a subpart of the website.

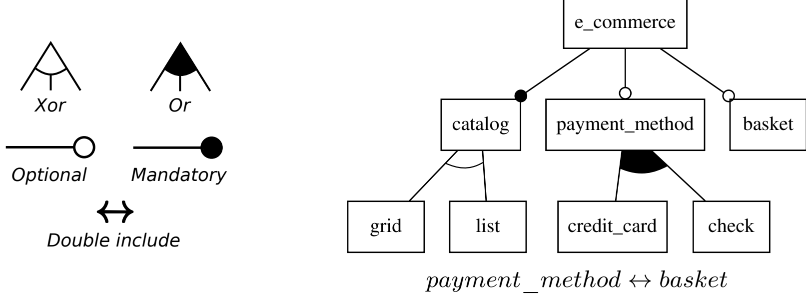

Fig. 1: (Left) Basic feature model's relationships and their corresponding edges; (Right) A basic feature model for an e-commerce website

A variability representation also has a *logical* semantics: for example an alternative representation of the FM is an equivalent propositional formula with $|F|$ variables and constraints defined using propositional connectives ($\wedge, \vee, \rightarrow, \leftrightarrow$ and $\neg$) [5, 8]. Automated analysis can then be performed using SAT-solvers, generally on the Conjunctive or Disjunctive Normal Forms (CNF or DNF). Fig. 2 shows the propositional formula equivalent to the FM of Fig. 1. The down side of the propositional formula is that *ontological* semantics is lost, e.g. an implication in the formula may represent a subpart relationship or a cross-tree constraint.

$$
\begin{array}{ll}
\textit{hierarchy}: & (Ca \rightarrow Ec) \wedge (G \rightarrow Ca) \wedge (L \rightarrow Ca) \wedge \\
 & (Pm \rightarrow Ec) \wedge (Cc \rightarrow Pm) \wedge (Ch \rightarrow Pm) \wedge (B \rightarrow Ec) \wedge \\
\textit{xor-groups}: & (Ca \rightarrow (G \oplus L)) \wedge \\
\textit{or-groups}: & (Pm \rightarrow (Cc \vee Ch)) \wedge \\
\textit{mandatory}: & (Ec \rightarrow Ca) \wedge \\
\textit{cross-tree}: & (Pm \leftrightarrow B)
\end{array}
$$

Fig. 2: Propositional formula corresponding to the feature model of Fig. 1

The third semantics is the *configuration-semantics* that associates to any variability representation the set of its valid configurations. The set of the 8 valid configurations for the FM of Fig. 1 is given in Table 1. For the sake of space, it is shown using the Formal Context representation, which is equivalent.

An important property of a formalism is *canonicity*. Given a set of configurations that are to be represented, and considering a chosen formalism, there are often different ways of writing a representation of a given set of configurations following this formalism. For example different feature models can have the same *configuration-semantics*. *Concision* is also an interesting property: a variability representation can be extensional if it enumerates all the possible configurations, or intensional if it represents these configurations in a more compact way. For example, the formal context of Table 1 is an extensional representation of the

Table 1: Set of valid configurations of the FM of Fig. 1

	e_com.(Ec)	catalog(Ca)	grid(G)	list(L)	pay.met.(Pm)	cred.card(Cc)	check(Ch)	basket(B)
1	x	x	x					
2	x	x		x				
3	x	x	x		x	x		x
4	x	x	x		x		x	x
5	x	x	x		x	x	x	x
6	x	x		x	x	x		x
7	x	x		x	x		x	x
8	x	x		x	x	x	x	x

FM of Fig. 1, whereas the FM is an intensional representation of variability.

In this section, we study graph-like representations which have been used in the literature to express software product line variability of a feature model starting from a propositional formula. For each representation, we give a definition and discuss its canonicity, concision, configuration semantics and ontological semantics.

A **binary decision tree** (BDT) is a tree-like graph used to represent the truth table of a boolean function equivalent to a propositional formula: it is an extensional representation. This representation has redundancies which can be avoided by node sharing, which results in a graph called **binary decision diagram** [8, 6] (BDD): this representation is more concise than the BDT. BDD usually refers to ROBDD (for reduced ordered binary decision diagram), which is unique for a given propositional formula. A BDD depicts the same set of configurations as the original feature model, but the ontological semantics is lost in the transformation. A propositional formula can also be represented as an **implication hypergraph** [8]. As the implication set for a given formula is not necessarily unique (except if it is a canonical basis), neither is the obtained hypergraph. The hypergraph depicts exactly the same configuration set. It also keeps a part of the ontological semantics, as feature groups patterns can be extracted from the hyperedges. Another similar representation is the **implication graph** [8], which only depicts binary implications, and thus does not express feature groups. For a given propositional formula, several implication graphs can be constructed, but two induced structures are unique: the transitive closure and the transitive reduction of the graph. Its configuration semantics is not always the same as the original feature model, because an implication graph can eventually depict more configurations, as it expresses less constraints than the original feature model or propositional formula. Finally, a **feature graph** [19] is a diagram-like representation which seeks to describe all feature models which depict a same set of configurations. Because configuration semantics do not formulate mandatory relationships between features, they are not expressed in feature graphs either. As for FMs, a feature graph is not necessarily unique for a given set of configurations, but the transitive reduction and the transitive closure of the feature graph are canonical. All these representations express variability in a compact way.

Table 2: Properties of the different formalisms

Domain	Representation	Canonicity	Same Conf. Sem. as the FM	Same Logical Sem. as the FM	Ontol. sem.			$F \times F$	$Config. \times F$	$Config. \times Config.$	textual representation	graphical representation	extensional representation	intensional representation
					Groups	include vs. refin.	double include vs. mand							
SPLE	Set of configurations	x	x	x					x		x		x	
SPLE	Feature model		x	x	x	x	x	x			x	x		x
SPLE	Propositional formula		x	x				x			x			x
SPLE	Binary decision tree	x	x	x				x				x	x	
SPLE	Binary decision diagram	x	x	x				x				x		x
SPLE	Implication hypergraph		x	x	x			x				x		x
SPLE	Implication graph (IG)							x				x		x
SPLE	IG $\rightarrow$ Transitive reduction	x						x				x		x
SPLE	IG $\rightarrow$ Transitive closure	x						x				x		x
SPLE	Feature graph (FG)		x	x	x			x			x	x		x
SPLE	FG $\rightarrow$ Transitive reduction	x	x	x	x			x			x	x		x
SPLE	FG $\rightarrow$ Transitive closure	x	x	x	x			x			x	x		x
FCA	**Formal Context**	x	x	x					x		x		x	
FCA	**Concept lattice**	**x**	**x**	**x**				**x**	**x**	**x**		x	x	
FCA	**Labelled feature closed set lattice**	**x**	**x**	**x**	**x**			**x**	**x**	**x**		x	x	

The upper part of Table 2 compares the different formalisms used in SPLE domain with respect to canonicity and their ability to encompass or highlight the different semantics. Besides, it shows which kinds of relationships can be read in the formalism: between features only, between configurations and features, or between configurations. Then it indicates if this is a textual or a graphical formalism, and if this is an intensional or an extensional representation. In SPLE domain, all representations (except the set of configurations and the BDT) consider an intensional point of view with only feature organization. FM is the only representation which clearly expresses all ontological information, but it is not canonical, since many relevant FMs can be built from domain information. Implication hypergraph and feature graph preserve the notion of groups, but refinement and mandatory information of features are lost.

To sum up, these formalisms concentrate on feature organization (except the set of configurations and the BDT), are more or less respectful of initial semantics of the FM they represent and none of them considers a mixed representation of features and configurations. In the next section, we show the benefits of having such a mixed representation and in general, the contributions that a concept lattice based representation may bring to the SPLE domain as a complement to the existing representations.

3 Contribution of concept lattices to variability representation and related work

Formal Concept Analysis [10] provides an alternative framework for variability representation, based on a configuration list, given in the form of a formal context (as in Table 1). Formal objects are the configurations, while formal attributes are the features. Fig. 3 presents the corresponding concept lattice. A concept groups a maximal set of configurations sharing a maximal set of features. In the representation, configurations appear in the lower part of the concepts and are inherited from bottom to top. Features appear in the upper part of the concepts and are inherited from top to bottom. This representation includes the FM, in the sense that if there is an edge indicating F_2 sub-feature of F_1 in the tree, these features are respectively introduced in two comparable concepts $C_2 \leq C_1$, furthermore, the cross-tree constraints are verified by the logic formula that describes the concept lattice.

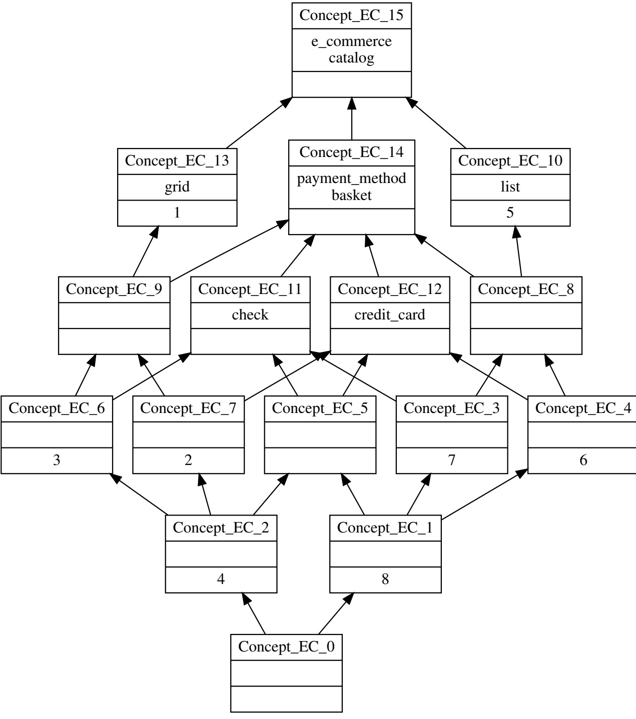

Fig. 3: Concept lattice for the formal context of Table 1, built with RCAExplore[3]

[3] http://dolques.free.fr/rcaexplore.php

A concept lattice organizes configurations and features in a single structure, which has a canonical form (only one concept lattice can be associated with a formal context). If the configurations in the formal context are the valid configurations of a feature model, the configuration semantics of the concept lattice is the same and the configurations can be read from the lattice. The logical semantics is the same too. However, the ontological semantics is incomplete as in the structure, we cannot distinguish ontological relationships: for example, when a feature F_2 is in a sub-concept of a concept that introduces another feature F_1, we cannot know whether F_2 implies F_1 (having a basket *implies* having a payment_method) or F_2 refines F_1 (pay by check *is a kind of* payment_method).

The concept lattice has many qualities regarding the variability representation and relationships between configurations, features, as well as between configurations and features, including highlighting:

- bottom features that are present in all configurations (e.g. `catalog, e_commerce`)
- mutually exclusive features (in concepts whose supremum is the top)
- feature co-occurrence (introduced in the same concept, e.g. `basket` and `payment_method`)
- feature implication (one is introduced in a sub-concept of another one, e.g. `credit_card` implies `basket`)
- how a configuration is closed to or specializes another one, or a merge of other configurations. E.g. `8` is a specialization of `5,6,7`.
- features that are specific to a configuration, or shared by many.

The concept lattice is also an interesting structure to navigate between these features and configurations, and is a theoretical support for association rule extraction, a domain that has not been explored yet in SPLE, as far as we know.

Besides, lattice theory defines irreducible elements, useful for identifying irreducible features and configurations (in a polynomial time), that are used for defining canonical representations of a context or a rule basis. In lattice theory, an element is called join-irreducible if it cannot be represented as the supremum of strictly lower elements. They are easily identifiable in a lattice because they have only one predecessor in lattice transitive reduction. All join-irreducible elements are present in the formal context, so they all correspond to valid configurations.

Research work done in the framework of reverse engineering exploits some of the relevant properties of the concept lattice. Formal Concept Analysis has been used to organize products, features, scenarios, or to synthesize information on the product line. In [13], the authors classify the usage of variable features in existing products of a product line through FCA. The analysis of the concept lattice reveals information on features that are present in all the products, none of the products, on groups of features that are always present together, and so on. Such information can be used to drive modifications on the variability management. In the same range of idea, the authors of [2] explore concept lattices as a way to represent variability in products, and revisit existing approaches to handle variability through making explicit hidden FCA aspects existing in them. The authors of [7] go a step further in the analysis of the usage of FCA, by studying

Relational Concept Analysis (RCA) as a way to analyze variability in product lines in which a feature can be a product of another product family.

Different artifacts are classified in [15]: the authors organize scenarios of a product line by functional requirements, and by quality attributes. They identify groups of functional requirements that contribute to a quality attribute, detect interferences between requirements and quality attributes, and analyze the impact of a change in the product line w.r.t functional requirement fulfillment.

Several proposals investigate with FCA the relationships between features and source code of existing products. References [4, 21] aim at locating features in source code: existing products described by source code are classified though FCA, and an analysis of the resulting concepts can detect groups of source code elements that may be candidates to reveal a feature. In the same idea, traceability links from source code to features are mined in reference [17]. In reference [9], the authors mine source code in order to identify pieces of code corresponding to a feature implementation through an FCA analysis with pieces of source code, scenarios executing those pieces of source code, and features.

FCA is also used in several approaches to study the feature organization in feature models. Concept structures (lattices or AOC-poset) are used to detect constraints in feature models, and propose a decomposition of features into sub-features. The authors of [16] extract implication rules among features, and covering properties (e.g. sets of features covering all the products). References [3, 18] produce logical relationships between the features of a FM, as well as cross-tree constraints.

Concept lattice could also be a tool in the framework of forward engineering, using a transformation chain starting from a FM, building with the existing tools, as FAMILIAR [1], the configuration set (which is equivalent to having a formal context), then building the corresponding lattice. But applying in practice this approach to the FMs repository SPLOT[4] [14], we noticed that tools hardly compute more than 1000 configurations, thus we faced a scaling problem.

4 Addressing scaling issues

4.1 From feature models dependencies to implicative systems

The set of all valid attribute implications of a formal context represent a closure operator, which produces attribute closed sets corresponding to concept intents of the context. The associated attribute closed set lattice is thus isomorphic to the concept lattice of the formal context. It is noteworthy that (1) FMs represent features interaction by graphically depicting a set of features and dependencies between them, and that (2) the set of all valid implications also describes dependencies between attributes (i.e. features). Thus, an analogy can be done between implicative systems and FMs dependencies.

We have previously mentioned a method to build a concept lattice from a FM, which consists in enumerating all the FM configurations (i.e. all combinations of

[4] http://www.splot-research.org/

features w.r.t its dependencies) in a formal context. However, FMs are intensional representations which can potentially depict a large number of configurations, making difficult their enumeration and the context computation. In order to avoid this enumeration, we propose a way to express FMs dependencies as sets of implications $\mathcal{P}(F) \times F$, without building the formal context. We made an experiment in which we generated several FMs of small size (< 10 features) and built their equivalent formal contexts, from which we extracted a complete set of valid implications with the tool Concept Explorer[5] [22]. When comparing these FMs to their corresponding set of implications, we noticed that each type of FM dependency generates the same kind of implications, as presented in Table 3.

Table 3: FM dependencies and their corresponding implications

	Root	Hier.	Opt.	Mand.	Or-group	Xor-group
dependencies	R	A B	A B	A B	A B C	A B C D
impl.	$\emptyset \rightarrow R$	$B \rightarrow A$	None	$A \rightarrow B$	None	$BC \rightarrow ABCD$ $BD \rightarrow ABCD$ $DC \rightarrow ABCD$

The root feature traditionally represents the name of the modeled software system, and thus is present in all configurations. This peculiarity is translated by $\emptyset \rightarrow root$, requiring the presence of *root* in all closed sets. Hierarchy constraints (subpart relationships) require that a child feature can be selected only if its parent feature is already selected, and thus produce a $child \rightarrow parent$ implication. Optional relationships actually express the absence of dependencies between a feature and its child, and do not generate any implication. Mandatory relationships imply that a child feature is necessarily selected with its parent and produce a $parent \rightarrow child$ implication. Or-groups behave as optional relationships with an obligation to select at least one feature: this kind of constraints do not produce any implication. Finally, xor-groups require that two of their features cannot appear together in any configuration: each pair of features thus implies the set of all features.

We can also determine implications for cross-tree constraints, i.e. include and exclude constraints. Let F be the set of all features and $f_1, f_2 \in F$ two features. f_1 *includes* f_2 can naturally be translated by $f_1 \rightarrow f_2$, and f_1 *excludes* f_2 can be translated by $f_1\ f_2 \rightarrow F$, as in xor-groups.

Table 3 thus permits to translate FM dependencies in implicative systems without building a formal context. The fact that the obtained implicative system is exactly the system corresponding to the original FM can be proved by construction. When adding a new feature (resp. feature group) to a FM, this adds

[5] http://conexp.sourceforge.net/

new dependencies which only involve the added feature (resp. feature group) and its parent. It does not change the previous dependencies expressed in the FM, but only adds new ones. In our approach, we first identify the implications corresponding to each type of feature groups and optional/mandatory relationships. Then, if we construct the FM step by step, we can create the implications corresponding to each added feature (resp. feature group), and thus no implication is missing, nor needs to be changed afterward. We applied our method on the FM of Fig. 1 and obtained the implicative system presented in Fig. 4.

root :	$\varnothing \rightarrow Ec$
hierarchy :	$Ca \rightarrow Ec$; $G \rightarrow Ca$; $L \rightarrow Ca$;
	$Pm \rightarrow Ec$; $Cc \rightarrow Pm$; $Ch \rightarrow Pm$; $B \rightarrow Ec$
mandatory :	$Ec \rightarrow Ca$
xor-group :	$G, L \rightarrow Ec, Ca, G, L, Pm, Cc, Ch, B$
cross-tree :	$Pm \rightarrow B$; $B \rightarrow Pm$

Fig. 4: Implicative system corresponding to the feature model of Fig. 1

These implications can be extracted by performing a graph search on the FM, and their number can be predicted by analysing its dependencies (# stands for "number of"):
$1 + \#\textit{child-parent relationships} + \#\textit{mandatory relationships}$
$+\#pairs\ of\ features\ in\ each\ \textit{xor-group} + \#\textit{cross-tree constraints}$

For example, a representative FM of SPLOT (e-commerce) with 19 features and 768 configurations is equivalent (with the configuration-semantics) to an implicative system with 27 implications.

4.2 Identification of the set of possible configurations

In a concept lattice representing a FM, an object introduced in a concept extent represents a valid configuration, which corresponds to the feature set of the concept intent. Because each configuration in a FM is unique, a concept can introduce at most one object. Thus, for SPLE, a concept intent represents either a valid configuration or an invalid one. In the isomorphic feature closed set lattice, each closed set corresponds to a concept intent from the context: therefore, each valid configuration of the FM matches a feature closed set. However, the feature closed set lattice does not display objects and thus we cannot identify which closed set corresponds to a valid configuration. To be able to retrieve knowledge about configurations as in concept lattices, their identification in feature closed set lattice is necessary.

As previously said, all join-irreducibles correspond to valid configurations. But there can exist valid configurations which do not correspond to join-irreducibles, and thus they cannot be discerned from invalid ones in feature closed set lattices. A solution is to add a unique attribute for each configuration, as an

identifier. Thus, we change the lattice structure to make each configuration correspond to a join-irreducible element, which can be detected. However, the obtained lattice is not isomorphic to the original concept lattice, and its size is larger. A way to keep the isomorphism is to add "reducible" attributes, which do not modify the lattice structure and which can label the lattice's elements.

In what follows, we investigate a way to label feature closed sets to help the identification of valid configurations. We seek to produce a labelled implicative system that generates a labelled feature closed set lattice, isomorphic to the concept lattice associated with the formal context. We recall that a valid configuration is a combination of features w.r.t. all the FM dependencies: thus, we seek to retrieve valid configurations by detecting which feature closed sets respect all these dependencies.

Features linked by mandatory relationships always appear together in closed sets: this type of dependencies is respected. Optional relationships express the absence of dependencies and do not create difficulties. Or-groups and xor-groups, however, are more problematic. Let us consider the or-group in Table 3, composed of B and C, which are two sub-features of A. $\{A, B\}$ and $\{A, C\}$ are two valid combinations of features of this group. Because our feature closed set family is closed under intersection/join, $\{A, B\} \cap \{A, C\} = \{A\}$ is also a feature closed set of the family, but it does not respect the dependencies induced by the or-group (i.e. contains at least B or C). The same reasoning can be applied to xor-groups. To identify if a feature closed set respects the dependencies induced by or-groups and xor-groups, we choose to make *constraints* related to feature groups appear directly in feature closed sets, as labels.

Let $\{f_1, \ldots, f_n\}$ be a subset of features involved in a feature group. If they form an or-group, each feature closed set containing the parent feature of this group will be labeled $(f_1, \ldots, f_n)$, defining the constraint: "this feature closed set must have at least one feature from $\{f_1, \ldots, f_n\}$ to correspond to a valid configuration". If they form a xor-group, each feature closed set containing the parent feature of the group will be labeled $[f_1, \ldots, f_n]$, defining the constraint: "this feature closed set must have exactly one feature from $\{f_1, \ldots, f_n\}$ to correspond to a valid configuration". As example, the FM of Fig. 1 produces two different labels: one for the xor-group of the feature *catalog* (Ca), and another for the or-group of the feature *payment_method* (Pm). Each feature closed set possessing *catalog* has to be labeled $[grid, list]$, and each feature closed set possessing *payment_method* has to be labeled $(check, credit_card)$.

We choose to represent these labels in the labelled implicative system as attributes. A label is attached to a feature by adding to the original implicative system a double implication between the feature and the corresponding label-attribute. Fig. 5 presents the implications added to the implicative system of Fig. 4 in order to take into account labels $[grid, list]$ ($[G, L]$) and $(check, credit_card)$ ((Ch, Cc)).

A feature closed set with a $(check, credit_card)$ label is a valid configuration if it contains at least features *credit_card* or *check*. A feature closed set with a

$$labels: \quad \begin{array}{l} Pm \rightarrow (Ch, Cc)\ ;\ (Ch, Cc) \rightarrow Pm\ ; \\ Ca \rightarrow [G, L]\ ;\ [G, L] \rightarrow Ca \end{array}$$

Fig. 5: Adding labels in the implicative system of Fig. 4

$[grid, list]$ label is a valid configuration if it contains *grid* or *list*, but not both. A feature closed set respecting the constraints expressed by all its labels represents a valid configuration. A label is associated with the parent feature of the group, and thus does not change the original lattice structure.

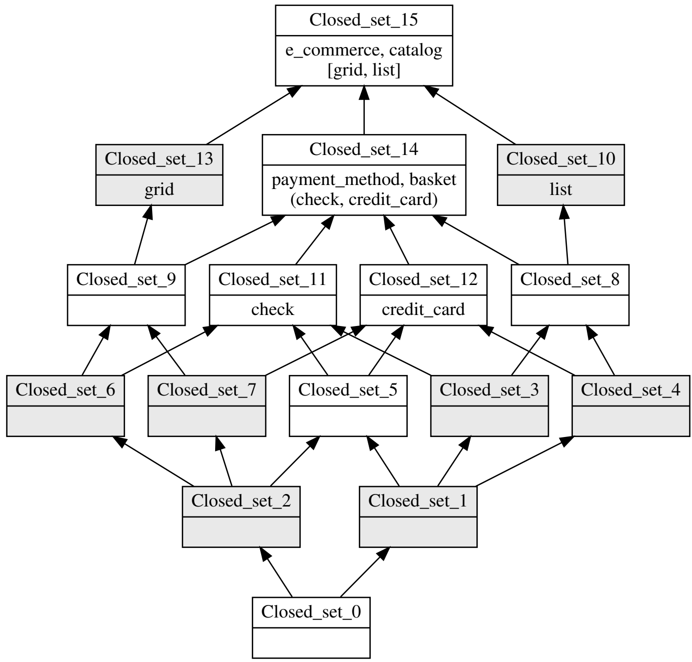

Fig. 6: Feature closed set lattice built with the implicative system of Fig. 4, labeled with implications of Fig. 5

Fig. 6 represents the feature closed set lattice associated with the labeled implicative system of Fig. 4 plus Fig. 5: feature closed sets which respect all the constraints defined by their labels are colored, and correspond to the 8 configurations of the formal context of Table 1. In the lattice, "label features" are inherited from top to bottom, as usual features. For example, *Closed_set_3* possesses features $\{e_commerce, catalog, list, basket, payment_method, check\}$ and the two labels $[grid, list]$ and $(check, credit_card)$. This feature closed set possesses feature *list* and not feature *grid*, and thus respects the constraint of label $[grid, list]$. Moreover, it possesses feature *check*, and thus also respects the

constraint of label ($check, credit_card$). The constraints corresponding to all its labels are respected: *Closed_set_3* is thus a valid configuration of the software product line. Note that in this particular case, the valid configurations are all irreducible.

To conclude, the labelled implicative system permits to construct a lattice from a FM without enumerating all its configurations: the obtained feature closed set lattice is a canonical representation, isomorphic to the concept lattice of a formal context, in which one can retrieve exactly the same information about features and configurations.

5 Conclusion

In this paper, we compare the various formalisms used in the literature to represent and manage the variability of a software product line. Especially, we study their different semantics, their canonicity and the type of information they can highlight. We investigate formal concept analysis and concept lattices to represent a software product line originally described by a feature model. Contrary to FMs, concept lattices represent commonalities and variabilities in a canonical form. Moreover, they permit to extract relationships between features, between features and configurations and between configurations.

Constructing a concept lattice from a FM requires to enumerate all its configurations in a formal context, but this method can be difficult to realize when their number is too high. We propose a method to extract feature implications directly from feature models dependencies. The obtained implicative system produces a feature closed set lattice isomorphic to the concept lattice which can be built from the context. We also propose a method to label these implicative systems in order to identify the set of valid configurations, and thus retrieve the same informations as in concept lattices.

In the future, we will make experiment on the existing FMs repositories in order to assess the size of FMs, implicative systems, and closed set lattices and how frequent are the FMs that have reducible configurations. We will also expand our study to multiple software product lines. We will study relational concept analysis to connect several software product lines represented by concept lattices, and analyze their properties and the issues they permit to answer.

References

1. Acher, M., Collet, P., Lahire, P., France, R.B.: FAMILIAR: A domain-specific language for large scale management of feature models. Sci. Comput. Program. 78(6), 657–681 (2013)
2. Al-Msie 'deen, R., Seriai, A.D., Huchard, M., Urtado, C., Vauttier, S., Al-Khlifat, A.: Concept lattices: A representation space to structure software variability. In: 5th Int. Conf. on Inf. and Comm. Systems (ICICS). pp. 1 – 6 (2014)
3. Al-Msie'deen, R., Huchard, M., Seriai, A., Urtado, C., Vauttier, S.: Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In: 11th Int. Conf. on Concept Lattices and Their Applications (ICFCA). pp. 95–106 (2014)

4. Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: Mining Features from the Object-Oriented Source Code of a Collection of Software Variants Using Formal Concept Analysis and Latent Semantic Indexing. In: 25th Conf. on Soft. Eng. and Know. Eng. (SEKE). pp. 244–249 (2013)
5. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: 9th Int. Conf. on Soft. Product Lines (SPLC). pp. 7–20 (2005)
6. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
7. Carbonnel, J., Huchard, M., Gutierrez, A.: Variability representation in product lines using concept lattices: Feasibility study with descriptions from wikipedia's product comparison matrices. In: Int. Works. on Formal Concept Analysis and Applications, FCA&A 2015, co-located with ICFCA 2015). pp. 93–108 (2015)
8. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: 11th Int. Conf. on Soft. Product Lines (SPLC). pp. 23–34 (2007)
9. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Trans. Softw. Eng. 29(3), 210–224 (2003)
10. Ganter, B., Wille, R.: Formal Concept Analysis – Mathematical Foundations. Springer (1999)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA): Feasibility Study (1990)
12. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE software 19(4), 58–65 (2002)
13. Loesch, F., Ploedereder, E.: Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. In: 11th Eur. Conf. on Soft. Maintenance and Reengineering (CSMR). pp. 159–170 (2007)
14. Mendonça, M., Branco, M., Cowan, D.D.: S.P.L.O.T.: software product lines online tools. In: Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA. pp. 761–762 (2009)
15. Niu, N., Easterbrook, S.M.: Concept analysis for product line requirements. In: 8th Int. Conf. on Aspect-Oriented Software Development (AOSD). pp. 137–148 (2009)
16. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of feature models from formal contexts. In: 15th Int. Conf. on Soft. Product Lines (SPLC) Workshop Proceedings (Vol. 2). p. 4 (2011)
17. Salman, H.E., Seriai, A., Dony, C.: Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In: 14th Conf. on Inf. Reuse and Integration (IRI). pp. 209–216 (2013)
18. Shatnawi, A., Seriai, A.D., Sahraoui, H.: Recovering architectural variability of a family of product variants. In: 14th Int. Conf. on Soft. Reuse (ICSR). pp. 17–33 (2015)
19. She, S., Ryssel, U., Andersen, N., Wasowski, A., Czarnecki, K.: Efficient synthesis of feature models. Information & Software Technology 56(9), 1122–1143 (2014)
20. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Work. IEEE/IFIP Conf. on Soft. Arch. (WICSA). pp. 45–54 (2001)
21. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: 19th Working Conf. on Reverse Engineering (WCRE). pp. 145–154 (2012)
22. Yevtushenko, S.A.: System of data analysis "Concept Explorer". In: 7th national conference on Artificial Intelligence KII-2000. pp. 127–134 (2000)