FCA for Software Product Line representation: Mixing configuration and feature relationships in a unique canonical representation

Jessie Carbonnel, Karell Bertet, Marianne Huchard, Clémentine Nebut

FCA for Software Product Line representation: Mixing configuration and feature relationships in a unique canonical representation

Jessie Carbonnel[a], Karell Bertet[b], Marianne Huchard[a], Clémentine Nebut[a]

[a] *LIRMM, CNRS & Université de Montpellier*
161 rue Ada, 34095 Monpellier Cedex 5, France
[b] *L3i, Université de La Rochelle*
Av. Michel Crépeau, 17042 La Rochelle Cedex 1, France

Abstract

Software Product Line Engineering (SPLE) is a set of methods to help build a collection of software systems which are similar enough to enable appropriate artefact reuse. An important task consists in documenting in variability models the common and variable features which may compose the similar software systems along with compatibility constraints between these features. Several models and formalisms have been proposed to model variability: each one of them has specific properties making it pertinent to support certain management operations, which are of primary importance in SPLE. Switching from one kind of variability model to another is thus important to benefit from a wide range of operations and efficiently manage a software product line. In this paper, we review the various approaches proposed to manage and organise features and product configurations (a product configuration being a chosen subset of features). We discuss the originality of concept lattices, canonical structures presenting a dual view on features and configurations, and the advantages to use these structures as variability representations. Switching from existing variability models to an equivalent concept lattice raises scaling issues related to the size of the needed input dataset and thus hinders their exploitation. We propose an alternative relying on implicative systems and define a transformation method which does not suffer from scaling issues.

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

1. Introduction

Software Product Line Engineering (SPLE) [1] is a set of methods to help build a collection of software systems which are similar enough to enable appropriate artefact reuse. In this way, it optimises development and maintenance costs of the whole collection. To achieve efficient artefact reuse, analysing and modelling the software systems' *variability* is a crucial step. This task mainly consists in documenting the common and variable artefacts which may compose the similar software systems, as well as constraints about the compatibility of the different artefacts. A common way to document variability is to describe software systems depending on a set of high level *features* representing the systems' distinguishable characteristics or behaviours [2, 3] and which are relevant to both designers and end-users. Each feature then matches a set of low level software artefacts having eventually less meanings for the end-user. These features are then organised in models depicting the corresponding artefact compatibilities: a combination of these features satisfying the model's compatibility constraints (called a *valid configuration* of the model) represents a high level description of a functional software system which can be built by combining the corresponding artefacts.

These variability models are central to the SPLE paradigm because they are used to manage, maintain and evolve the collection of software systems through different operations. Several notations and underlying

Email addresses: jcarbonnel@lirmm.fr (Jessie Carbonnel), karell.bertet@univ-lr.fr (Karell Bertet), huchard@lirmm.fr (Marianne Huchard), nebut@lirmm.fr (Clémentine Nebut)

formalisms were proposed to manage both features and configurations. Some formalisms are better adapted to support certain operations, and switching from one formalism to another is thus important to benefit from all existing operations. For instance, feature models, the de facto standard for variability modelling, are graphical and compact representations of all valid configurations: they are a pertinent notation to visualise the existing features and their compatibilities, but support automated analysis with difficulty. In this case, switching to another variability representation such as propositional formulas [4] or constraint programming [5] may be more appropriate. We review in this paper the main proposed notations and formalisms to model variability in terms of features, along with some of their properties and the operations they support: this is our first contribution. Among these variability representations, concept lattices [6] are interesting structures to model variability of a given set of configurations: a concept lattice is a canonical structure and, contrary to the other proposed formalisms, it highlights relationships between the considered configurations, between their features and between configurations and their features. They are unique in the variability modelling landscape because they are the only formalism proposing a dual view on features and configurations, whereas prevalent variability representations usually present only views on features. Due to this singularity, concept lattices and their associated sub-hierarchies provide an original support for operations such as variability restructuring [7], variability model composition [8] or exploratory search in the set of valid configurations [9]. The transformation steps enabling to represent in a concept lattice the variability information of a feature model are then of primary importance but still not studied yet: this is the goal of this paper. A naive solution would be to list the valid configurations of a feature model into a formal context, the input format to build a concept lattice. However, the number of possible valid configurations of a feature model grows exponentially with the number of features it depicts, and therefore restricts this solution to only small feature models having computable and manageable numbers of valid configurations. In other words, the benefits and originality of concept lattices reside in their features $\times$ configurations view, but it thus necessitates to have a list of the set of valid configurations which is impossible in most of real industry cases. Therefore, to use the operations supported by these structures, we need to avoid using input data representing the set of configurations in extension and to favour compact input representations. We propose a solution to avoid scaling issues related to formal context computation by showing how to obtain implicative systems on features to build a feature closed sets lattice without building a formal context: this is our second contribution. We also discuss the fact that these implicative systems are an useful representation of variability. Other scaling issues are the ones related to the concept lattice computation, which can be avoided by using on-demand and local generation algorithms [9, 10]: this is out of the scope of this paper. It is noteworthy that the opposite transformation (from concept lattices to feature models) has been studied several times in the context of reverse engineering feature models from software system descriptions [11, 12, 13].

This paper extends the work proposed in [14], in particular studying the feature model semantics so as to provide elements of proof of the matching between feature models and the extracted implicative systems. An algorithm is also given to extract an implicative system from a feature model. Finally, the size of the different kinds of input datasets (representing closure operators) necessary to build concept lattices and generated from feature models are compared in an experimental study.

The remaining of the paper is organised as follows. Section 2 provides background on software product lines, shows how concept lattices can be used to represent variability, and how Formal Concept Analysis is used for that purpose. In Section 3, the various formalisms to capture variability of a software product line are analysed and compared. Section 4 details the use of implicative systems to face scaling issues related to formal context computation for variability management. Section 5 studies the orders of magnitude of the size of formal contexts and implicative systems built from feature models. It shows that implicative systems are a practicable alternative when all possible objects (here the configurations) cannot be listed to construct formal contexts.

2. Background

In this section, we first present Software Product Line Engineering (SPLE) and the prevailing approaches to model variability (Section 2.2). Then, we define the basic elements needed for understanding Formal

Concept Analysis (FCA) and its associated structures (Section 2.3). Lastly, we review the approaches that use FCA in the domain of Software Product Line Engineering (Section 2.4).

2.1. Software product lines engineering

Software Product Line Engineering (SPLE) [1] is a development paradigm that seeks to produce a set of related and similar software systems as a single entity rather than individually produce each one of them. SPLE is a collection of methods relying on software reuse and mass customisation to lower the development cost and the time-to-market, to increase the quality of the produced software systems, and to broaden the scope of offered products. From an implementation point of view, a Software Product Line (SPL) is based on a common software architecture on which different software artefacts can be plugged depending on user requirements, from what results an independent software system. The Software Product Line is composed of the core architecture, the set of software artefacts that can be plugged on the architecture, and the set of software systems that can be derived from them. A central task of this paradigm is thus to model the SPL *variability*, i.e., what is common, what varies between the different software systems, and how it varies. Variability representations are used, among others, for knowledge representation [15], to define the scope of the product line [16], manage its evolution and maintenance [7], to perform design operations [17, 18] or for product derivation [19].

2.2. Variability modelling

To capture and describe the variability of a software product line, Feature Modelling [16, 2, 20] and Decision Modelling [21, 22] are the two prevailing approaches. Feature-oriented representations rely on a set of distinguishable characteristics, called features, to describe and discriminate the software products. As an example, features for a mail client may include having different *interface*s, proposing *filtering* methods, or offering to the users to define their own *macro*s. Decision-oriented models define a software system through the decisions taken by the user concerning the software composition. For instance, a decision may concern the type of interface proposed by the mail client, or whether the user wants the filtering methods to be run server-side or locally. In what follows, we call a product configuration (or simply a configuration) a description of a software system through a variability modelling approach. In feature modelling, a configuration is represented by a subset of features, whereas in decision modelling it corresponds to a set of user's decisions.

The most commonly used formalisms to represent and organise features are Feature Models (FMs) [16, 2]. They are a family of visual languages that allow to represent 1) a set of features F hierarchically organised in a feature tree, and 2) a set of relationships (also called dependencies or constraints) between these features. These relationships can be expressed by graphical decorations in the feature tree and by additional cross-tree constraints, that can be graphically represented on the tree, or textually expressed separately. The relationships expressed between the features constrain the way they can be combined to compose a software product. In other words, an FM delimits the scope of a product line by representing in a compact and graphical way its set of valid product configurations. The vertices of the tree represent the features (from F), while the edges (in $F \times F$) correspond to refinement or sub-feature relationships in the domain (e.g., *part of*, *implements*). Edges can be decorated by a white disc meaning that if the parent feature is selected, the child feature can be selected or not (*optional*). A black disc indicates that if the parent feature is selected, the child feature is necessarily also selected (*mandatory*). Groups of edges rooted in a feature represent feature groups. A *xor-group*, represented by a non-filled arc across an edge group, states that if the parent feature is selected, exactly one feature has to be selected in the group. An *or-group*, represented by a black-filled arc across an edge group, states that if the parent feature is selected, at least one feature has to be selected in the group. Finally, cross-tree constraints, typically *requires* and *exclude* constraints, can be added to complete the information given in the feature tree. Figure 1 shows a simple FM for mail client websites.

A configuration derived from this FM necessarily includes an *interface* which is either a *webmail* or (exclusive) an *application*. Optionally, a functionality allowing the user to define *macro*s is offered. A *filtering* service may also be optionally proposed. Some filtering services may be run *server* side to prevent the reception of junk mails, or (inclusive) they may be run in *local* on already received mails. Two "require"

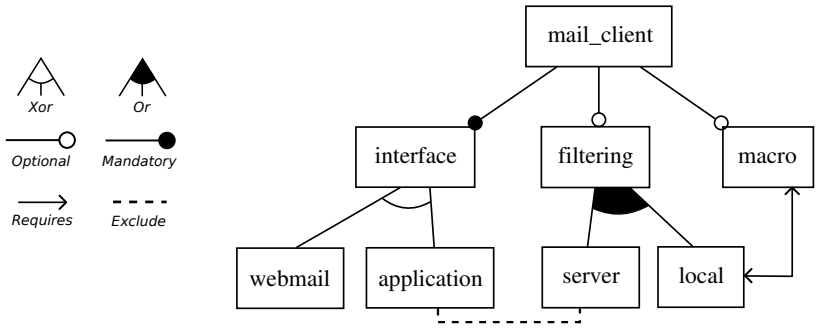


Figure 1: (Left) Basic feature model's relationships and their corresponding graphical representation; (Right) A basic feature model for mail client systems

cross-tree constraints indicate that local filtering services need the macro functionality, and conversely. An "exclude" cross tree constraint states that server side filtering services cannot be proposed on applications.

Decision Models (DMs) are textual representations that seek to document each decision a user can make during the requirement specification phase, when deriving the final software product [21, 22]. DMs are generally depicted in tabular form, where each line of the table describes a decision in the form of a question, and presents the type of the expected answer and its range. It may also present constraints about the required form of the answer (e.g., cardinality) and conditions on when to consider the question. Table 1 presents a decision model about mail clients, inspired from the previous feature model of Figure 1.

Table 1: Decision model for mail clients

decision name	**description**	**type**	**range**	**cardinalities, constraints**	**visibility, relevant if**
filter	support filtering services?	Boolean	{true, false}	/	/
filtering services	proposed filtering services?	Enum	{server, local}	1:2, if *macro* then {local} or {server, local}	$filter$ = true
interface	proposed client interface?	Enum	{webmail, application}	1:1, if $filtering\_services$ $\supseteq$ {server} then {webmail}	/
macro	support user macros?	Boolean	{true, false}	if $filtering\_services$ $\supseteq$ {local} then true	/

While feature models focus on representing the domain by depicting the relevant features and their interactions, decision models seek to simulate the product selection process by listing the set of decisions the user has to make concerning the product to be derived. Thus, DMs guide the user into selecting a final software, but do not necessarily give a correct overview of the product line domain. In fact, even though some parts of the domain can be identified with a decision model (e.g., webmail and application are two possible types of interface, macro is an optional functionality), it does not necessarily cover all the domain information contrarily to a feature model. For instance, one can see in the feature model of Figure 1 that the feature *interface* is a sub-feature of *mail_client* that is mandatory. However, this information does not clearly appear in the decision model of Table 1; because having an interface is mandatory, it does not represent any user decision. Only selecting the type of interface needs a user choice.

In this paper, we focus on the prevalent variability modelling approach that represents all domain infor-

mation: feature modelling. Therefore, in what follows, we work on software configurations represented as a combination of features. We study Formal Concept Analysis, a mathematical framework widely used for knowledge discovery and representation [23], to handle the variability of such software system configurations.

2.3. Concept lattices and Variability Representation

As Formal Concept Analysis (FCA) [6] provides an alternative framework for variability representation, we here present the basic definitions and the qualities of FCA related to this domain. A straightforward application of FCA to variability modelling is based on a valid configuration set, given in the form of a formal context (Def. 1).

Definition 1 (Formal context). A formal context is a 3-tuple $K = (O, A, R)$ where O and A are two finite sets, and $R \subseteq O \times A$ is a binary relation. Elements of O are called objects and elements of A are called attributes. A pair (o, a) of R means "the object o owns the attribute a".

An example of a formal context encoding variability information for the mail client example is given in Table 2. The formal objects are all the valid configurations (lines), while the formal attributes are the features (columns). A concept groups a maximal set of configurations sharing a maximal set of features (Def. 2).

Table 2: Set of valid configurations of the FM of Figure 1 about mail clients (MC)

MC	*mail_client*	*interface*	*webmail*	*application*	*filtering*	*server*	*local*	*macro*
conf_1	×	×	×					
conf_2	×	×		×				
conf_3	×	×	×		×	×		
conf_4	×	×	×		×	×	×	×
conf_5	×	×	×		×		×	×
conf_6	×	×		×	×		×	×

Definition 2 (Concept). Let $K = (O, A, R)$ be a formal context. A concept is a pair (E, I) such that $E \subseteq O$ and $I \subseteq A$. $E = \{o \in O \mid \forall a \in I, (o, a) \in R\}$ is the concept extent and $I = \{a \in A \mid \forall o \in E, (o, a) \in R\}$ is the concept intent. We denote by C_K the set of all concepts of K.

The concepts are provided with a specialisation order (Def. 3), based on extent and intent inclusion.

Definition 3 (Concept Specialisation). Let K be a formal context, and let $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ be two formal concepts of C_K. C_1 is a specialisation of C_2, denoted by $C_1 \leq_K C_2$, if and only if $E_1 \subseteq E_2$ (and equivalently $I_2 \subseteq I_1$). C_1 is called a sub-concept of C_2 and C_2 is called a super-concept of C_1.

The concept set C_K of the formal context K provided with $\leq_K$ is a partially ordered set, which in addition has the property to be a lattice, called the concept lattice (Def. 4, 5 and 6).

Definition 4 (Partially ordered set, (least) upper bound, (greatest) lower bound). A partially ordered set (poset) is a set of elements E provided with a partial order $\leq$ on its elements ($\leq \subseteq E \times E$ is reflexive, antisymmetric and transitive).
An element u is an upper bound of $S \subseteq E$, if for all $s \in S$, $s \leq u$.
An upper bound lub of S is called the least upper bound (or supremum) of S, if for all u upper bound of S, $lub \leq u$. For two elements a and b, we denote by $a \vee b$ the least upper bound of a and b.
An element l is a lower bound of $S \subseteq E$, if for each $s \in S$, $l \leq s$.
A lower bound glb of S is called the greatest lower bound (or infimum) of S, if for all l lower bound of S, $l \leq glb$.

Definition 5 (Lattice). A lattice $(E, \leq)$ is a partially ordered set (poset) where each 2-element subset of E owns a least upper bound and a greatest lower bound for $\leq$.

Definition 6 (Concept lattice). Let C_K be the concept set of the formal context K. The concept lattice of K is the concept set C_K provided with the partial order $\leq_K$, and is denoted by $(C_K, \leq_K)$.

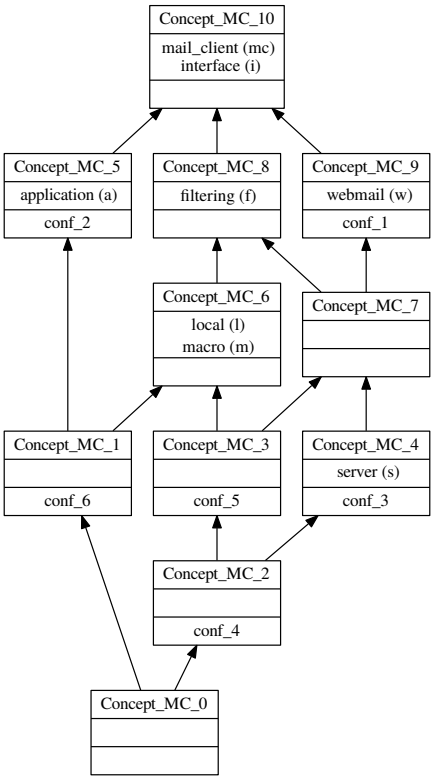


Figure 2: Concept lattice for the formal context of Table 2, built with RCAExplore[1]

Fig. 2 presents the concept lattice associated with the formal context of Table 2. A concept is represented by a 3-compartment box. The upper part contains a concept identifier. Configurations (concept extent) appear in the lower part of the concepts and features appear in the middle part of the concepts (concept intent). In the representation, intents and extents of concepts are simplified by making attributes (features) and objects (configurations) appear only once in the concept lattice. The concepts in which appear features and/or configurations are called "introducer-concepts" (Def. 7 and 8).

Definition 7 (Object-Concept). A concept which introduces a configuration is called an Object-Concept (because configurations are the objects in FCA terms). It is the lowest concept where this configuration appears.

Definition 8 (Attribute-Concept). A concept which introduces a feature is called an Attribute-Concept (because features are the attributes in FCA terms). It is the greatest (highest) concept where this feature appears.

By construction, in a concept lattice, configurations are inherited from bottom to top, and features are inherited from top to bottom.

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

In some situations, rather than using the concept lattice, one can use sub-orders as the AC-poset or the AOC-poset (Def. 9 and 10):

Definition 9 (AC-poset). Given C_K the set of all concepts of a formal context K, the AC-poset (for Attribute-Concepts partially ordered set) is the sub-order of $(C_K,\leq_K)$ restricted to the Attribute-Concepts.

Definition 10 (AOC-poset). Given C_K the set of all concepts of a formal context K, the AOC-poset (for Attribute-Object-Concepts partially ordered set) is the sub-order of $(C_K,\leq_K)$ restricted to the Attribute-Concepts and the Object-Concepts.

The concept lattice can have up to $2^{min(|O|,|A|)}$ concepts in the worst case (when it corresponds to the powerset of the smallest set among O and A), while the concept number of an AC-poset (resp. AOC-poset) is bounded by $|A|$ (resp. $|A| + |O|$).

The concept lattice representation includes the FM, in the sense that if there is an edge indicating F_2 sub-feature of F_1 in the feature tree, these features are respectively introduced in two comparable concepts $C_2 \leq_K C_1$ (C_1 introduces F_1; C_2 introduces F_2). Furthermore, the cross-tree constraints are verified by the logical relationships that can be extracted from the concept lattice (i.e., implications between features, mutual exclusions).

A concept lattice organises configurations and features in a single structure, which has a canonical form: only one concept lattice can be associated with a formal context, and thus from a set of configurations. The valid configurations of a feature model can be read from the lattice, since they correspond to the intent of an Object-Concept.

The concept lattice has many qualities regarding the variability representation and relationships between configurations, between features, as well as between configurations and features. These kinds of information can be useful for several SPLE activities as representing how features may interact or guide the user into selecting a final configuration. A concept lattice may highlight [7, 11, 13]:

- top features that are present in all configurations (e.g. *mail_client*, *interface*);
- mutually exclusive features: two or more features appearing in distinct concepts whose infimum is the bottom (e.g., *application* and *server*);
- feature co-occurrence: features that are introduced in the same concept (e.g., *local* and *macro*);
- feature implication: a feature introduced in a concept implies another one if the latter is introduced in a super-concept (e.g., *server* implies *webmail*);
- how a configuration is closed to or specialises another one, or is a merge of other configurations (e.g., `conf_3` is a specialisation of `conf_1`, `conf_4` is the fusion of `conf_5` and `conf_3`);
- features that are specific to a configuration, or shared by many.
- if the union of the configuration sets (extents) of a set of concepts is equal to the configuration set (extent) of one of their common super-concepts, this may be a group.

The concept lattice is also an interesting structure to support exploratory search through *conceptual navigation* [24, 25]. It is an information retrieval task which assists the users in progressively investigating and discovering the available configurations by navigating (or browsing) into the dataspace by moving from one concept to another according to the features they select. Given a navigation step, the user is placed in a precise concept of the concept lattice structuring the variability. This concept represents the *research state* and depicts the currently selected features. From this concept, the users have to choose between adding or removing features, which corresponds to several navigation choices that will guide them to other concepts. Also, it is noteworthy that FCA is a theoretical support for association rule extraction, a domain that has not been explored yet in SPLE, as far as we know. Association rules are in our case relationships between features that hold for a certain part of the configuration set: they are not true for all the configurations,

but strong rules (i.e., that hold for a major part of the configuration set) can be worth to consider from a variability point of view, as for instance to identify the most prevalent feature choices in the configurations.

Besides, lattice theory defines irreducible elements. An element is called join-irreducible if it cannot be represented as the supremum of strictly lower elements.

Definition 11 (Join-irreducible element). Let $(E, \leq)$ be a lattice, an element $e \in E$ is a join-irreducible element if, for all $a, b \in E$, $x = a \vee b$ implies $x = a$ or $x = b$

The join-irreducible elements are easily identifiable in a lattice because they have only one predecessor in the lattice transitive reduction. In a concept lattice, they necessarily introduce an object. In our framework, they are useful for identifying irreducible configurations (in a polynomial time), that are used for defining canonical representations of a context or a rule basis. All join-irreducible elements are present in the formal context, so they all correspond to valid configurations. Note however that some (reducible) configurations may be introduced in elements that are not join-irreducible.

2.4. FCA in Software Product Line Engineering

Formal Concept Analysis has many applications in software engineering community, as developed in the survey of Tilley et al. [26] in 2005. In the specific domain of software product line engineering, it has been used either in reverse engineering, or in forward engineering.

Forward engineering. Concept lattice can be a tool in the framework of forward engineering in SPLE. Authors of [27] describe the connections between a feature model and a representation of the context in context-aware applications. The context representation is based on an extension of FCA, namely Relational Concept Analysis [28], and it shows how services are connected to their description. The approach uses OWL-DL to encode the feature model, as well as rules in Semantic Web Rule Language (SWRL), to be able to reason about and to adapt configurations

Reverse engineering. Formal Concept Analysis has been used in reverse engineering activities to organise products, features, scenarios, or to synthesise information on the product line. In [7], 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 [29] 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 [30] go a step further in the analysis of the usage of FCA, by studying Relational Concept Analysis (RCA) as a way to analyse variability in interconnected product lines in which a feature of a product line (like a screen for a computer product line) can be a product of another product line (the screen product line).

Different artefacts are classified in [31]: the authors organise 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 analyse 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 software products. References [32, 33] aim at locating features in source code: existing products described by source code are classified through FCA, and an analysis of the resulting concepts can detect groups of source code elements that may be candidates to reveal a feature. Following the same track of research, traceability links from source code to features are mined in reference [34]. In reference [35], the authors mine source code in order to identify pieces of code corresponding to a feature implementation. They analyse through FCA 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 organisation in FMs. Concept structures (concept lattices, AOC-posets, AC-posets) are used to detect constraints in FMs, and propose a decomposition of features into sub-features. References [36, 12, 37] start from configurations and produce logical

relationships between the features of an FM, as well as cross-tree constraints. Ryssel et al. [11] extract implication rules among features, and covering properties (e.g. sets of features covering all the products). Compared to the approach we will develop in this paper, Ryssel et al. start from a formal context and aim to build an FM and a set of implications rules that express the constraints that the FM does not contain; while the approach we will pursue here consists in starting with an FM and providing an alternative representation with an implicative system.

In [38, 8], FCA is used to provide foundations to an approach that builds an intermediary, canonical, fully graphical representation, the Equivalent Class Feature Diagram (ECFD), starting from an FM or from a configuration set, or both. The ECFD can be used either for FM re-design, configuration and feature analysis [38], or FM composition (union and intersection [8]). The approach that we explored in [8] uses a transformation chain which starts from an FM, then builds the configuration set (which is equivalent to having a formal context) with existing tools, as FAMILIAR [39], and lastly builds the corresponding concept lattice or AC-poset. When we applied it in practice to the FMs repository SPLOT[2] [40], we noticed that FAMILIAR hardly computes more than 10.000 configurations. There are different ways to address this scaling problem (that is not encountered when the configuration set is known), including decomposing the FMs, or being able to obtain the concepts without going through the generation of the whole configuration set. This motivated the present work.

3. Comparing Formalisms for Variability Representation

To be able to position FCA conceptual structures among the existing variability representations, in this section we 1) discuss different properties useful to compare variability representations and 2) present the existing formalisms to represent variability in terms of features. We end this section with a comparison of these representations.

3.1. Criteria to Compare the Formalisms

Variability representations describe different kinds of information, and have various properties that can be useful in different situations. For instance, the compact and graphical representation given by an FM is interesting to make an overview of a product line, whereas the textual and tabular representation given by DMs is less suitable for this task. The choice of a variability representation strongly depends on the properties of the model and the operation or task to be realised.

These properties are usually related to the semantics of the representation. A feature-oriented variability representation always has a *configuration semantics* that associates to any variability representation the set of its valid configurations. For instance, the FM of Figure 1 depicts 6 valid configurations, which are given in Table 2. A variability representation is considered `extensional` if it explicitly depicts the set of valid configurations, and `intensional` if it is expressed through constraints between features. `Concision` is an interesting property indicating if the representation suffers from scalability issues due to the growth of its configuration semantics. For example, the formal context of Table 2 is an extensional representation, whereas the FM is an intensional representation of variability. FMs are concise, while configuration sets are not. A variability representation may also be associated with a *logical semantics* expressing with propositional logic the constraints defined over the set of features. Models of the propositional formula thus correspond to the valid configurations. The property indicating if a representation can `express any configuration set` can then be formulated through logical completeness. Comparison of graphical variability model expressiveness based on their logical semantics is addressed in previous work [41], and we only indicate here the result of this previous analysis. We can thus identify formalisms which are able to `preserve FM configurations semantics` after a transformation from an FM notation. A variability representation may convey an *ontological semantics* providing knowledge about the domain of the product line. For instance, in an FM, the `edges` of the feature tree and the `groups` correspond to domain knowledge, e.g. the group {*webmail,*

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

application} indicates a semantic refinement of *interface*; the edge *(mail_client, interface)* indicates that *interface* is a sub-part of the mail client. Co-occurring features may represent `mandatory relationships` or `double require constraints`: this is an example of two ontological relationships represented by the same logical semantics. Another example are binary implications between features, which may represent either `refinement relationships` or `requirement relationships`. *Canonicity* is another interesting property. 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 (i.e., with different *ontological semantics*) can have the same *configuration semantics*. We also document which kinds of entities are involved in the relationships of the variability representation (between features (`F`×`F`), between configurations (`conf` × `conf`) or between features and configurations (`F` × `conf`)) and if the representation is textual of graphical.

Note that a previous comparison of graphical variability models [41] already gathers properties such as intensional/extensional representation, canonicity or logical completeness. In this paper, we recall these results and extend them by taking into account new properties (different ontological semantics, textual and graphical representations, concision) and new variability representations, notably textual ones and implicative systems obtained with the method proposed in the following sections. We also discuss the usage of the different variability representations in the literature.

3.2. A review of alternative representations

FCA conceptual structures allow to highlight variability by organising a set of features in graph-like representations. To be able to situate these structures among existing variability representations, we study other feature-oriented representations which have been used in the literature to express software product line variability. Some are graphical, some are textual, some are intensional or extensional. But they all have in common the fact that they represent a set of configurations, a configuration in this case being a set of features. Therefore, they can all be seen as alternative representations of a configuration set. For each representation, we give a definition, an illustration based on the mail client SPL represented in the FM of Fig. 1 (where the configurations are listed in the formal context of Table 2), and discuss its canonicity, concision, configuration/logical semantics and ontological semantics.

A **binary decision tree** (BDT) represents the truth table of a propositional formula in a binary tree graph. If the propositional formula is in n variables, then a path from the root to a leaf is of size $n + 1$. A path from the root to one leaf represents a variable assignment, or in our case, a configuration. A node has two outgoing edges: a low edge (assigning the variable to 0) and a high edge (assigning the variable to 1). The leaves are labelled by either 1 or 0 depending on the assignment value: a valid configuration is indicated by a leaf labelled by 1, and an invalid configuration by a leaf labelled by 0. BDTs are extensional representations of a configuration sets which suffer from redundancies due to the repetition of the variables representing the features. This representation does not have any ontological information and does not describe any relationships between features or configurations. However they can represent any configuration set. Condensed BDT representations can be obtained through node sharing to avoid redundancies: the resulting representation is a directed acyclic graph called a **binary decision diagram** (BDD) [4, 42]. The term BDD usually refers to ROBDD (for reduced ordered binary decision diagram), which is unique for a given configuration set. They have the same properties than BDTs except that they are intensional representations. The ROBDD corresponding to the configuration set of Table 2 is depicted in Figure 3.

ROBDDs are usually not used as end-user representations. Even though it depicts all valid feature combinations while limiting node redundancy, this representation remains large and convoluted. ROBDDs are rather used as internal representations of variability to support automated analysis as it allows us to perform some operations (as counting the number of valid configurations, for instance) more efficiently than with propositional formulas and SAT solvers [43]. They have been notably used to compute binary implications and prime implicants (used to detect feature groups) during FD synthesis [4, 44, 45].

Another graph-like representation is the **binary implication graph** [4], which only depicts binary implications between features. This is an intensional representation, where each vertex corresponds to a different variable (feature), and each implication is represented by a directed (binary) edge. These binary

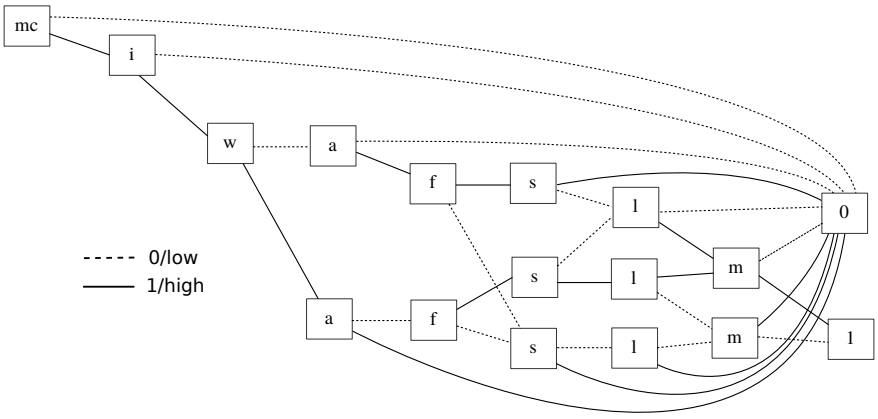


Figure 3: ROBDD corresponding to the configuration set of Table 2

implications correspond the child-parent relationships, the cross-tree constraints and the mandatory relationships: thus, feature groups are not expressed in this representation. 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. The transitive reduction of the implication graph corresponding to our example about mail clients is presented in Figure 4 (left-hand side). 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.

Binary implication graphs were first used in the context of variability modelling by Czarnecki and Wasowski [4] as a propositional formula visualisation in their work on FM extraction from propositional formulas. This representation has been used in other papers to synthesise FMs from a set of specified constraints [46, 47, 44, 48, 45]. Their popularity is due to the fact that the feature tree composing a feature model is a spanning tree of the implication graph, therefore building the implication graph is usually the first step of dedicated algorithms for FM synthesis.

Implication hypergraphs [4] extend the notation of binary implication graphs. A hypergraph is a graph generalisation which may have hyperacs, i.e., arcs connecting more than two vertices. The hypergraph vertices correspond to variables (features) and boolean constants. Each binary implication (cross-tree constraint, child-parent or mandatory relationship) is represented by a directed binary arc, while other constraints (feature groups and mutex) are represented by hyperarcs. If several vertices are connected to another vertex by an hyperarc, it represents a feature group. If two vertices are connected to the vertex representing the constant 0, then they are mutually exclusive. Features involved in a feature group and mutually exclusive represent a xor-group. Figure 4 (right-hand side) represents the implication hypergraph associated with the mail clients. As the implication set for a given formula is not necessarily unique (except if it is a canonical basis), neither is the obtained hypergraph. This intensional representation depicts exactly the same configuration set as the original propositional formula. It also keeps a part of the ontological semantics, as feature groups patterns can be extracted from the hyperarcs.

Despite their mentioned qualities, directed hypergraphs have been presented in [4] as a way to graphically represent a propositional formula for FM extraction, but have not been used further for variability modelling.

A **feature graph** [45] is a diagram-like representation which seeks to describe all feature models which depict a same set of configurations (we recall that for a given set of configurations, several different feature models can be built). A feature graph is composed of a set of vertices (F) corresponding to features and conjunction of features (co-occurring features), and a set of directed edges representing binary implications ($E \subseteq F \times F$), which form a connected directed acyclic graph (DAG). In addition to E, a feature graph possesses a set of undirected edges E_x (with $E_x \cap E = \varnothing$) expressing mutually exclusive features. Finally, G_o, G_x, and G_m are three sets of subsets of E, which indicate respectively edges involved in or-groups, xor-groups and mutex-groups. All edges present in a group of G_o, G_x or G_m have the same parent. Because configuration

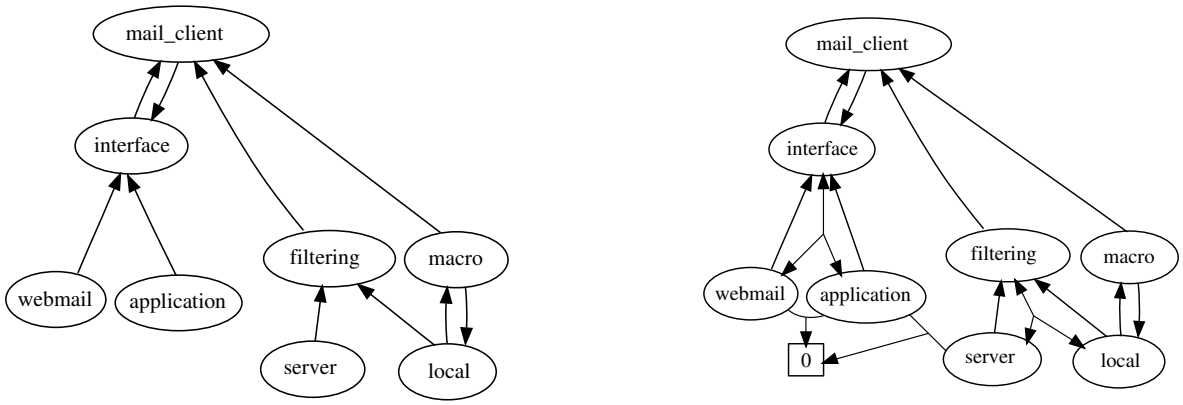


Figure 4: Binary implication graph (left-hand side) and implication hypergraph (right-hand side) of mail clients

semantics do not formulate mandatory relationships between features, they are not expressed in feature graphs either, and thus explain why a vertex of the DAG can either be a single feature, or a conjunction of features. 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. These representations express variability in a compact way.

Equivalence Class Feature Diagrams (ECFDs) [38] are also graphical and compact representations (close to the classical feature model notation) of the logical semantics that can be extracted from a set of configurations. Co-occurring features are introduced in the same box. Arrows between boxes indicate logical implications between features. Mutex are represented by a line marked with a cross between the mutually exclusive features. Feature groups are represented by their respective propositional connectives connecting the features involved in the groups and the parent-feature of the group. They depict the same information as in feature graphs, as well as un-rooted groups of features that never appear together at the same time [38]. These groups are not mutex groups (i.e., each pair of features are in mutual exclusion), and mutex groups are not represented in ECFDs. Originally, they seek to represent more intuitively logical relationships that can be read in a concept lattice structuring a set of configurations, and thus are canonical by construction. In the case of our example, the feature graph and ECFD are equivalent: left-hand side of Figure 5 shows the feature graph notation of Table 2, and right-hand side shows the ECFD notation. In the general case, the ECFD and the feature graph are not feature trees.

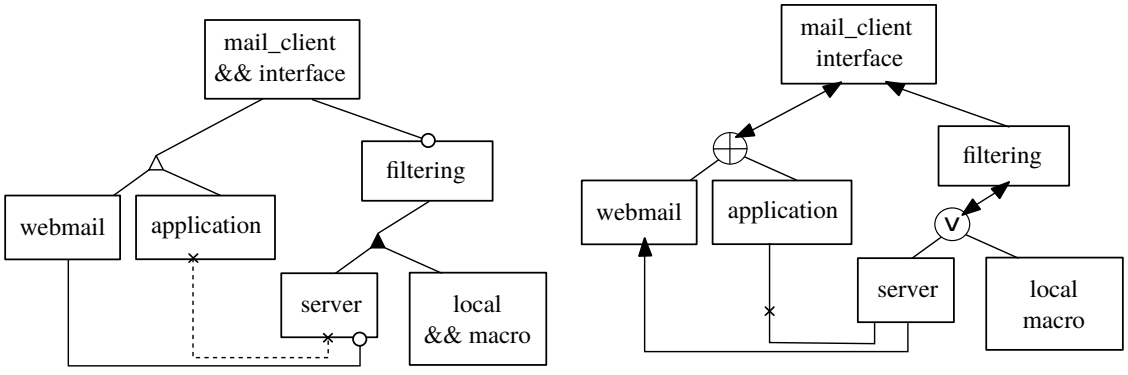


Figure 5: Feature graph (left-hand side) and ECFD (right-hand side)

These variability representations depict the variability information that can be extracted automatically from a configuration set. To obtain an FM from these representations, it is thus necessary to rely on user decisions. Therefore feature graphs and ECFD include all ontological user decision that have to be made by

a designer to synthesize an FM. They are used as "temporary variability representations" in some reverse engineering FM methodologies, and particularly in semi-automated ones [45, 13].

Constraints defined by previous variability representations can be expressed with propositional logic. It is possible to define a **propositional formula** over a set of propositional variables F to express constraints defined using propositional connectives ($\wedge, \vee, \rightarrow, \leftrightarrow$ and $\neg$) [49, 4]. The set of models of the obtained propositional formula then corresponds to the set of valid configurations. Automated analysis (e.g., counting the number of valid configurations, extracting the unused features, verifying the validity of a configuration) can then be performed using SAT-solvers, generally on the Conjunctive or Disjunctive Normal Forms (CNF or DNF). Figure 6 shows a propositional formula equivalent to the FM of Figure 1. The combinations of features respecting the propositional formula of Figure 6 correspond to the 6 configurations listed in Table 2.

$$
\begin{array}{rl}
\textit{root:} & (\top \Rightarrow cm) \\
\textit{hierarchy:} & (i \Rightarrow cm) \wedge (w \Rightarrow i) \wedge (a \Rightarrow i) \wedge \\
 & (f \Rightarrow cm) \wedge (s \Rightarrow f) \wedge (l \Rightarrow f) \wedge \\
 & (m \Rightarrow cm) \wedge \\
\textit{xor-groups:} & (i \Rightarrow (w \oplus a)) \wedge \\
\textit{or-groups:} & (f \Rightarrow (s \vee l)) \wedge \\
\textit{mandatory:} & (cm \Rightarrow i) \wedge \\
\textit{cross-tree constraints:} & (m \Leftrightarrow l) \wedge (s \Rightarrow \neg a)
\end{array}
$$

Figure 6: Propositional formula corresponding to the FM of Figure 1, having for models the ones listed in Table 2

A **Constraint Satisfaction Problem** (CSP) is a 3-tuple (V, D, C), where V is a set of variables, D a set of domains (one for each variable), and C a set of constraints over V. An assignment of values which satisfies all constraints of C represents a solution of the CSP. CSPs have been used to model and reason on SPL, with V representing the set of features, D representing $\{true, false\}$ domains and C modelling the constraints between features [5, 50]. A CSP solution is thus a valid configuration of the SPL. There are several ways to model a set of configurations with a CSP; Fig. 7 presents one of them, following the transformation rules of [50]. As for propositional formulas, CSPs are intensional and textual representations of the SPL.

In [50], the authors propose to compute the number of valid configurations and to list them, to apply filters to retrieve valid configurations satisfying given constraints, and to retrieve valid configurations close to a given feature set. In [51], authors propose to use CSPs to detect dead-features (i.e., unused features which do not appear in any configuration) and core-features (i.e., features present in all valid configurations). Constraints also are used for modelling purpose [52].

- $V = \{cm,\ i,\ a,\ w,\ f,\ s,\ l,\ m\}$
- $\forall d \in D, d = \{true, false\}$
- $C = \{cm = true,\ cm = i,\ i \Leftrightarrow (a \oplus w),\ f \Leftrightarrow (s \vee l),\ m = l, s \Rightarrow \neg a\}$

Figure 7: Constraint Satisfaction Problem

3.3. Feature-oriented representation comparison

The upper part of Table 3 compares the different formalisms used in SPLE domain with respect to properties discussed in Section 3.1.

In SPLE domain, most of the representations consider an intensional and graphical point of view with only feature organisation ($F \times F$). This is due to the fact that the number of potentially valid configurations exponentially grows with the number of features. Thus, to ensure a variability representation with a reasonable size and understandable by practitioners, the majority of formalisms possesses these three properties. The set of configurations and the BDT are the only exceptions concerning intensional feature organisation, as they enumerate the configurations and indicate which features they contain. Also, propositional formulas

Table 3: Properties of the different feature-oriented representation formalisms

		Rep. prop.						conf. sem.					onto. sem.		
Domain	Representation	Canonical	Textual	Graphical	$F \times F$	$F \times conf$	$conf \times conf$	any conf. sem.	preserve FM conf. sem.	extensional	intensional	concision	express feature groups	requires vs refinement	double requires vs mand.
SPLE	Configuration set		x					x	x	x					
SPLE	Feature model			x	x				x		x	x	x	x	x
SPLE	Binary decision tree	x		x				x	x	x					
SPLE	ROBDD	x		x				x	x		x				
SPLE	Binary implication graph (BIG)			x	x						x	x			
SPLE	BIG transitive closure/reduction	x		x	x						x	x			
SPLE	Hypergraph	x		x	x			x	x		x	x	x		
SPLE	Feature graph	x		x	x				x		x	x	x		
SPLE	ECFD	x		x	x				x		x	x	x		
SPLE	Propositional formula		x		x			x	x		x	x	x		
SPLE	CNF/DNF	x	x		x			x	x		x	x			
SPLE	CSP		x		x			x	x		x	x	x		
FCA	Formal context	x	x					x	x	x					
FCA	Concept lattice	x		x	x	x	x	x	x	x	x				
FCA	Labelled feature closed set lattice	x		x	x	x	x	x	x	x	x		x		
FCA	Implicative system		x		x						x	x			
FCA	Labelled implicative system		x		x				x		x	x	x		

and CSPs are the only ones that textually represent intensional feature organisation (apart from serialization formats of tools); as we have seen before, these formalisms are principally used for automated analysis of variability and not for an intuitive overview representation.

FM is the only representation which clearly expresses ontological information, but it is not canonical, since many relevant FMs can be built from a same configuration set. Implication hypergraphs, feature graphs and ECFD preserve the notion of feature groups, but refinement and mandatory information of features are lost, e.g., it is not possible to know from a logical implication between two features if the first feature refines the second one, or if they are semantically independent features but the first requires the second. In fact, these representations display the logical semantics of equivalent feature models, i.e., all feature models depicting the same set of configurations but having different ontological semantics. Thus, they are a way to represent equivalence classes of feature models.

In FCA domain, formal contexts allow to list the configuration set of a software product line in a $configurations \times features$ table. This representation is fully textual and extensional. The configuration semantics of the concept lattice is the same as the one of the FM. 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 (macro *implies* local) or F_2 refines F_1 (local *is a kind of* filtering). However, the application of FCA on a formal context permits to construct the ECFD, a canonical graph-like structure that highlights almost all logical relationships between features (e.g., implications, feature-groups) and relationships between configurations (e.g., specialisation/generalisation). FCA and its associated conceptual structures offer an original mixed representation of features and configurations that may bring to the SPLE domain a complement to the existing representations, by showing relations between features and configurations, as well as between configurations. The cases of labelled feature closed set lattice, implicative systems and labelled applicative systems are detailed in the next section.

The combinatorial explosion of the number of configurations may be responsible of some scaling issues, as it is necessary to list them all to build a formal context. In what follows, we propose a way to avoid scaling issues related to enumerating configurations in formal contexts. This will enable the usage of variability operations based on formal concept analysis structures, which are usually restricted to small feature models.

4. Addressing scaling issues

When using FCA, scaling issues may occur at two levels: when computing the formal context, and when computing the concept lattice. Concept lattices are well-known for their exponential growth and the difficulty one can have to exploit them when their size is too important. One of the existing solutions to limit the structure size is to make an on-demand generation of the concept lattice, using step-by-step algorithms such as the *immediate successor algorithm* [53] to generate only the needed concepts, as for instance in data exploration activities [25]. On-demand generation of a concept lattice sub-hierarchies such as AOC-poset [9] is another solution to avoid scaling issues related to the size of conceptual structures.

In this section, we address scaling issues related to the size of formal contexts (input datasets), especially when it is not possible to list all possible objects (i.e., in our case, to list the valid configurations of an SPL). We investigate implicative systems as an alternative intensional variability representation which does not necessitate to enumerate all possible configurations. This choice is motivated by the fact that implicative systems can be defined on the feature set only. Moreover, several algorithms enable to switch from a representation to another (formal context, concept lattice, implicative system basis) in a canonical way, as they are based on bijections between these objects through a closure operator [54].

4.1. Preliminaries

We first introduce the needed definitions and useful results that underlie our proposal: closure operator, closed element, closure system, implicative system and their connection to concept lattices [6, 55, 56].

A *closure operator* is a mapping over an ordered finite set $(X, \leq)$, which is extensive, increasing and idempotent.

Definition 12 (Closure operator and closed element). A closure operator over a set $(X, \leq)$ is a mapping $\phi : X \mapsto X$ such that, for all $x, y \in X$:

- $x \leq \phi(x)$ (extensive)
- if $x \leq y$, then $\phi(x) \leq \phi(y)$ (increasing)
- $\phi(\phi(x)) = x$ (idempotent)

An element x such that $\phi(x) = x$ is called a closed element.

Here, we will consider, for a finite set E, closure operators over $(2^E, \subseteq)$.

Two different definitions can be found in the literature for closure systems. Here we will define a *closure system* (also called a Moore family) as a set of subsets of a finite set E, which is closed by intersection, and contains the set E. Alternatively, a closure system can be defined by a pair (E, ϕ), where E is a set and ϕ a closure operator.

Definition 13 (Closure system (Moore family), closed set). $\mathcal{F}$ is a closure system over a finite set E if and only if:

- $\mathcal{F} \subseteq 2^E$
- $E \in \mathcal{F}$
- for all $S_1, S_2 \in \mathcal{F}$, we have $S_1 \cap S_2 \in \mathcal{F}$

An element of $\mathcal{F}$ is called an $\mathcal{F}$-closed set (or simply a closed set, when there is no ambiguity).

If $\mathcal{F}$ is a closure system, the partially ordered set $(\mathcal{F}, \subseteq)$ is a lattice. Given a closure operator ϕ over $(2^E, \subseteq)$, the associated closure system is $\mathcal{F} = \{\phi(S) \mid S \subseteq E\}$. Reversely, given a closure system $\mathcal{F}$ over a finite set E, the associated closure operator is $\phi_{\mathcal{F}} : 2^E \mapsto 2^E$, where, for $S_1 \in E$, $\phi(S_1) = \bigcap_{\{S_2 | S_1 \subseteq S_2\}} S_2$.

Now, we connect closure systems and closure operators to concept lattices. Given a formal context $K = (O, A, R)$, we consider the mappings α (which associates the shared attributes to a set of objects) and ω (which associates the common objects to a set of attributes) such that, for $S_O \subseteq O$ and $S_A \subseteq A$:

- $\alpha(S_O) = \{a \in A \mid \forall\ o \in S_O : (o, a) \in R\}$
- $\omega(S_A) = \{o \in O \mid \forall\ a \in S_A : (o, a) \in R\}$

.

Then, the context can be defined by (O, A, α, ω). (α, ω) is a Galois connection [57] and $\alpha \circ \omega$ and $\omega \circ \alpha$ are closure operators. In [6], α and ω are both denoted by $(.)'$, which substantially alleviates the notations. Their compositions are denoted by $(.)''$. The sets S_A", for $S_A \subseteq A$, form a closure system for the attribute set A. They also are the concept intents of the concept lattice of K. Symmetrically, the sets S_O", for $S_O \subseteq O$, form a closure system for the object set O. They also are the concept extents of the concept lattice of K. By construction, the attribute closed set lattice $(\{S_A" \mid S_A \subseteq A\}, \subseteq)$ is isomorphic to the concept lattice of K.

Definition 14 (Implicative systems). Let E be a finite set, an implicative system Σ over E is a set of pairs $(P, C) \in 2^E \times 2^E$. A pair (P, C) is called an implication, P is the premise and C is the conclusion. An implication will be denoted by $P \rightarrow C$.

To an implicative system, one can associate a closure system $\mathcal{F}_\Sigma$ as the set $\{S \subseteq E \mid \forall P, P \subseteq S \textit{ and } P \rightarrow C \in \Sigma \textit{ implies } C \subseteq S\}$, and a closure operator $\phi_\Sigma : 2^E \mapsto 2^E$ where $\phi_\Sigma(S)$ is the smallest subset $S_\Sigma \subseteq E$ such that $S \subseteq S_\Sigma$ and for every $P \subseteq S_\Sigma$ and $P \rightarrow C \in \Sigma$, we also have $C \subseteq S_\Sigma$.
Several implicative systems can be associated with the same closure system, and some have specific properties. For example, an implicative system is *minimal* if no implication can be removed without changing the associated closure system.

Finally, we connect implicative systems and formal contexts. Implicative systems can be defined over the set of attributes (features) or the set of objects (configurations), however, we focus on the ones defined over features, as features are fewer than configurations. Given a formal context $K = (O, A, R)$ and $P_A, C_A \subseteq A$, the implication $P_A \rightarrow C_A$ over A is said valid for the formal context K if and only if $P'_A \subseteq C'_A$ (i.e., if all the objects from K having all the attributes in P_A also have all the attributes in C_A). Let Σ_A be the implicative system constituted of the set of all implications over A that are valid for K, then $\forall S_A \subseteq A, \phi_{\Sigma_A}(S_A) = S''_A$. The attribute closed set lattice $(\{\phi_{\Sigma_A}(S_A) \mid S_A \subseteq A\}, \subseteq)$ is isomorphic to the concept lattice of K, and each $\phi_{\Sigma_A}(S_A)$ for $S_A \subseteq A$ corresponds to a formal concept's intent of K.

4.2. From feature model dependencies to implicative systems

In what follows, we propose a method to avoid scaling issues related to the FCA input formats (i.e., the ones representing a closure operator) that are necessary to build FCA conceptual structures for SPLE purpose. As we have seen before, using formal contexts, that are an extensional representation of variability, as closure operators, is not appropriate in these conditions. In the previous section, we have seen that one can use an implicative system instead of a formal context, and obtain a closed set lattice isomorphic to the concept lattice. Implicative systems over the set of features are more appropriate than formal contexts: they are intensional representation of SPL variability, expressing interactions between features. However, in the associated feature closed set lattice, the feature organisation is preserved, but the objects (i.e., configurations) do not appear anymore.

As we study the case where the set of configurations cannot be enumerated, we have to start from an intensional representation of the SPL variability. Feature models being the *de facto* standard, they will constitute our inputs. Let $K_F = (O_F, A_F, R)$ be the formal context associated with the set of valid configurations of a feature model F, and Σ_{A_F} the set of all implications over A_F that are valid for K_F. The feature closed set lattice of Σ_{A_F} is isomorphic to the concept lattice of K_F, and represents the formal concept's intents of K_F. In what follows, we call Σ_{A_F} a *corresponding implicative system* of the feature model F. More precisely, as the implicative system we consider seeks to represent variability information, we will refer to it as the *corresponding variability implicative system* of the feature model, so as not to be confused with other equivalent implicative systems. Now, our goal can be summarised by the following question:

RQ1: *Given a feature model F, how to obtain the corresponding variability implicative system Σ_{A_F} without having to build the formal context K_F?*

To enable the transformation from an FM to its corresponding variability implicative system, we study more formally the semantics of FM relationships with regards to the configurations they produce in the formal context. More specifically, given a feature model and its corresponding formal context, we study the impact on the formal context of the addition of a feature to the FM. It allows us to deduce which implications over features correspond to each type of FM relationship, and how these relationships modify the existing configurations. The matching between FM relationships, formal contexts and implications is summed up in Table 4. To express FM relationships according to formal contexts, let us consider the formal context K_F as a set of configurations $C_F \subseteq 2^{A_F}$.

- **Root feature** The root feature is a particular case that traditionally represents the name of the modeled product line, and thus it is present in all configurations: let $r \in A_F$ be the root feature of F, $\forall c \in C_F$, we have $r \in c$. This is expressed by the implication $\emptyset \rightarrow r$, requiring the presence of r in all configurations.

- **Feature hierarchy** Refinement relationships defined by the feature tree express the fact that a child feature can be in a configuration only if its parent feature is already part of the configuration. Adding a new feature y as a child of an existing feature x adds a new column y in the associated formal context. Then, all configurations of K_F possessing y will necessarily possess x: $\forall c \in C_F \mid y \in c$, we have $x \in c$. Adding this feature makes the implication $y \rightarrow x$ valid in the formal context. This relationship and its associated implication are implicitly present in the 4 following relationships.

- **Optional relationship** Optional relationships actually express the absence of dependencies between a feature and its child. This means that if x is in a configuration, its child feature y may be in the configuration, or not: $\forall c \in C_F \mid x \in c$, we have $(y \in c \ \vee \ y \notin c)$. Connecting the new feature y by an optional relationship to x duplicates all configurations having x in the formal context, and adds y to these new configurations. It does not generate any implication other than the one corresponding to the hierarchy (i.e., $y \rightarrow x$).

- **Mandatory relationship** Mandatory relationships indicate that if a parent feature is in a configuration, its child feature also has to be. Connecting the new feature y by a mandatory relationship to a feature x means that all configurations of the associated formal context having x now also have y: $\forall c \in C_F \mid x \in c$, we have $y \in c$. The number of configurations does not change, but the implication $x \rightarrow y$ is now valid for K_F.

- **Or-group** An or-group behaves as a set of optional relationships with an obligation to select at least one sub-feature of the group (i.e., the parent feature cannot appear without at least one of its child features in any configuration). Let $\{f_1, ..., f_n\} \subseteq A_F$ be the sub-features involved in the group added to feature x: $\forall T \in 2^{\{f_1,...,f_n\}} \mid T \neq \varnothing$, $\exists c \in C_F \mid x \in c$ and $T \subseteq c$. Connecting the new features $\{f_1, ..., f_n\}$ by an or-group to x duplicates each configuration having x to make them appear $2^n - 1$ times in the updated formal context. $\forall T \in 2^{\{f_1,...,f_n\}}$, T is added to one of the duplicated configurations. As for optional relationships, this kind of constraints does not produce any new valid implication in the formal context.

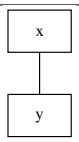
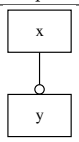
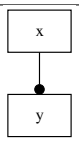
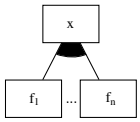
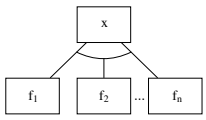
- **Xor-group** Finally, xor-groups require that, if the parent feature is present in a configuration, exactly one feature involved in the group has to be present. In other words, two of the involved sub-features cannot appear together in any configuration: $\forall f_i \in \{f_1, ..., f_n\}$, $\exists c \in C_F \mid \{x, f_i\} \subseteq c$ we have $\nexists f_j \in \{f_1, ..., f_n\} \mid f_j \neq f_i$, and $f_j \in c$. Connecting the new features $\{f_1, ..., f_n\}$ by a xor-group to x duplicates each configuration having x to make it appear n times in the updated formal context. $\forall f_i \in \{f_1, ..., f_n\}$, f_i is added to one of the duplicated configurations. Any pair (f_i, f_j) such that $f_i, f_j \in \{f_1, ..., f_n\}$ *and* $f_i \neq f_j$ does not appear in any configuration, and every pair (f_i, f_j) generates an implication $f_i, f_j \rightarrow \bot$. Therefore, as $A_F \notin C_F$ (because if there exists at least a xor-group, a configuration of the SPL cannot have all the features of A_F), it is denoted by $f_i, f_j \rightarrow A_F$.

- **Cross-tree constraints** We can also determine implications for cross-tree constraints, i.e. *requires* and *exclude* constraints. Let $f_1, f_2 \in A_F$ two features of F. f_1 *requires* f_2 can naturally be translated by the implication $f_1 \rightarrow f_2$. This constraint states that if f_1 is in a configuration, f_2 has to be in the configuration as well: $\forall c \in C_F \mid f_1 \in c$, we have $f_2 \in c$. Adding this constraint removes all the configurations having f_1 but not f_2 from the associated formal context. The constraint f_1 *excludes* f_2 states that if a configuration has f_1, it cannot have f_2, and conversely. It can be translated by $f_1\ f_2 \rightarrow A_F$, as in xor-groups, to express the fact that f_1 and f_2 cannot appear together in the same configuration: $\forall c \in C_F, \{f_1, f_2\} \not\subseteq c$. Adding this constraint to an FM removes from the formal context all configurations in which f_1 and f_2 appear together.

This analysis allowed to highlight this property:

Property 1. When adding a new feature (resp. a feature group) to an FM, it adds new dependencies which only involve the added feature (resp. a feature group) and its parent: it does not change the previous dependencies expressed in the FM.

Table 4: Graphical FM dependencies and their corresponding implications

	Root	Hier.	Opt.	Mand.	Or-group	Xor-group
dependencies	r					
configurations	$\forall c \in C_F$, we have $r \in c$	$\forall c \in C_F \mid y \in c$, we have $x \in c$	$\forall c \in C_F \mid x \in c$, we have $y \in c$ or $y \notin c$	$\forall c \in C_F \mid x \in c$, we have $y \in c$	$\forall T \in 2^{\{f_1,...,f_n\}} \mid T \neq \varnothing$, $\exists c \in C_F$ s.t. $x \in c$ and $T \subseteq c$	$\forall f_i \in \{f_1, ..., f_n\}$, $\exists c \in C_F \mid \{x, f_i\} \subseteq c$ we have $\nexists f_j \in \{f_1, ..., f_n\} \mid f_j \neq f_i$, and $f_j \in c$
impl.	$\emptyset \rightarrow r$	$y \rightarrow x$	None	$x \rightarrow y$	None	$\forall f_i, f_j \in \{f_1, ..., f_n\} \mid f_i \neq f_j$ we have $f_i, f_j \rightarrow A_F$

Therefore, one can transform any FM into its corresponding variability implicative system by performing a graph search on the feature tree and using Table 4. Note that part of the information about feature-groups is lost during the transformation: $x \rightarrow f_1 \vee ... \vee f_n$ (for or-groups) and $x \rightarrow f_1 \oplus ... \oplus f_n$ (for xor-groups) cannot be expressed in implicative systems. We will discuss this aspect later. If one constructs the FM step by step (i.e., starting with only the root feature, then adding each feature or group of features one after another), he/she can create the implications corresponding to each added feature (resp. feature group) as stated in Table 4: no implication is missing, nor needs to be changed afterward.

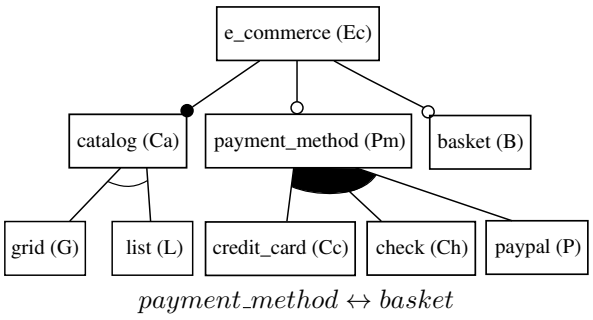
Algorithm 1 shows how to construct the variability implicative system corresponding to an FM. It follows the construction steps of an FM: it starts from the root and goes through all the FM relationships, adding for each one of them the corresponding implications into the output variability implicative system. As we have seen before, the order to process the relationships is not important, as adding a feature to an FM does not impact previously established relationships.

As an example to illustrate our method, we use a variant of an FM about e-commerce taken from [13]. It is presented in Fig. 8 (left-hand side) with the formal context associated with its set of valid configurations (right-hand side). This FM states that all e-commerce applications have a catalog. An e-commerce catalog is either a displayed in a grid or in a list. Payment methods are optional, but if some are implemented, then it is at least one of the following: credit card, check or paypal. A basket management feature is also proposed optionally. If the application has a basket management, then it also has at least one payment method, and conversely.

We applied our algorithm on the FM of Fig. 8 and obtained the variability implicative system presented in Fig. 9.

Data: F a feature model
Result: IS the corresponding variability implicative system
begin
  Add $\{\varnothing \rightarrow root(F)\}$ in IS // root case in Table 4
  $FEAT \longleftarrow GetFeat(F)$ // $FEAT$ set of features of F
  for $f \in FEAT$ **do**
    $S_h \longleftarrow children(f)$
    for $s \in S_h$ **do**
      Add $\{s \rightarrow f\}$ in IS // hierarchy case in Table 4
    end
    $S_m \longleftarrow mandatoryChildren(f)$ // get children of f connected by a mandatory relationship
    for $s \in S_m$ **do**
      Add $\{f \rightarrow s\}$ in IS // mandatory case in Table 4
    end
    $S_x \longleftarrow xorGroups(f)$ // get xor-groups having f for parent-feature
    for $S \in S_x$ **do**
      for $P \in pairsOf(S)$ **do**
        Add $\{P \rightarrow FEAT\}$ in IS // xor-group case in Table 4
      end
    end
    $S_i \longleftarrow requiresCTCs(FM)$ // requires CTCs
    for $(f_1, f_2) \in S_i$ **do**
      Add $\{f_1 \rightarrow f_2\}$ in IS
    end
    $S_e \longleftarrow excludeCTCs(FM)$ // exclude CTCs
    for $\{f_1, f_2\} \in S_e$ **do**
      Add $\{\{f_1, f_2\} \rightarrow FEAT\}$ in IS
    end
  end
end

Algorithm 1: Extracting the corresponding variability implicative system from an FM



	Ec	Ca	G	L	Pm	Cc	Ch	P	B
1	×	×	×						
2	×	×		×					
3	×	×	×		×	×			×
4	×	×	×		×		×		×
5	×	×	×		×			×	×
6	×	×	×		×	×	×		×
7	×	×	×		×		×	×	×
8	×	×	×		×	×		×	×
9	×	×	×		×	×	×	×	×
10	×	×		×	×	×			×
11	×	×		×	×		×		×
12	×	×		×	×			×	×
13	×	×		×	×	×	×		×
14	×	×		×	×		×	×	×
15	×	×		×	×	×		×	×
16	×	×		×	×	×	×	×	×

Figure 8: FM about e-commerce [13] (left-hand side), and its set of valid configurations (right-hand side)

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

Figure 9: Implicative system corresponding to the feature model of Fig. 8

4.3. Identifying the configurations in feature closed set lattices

We have seen that we can build a feature closed set lattice from a variability implicative system directly extracted from an FM, to avoid scaling issues related to enumerating the FM configurations in a formal context. The obtained feature closed set lattice is isomorphic to the concept lattice associated to the FM configurations. However, it does not capture the configurations, which are necessary to perform tasks as exploratory search or variability reorganisation. This leads us to a second research question:

RQ2: *How to identify the FM valid configurations in the feature closed set lattice?*

First, let us recall that in a concept lattice representing an FM, an object introduced in the extent of a concept represents a valid configuration, which corresponds to the combination of features in the concept's intent. Because each configuration in an FM is unique, each concept of the corresponding concept lattice can introduce at most one configuration. Thus, each concept's intent represents either a valid configuration or an invalid one[3].

Now, let us examine the feature closed sets obtained from an implicative system corresponding to an FM. We have seen previously that in the isomorphic feature closed set lattice, each closed set corresponds to a concept's intent from the context: therefore, each valid configuration of the FM matches a feature closed set. But, some feature closed sets do not represent an FM valid configuration (the ones that correspond to concepts of the concept lattice that do not introduce a configuration). To be able to retrieve knowledge about configurations as in concept lattices, identifying which closed set corresponds to a valid configuration in the feature closed set lattice is necessary.

We recall that, from an FM point of view, a valid configuration is a combination of features respecting all the FM dependencies. Therefore, a feature closed set represents a valid configuration if it respects all the FM dependencies. But, only some of the FM dependencies naturally appear in the structure of the lattice. All implications between features are highlighted in the lattice, and thus feature hierarchy, *requires*

[3]Not in the sense where there are incompatible features: these intents represent incomplete configurations, i.e., they can become valid by adding features. The bottom concept is an exception as it is the only one that may represent incompatible features, in the case where no configuration has all features.

cross-tree constraints and mandatory relationships are respected by all feature closed sets of the lattice. Mutually exclusive features also can be retrieved by computing the lower bound of two concepts.

As optional relationships express the absence of dependencies, they do not create any difficulty. Or-groups and xor-groups, however, are more problematic; in fact, some feature closed sets do not respect the feature-groups semantics. For instance, let us consider the or-group in Fig. 8 (left-hand side), composed of *credit_card*, *check*, and *paypal*, which are three sub-features of *payment_method*. $\{Ec, Ca, G, Pm, Cc\}$ and $\{Ec, Ca, G, Pm, P\}$ are two valid combinations of features of this group. But, because our closure system is closed under intersection, $\{Ec, Ca, G, Pm, Cc\} \cap \{Ec, Ca, G, Pm, P\} = \{Ec, Ca, G, Pm\}$ is also a feature closed set of the closure system, but it does not respect the dependencies induced by the or-group (which consists in containing at least Cc, Ch or P). The same reasoning can be applied with xor-groups.

To identify if a feature closed set respects the dependencies induced by or-groups and xor-groups, we choose to make labels appear in the feature closed sets. These labels will play the role of constraints over features: if a feature closed set contains a label, and if it respects the constraint indicated by the label, then it is a valid configuration. Thus, a post-treatment will be needed to verify the constraints represented by the labels in order to identify the valid configurations. There are two kinds of labels to introduce: one to indicate or-groups related constraints, and one to indicate xor-groups related constraints.

- **Xor-group label** This kind of label has to appear in each feature closed set having a feature being the parent of a xor-group. It has to indicate the sub-features involved in the xor-group, that we choose to represent as follows: given a xor-group $xor(p, \{f_1, ..., f_n\})$ with p being the parent feature and $\{f_1, ..., f_n\}$ being the sub-features involved in the group, the corresponding label is $(f_1 \oplus ... \oplus f_n)$ and has to appear in all feature closed sets having p. For instance, the label corresponding to the xor-group of FM from Fig. 8 is $(G \oplus L)$, and has to appear in all feature closed sets having the feature Ca. The constraint represented by this kind of label means "any feature closed set having the label $(f_1 \oplus ... \oplus f_n)$ has to possess exactly one feature of $\{f_1, ..., f_n\}$". If it is not the case, the feature closed set is not a valid configuration of the SPL.

- **Or-group label** This kind of label has to appear in each feature closed set having a feature being the parent of an or-group. It has to indicate the sub-features involved in the or-group, that we choose to represent as follows: given an or-group $or(p, \{f_1, ..., f_n\})$ with p being the parent feature and $\{f_1, ..., f_n\}$ being the sub-features involved in the group, the corresponding label is $(f_1 \vee ... \vee f_n)$ and has to appear in all feature closed sets having p. For instance, the label corresponding to the or-group of FM from Fig. 8 is $(Cc \vee Ch \vee P)$, and has to appear in all feature closed sets having the feature Pm. The constraint represented by this kind of label means "any feature closed set having the label $(f_1 \vee ... \vee f_n)$ has to possess at least one feature of $\{f_1, ..., f_n\}$".

The number of different labels necessary to label the feature closed set lattice is equal to the number of different feature-groups. A feature closed set which respects all the constraints depicted by its labels thus corresponds to a valid configuration.

To make these labels appear in the feature closed sets, we represent them in the labelled variability implicative system, as *label-features*. A label appears in all feature closed sets having a feature f by adding to the original variability implicative system an implication between the feature f and the corresponding label-feature. We also add an implication from the label-feature to f to make them always appear together in the labelled feature closed sets: in this way, the added label-feature does not change the lattice structure.

Fig. 10 presents the implications added to the variability implicative system of Fig. 9 in order to take into account labels $(G \oplus L)$ and $(Cc \vee Ch \vee P)$.

$$\begin{array}{ll} \textit{labels:} & Pm \rightarrow (Cc \vee Ch \vee P)\ ;\ (Cc \vee Ch \vee P) \rightarrow Pm \\ & Ca \rightarrow (G \oplus L); (G \oplus L) \rightarrow Ca \end{array}$$

Figure 10: Adding label-feature in the implicative system of Fig. 9

Fig. 11 represents the feature closed set lattice associated with the labelled variability implicative system of Fig. 9 plus Fig. 10: feature closed sets which respect all the constraints defined by their labels are colored

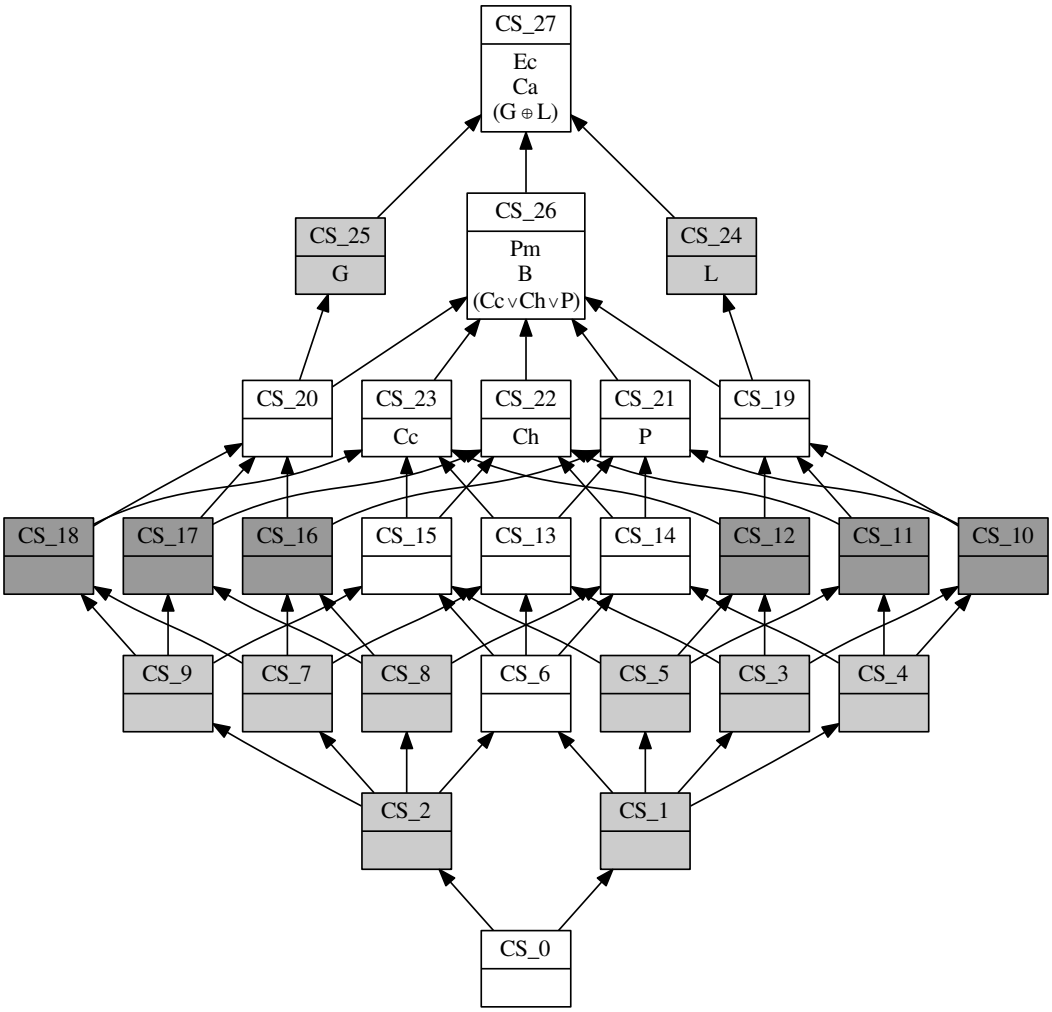


Figure 11: Feature closed set lattice built with the implicative system of Fig. 9, labelled with implications of Fig. 10

in gray, and correspond to the 16 configurations of the formal context of Fig. 8. For example, *CS_12* possesses features $\{Ec, Ca, L, Cc, Pm, B\}$ and the two label-features $(G \oplus L)$ and $(Cc \vee Ch \vee P)$. This feature closed set possesses feature L and not feature G, and thus respects the constraint of the label-feature $(G \oplus L)$. Moreover, it possesses feature Cc, and thus also respects the constraint of the label-feature $(Cc \vee Ch \vee P)$. The constraints corresponding to all its labels are respected: *CS_12* is thus a valid configuration of the software product line.

Light gray feature closed sets represent join-irreducible elements, i.e., feature closed sets having only one predecessor in the lattice (See Section. 2.3). Each join-irreducible feature closed set corresponds to a configuration, thus, it is not necessary to use the label-features in this case. However, all configurations do not correspond to a join-irreducible feature closed set; they are represented in Fig. 11 by dark gray elements. Using the information given by join-irreducible feature closed sets thus reduces the effort of configuration identification.

To conclude, the labelled variability implicative system is equivalent to a closure operator, which permits to construct a lattice from an 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, with a post-treatment.

5. Experimentations

In this section, we conduct an empirical experiment to get an order of magnitude of the size of the two FCA inputs studied in this paper, when they are built from feature models: the formal context based on the set of valid configurations, and the labelled variability implicative system obtained with the method presented in the previous section. Here, we only focus on the size of two possible inputs of FCA algorithms, and we do not study the size of the outputs (i.e., the conceptual structures), because we are targeting on-demand generation algorithms for the conceptual structures in complex cases. In what follows, we define the size of a formal context by its number of objects (i.e. valid configurations) and the size of an implicative system by its number of implications.

The size of the variability implicative system (V.IS) corresponding to a feature model can be predicted by analysing the feature model dependencies: Fig. 12 (1) shows how to calculate the number of its implications ("#" stands for "number of"). In order to label a variability implicative system with the approach presented in Section 4.3, we have seen that one has to add two implications for each feature-group, i.e. two implications for each label-feature. Fig. 12 (2) shows how to calculate the size of a labelled variability implicative system.

$$
\begin{aligned}
|V.IS| = \quad & 1 \text{ (root)} \\
& + \#\text{child-parent relationships} \\
& + \#\text{mandatory relationships} \\
& + \#\text{pairs of features in each xor-group} \\
& + \#\text{cross-tree constraints} \\
|\textit{Labelled } V.IS| = \quad & |V.IS| + (2 \times \#\text{feature-groups})
\end{aligned}
$$

Figure 12: Number of implications in the variability implicative system and the labelled variability implicative system corresponding to an FM

Our experiment is based on feature models that we have collected on SPLOT. The SPLOT project [40] (for Software Product Line Online Tools) is a website which offers the possibility to create, edit, share and analyse feature models online. It provides a repository of 900+ feature models[4], along with severals tools for automated analysis, as debugging and counting valid configurations. All feature models found on SPLOT repository are not necessarily representative of a software product line: this is due to the fact that the repository is sustained by the community which can create any type of feature models. Therefore, we have selected 20 representative feature models which (1) correspond to a software product line and (2) possess at least one valid configuration: they are listed in Table 5.

Table 5 displays 20 selected feature models against some of their characteristics. For each feature model, we first report their number of features, optional and mandatory relationships, feature groups and cross-tree constraints: this type of information was directly extracted from SPLOT repository. Then, we indicate the number of configurations depicted by these FMs, which permits to know the size of the corresponding formal context. The number of configurations has been computed using the BDD engine provided by SPLOT. Finally, we display information related to their corresponding variability implicative systems: the number of necessary label-features (i.e. #feature groups = #or-groups + #xor-groups), the number of implications of their corresponding variability implicative systems, and the number of implications of their corresponding labelled variability implicative systems. Even with average size feature models possessing between 30 and 50 features, we can see in Table 5 that the number of valid configurations is tremendous compared to the number of implications obtained with our method. For example, the feature model about Java EE 6 possesses 45 features and depicts 1.05×10^9 different configurations, while it only produces 78 implications.

We compare the evolution of the size of these two formalisms in the scatter plot of Fig. 13, using data from Table 5. We display the size of formal contexts and labelled variability implicative systems (Y axis, logarithmic scale) depending on the number of features in the feature models (X axis, linear scale).

[4]Last accessed in September 2017

Table 5: Characteristics of 20 representative FMs from SPLOT

Feature Model	# features	# opt. rel.	# mand. rel.	# or-groups	# xor-groups	# CTCs	# conf.	# labels	|V.IS|	|Labelled V.IS|
eshop	10	2	3	1	1	2	9	2	16	20
Mobile Games	16	10	1	1	0	1	3645	1	18	20
Web Game	16	5	6	1	1	2	84	2	24	28
Smart Home	22	5	3	5	0	2	8480	5	27	37
MobileApps Multiplatform	29	12	5	1	1	8	355688	2	45	49
Automotive system	31	3	8	1	7	9	1344	8	61	77
Robot Calibration	33	0	10	1	7	11	648	8	71	87
Online-book-shopping	36	2	21	0	5	3	90	5	59	69
Software Stack	37	2	1	3	8	6	17073	11	67	89
Linux	41	25	6	1	2	7	1.99E7	3	60	66
Obstacle identification	42	0	8	1	9	13	19152	10	111	131
Java EE 6	45	11	0	7	4	3	1.05E9	11	56	78
e-commerce	56	7	5	11	1	5	8.25E11	12	69	93
E-science application	61	7	1	5	11	2	4.72E8	16	105	137
ClassicShell	65	16	3	0	14	11	3.59E8	14	152	180
DATABASE_TOOLS	70	20	3	6	1	2	9.84E16	7	90	104
Car Selection	72	10	14	3	16	21	3E8	19	141	79
Billing	88	45	11	1	1	59	3.87E12	2	161	165
BankingSoftware	176	77	46	6	10	4	5.26E31	16	266	298
Electronic Shopping	290	82	75	40	0	21	4.52E49	40	386	386

Theoretically, the size of the formal context in our case grows exponentially with the number of features (as it represents the number of valid configurations of an FM), and the size of the labelled variability implicative system is polynomial (each FM relationship produces an implication, except for xor-groups that produce $n \times (n-1)/2$ implications. We can see on Figure 13 that the number of configurations, and thus the size of the formal context, exponentially increases with the number of features on the selected datasets. However, this is not the case with the size of the labelled variability implicative systems, which almost linearly grows with the number of features on the selected datasets.

To conclude, our method extracts from feature models labelled implicative systems which do not suffer of exponential growth, contrary to formal contexts which require the list of all valid configurations. Knowing that actual step-by-step algorithms have to access a closure operator for each concept they construct, labelled implicative systems remain relatively practicable compared to formal contexts, and represent an interesting alternative for lattice generation.

6. Conclusion

In this paper, we investigated formal concept analysis and its associated conceptual structures to organise and represent software product line variability. We showed that FCA permits to structure a set of software configurations depending on the features they share in canonical graph-like representations, that naturally highlight information about variability. Besides, we compared concept lattices with the different formalisms found in the literature that are used to depict variability in terms of features. Especially, we studied their different semantics, their canonicity, their concision, and the type of variability information they can express. We showed that concept lattices present some originality, as for instance $configurations \times configurations$

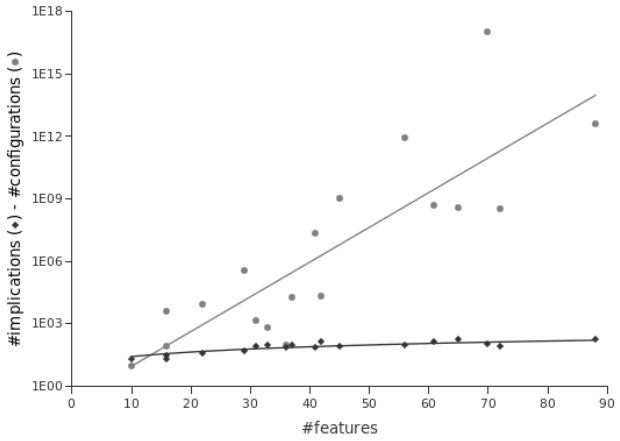


Figure 13: Evolution of the size of formal contexts (●) and labelled implicative systems (◆) built from features models depending on the number of features

relationships and $configurations \times features$ relationships. We deem that they could be used to perform novel kinds of approaches in the domain of software product line engineering.

A way to construct a concept lattice from an FM is to list all the FM valid configurations in a formal context. To avoid scaling issues related to the enumeration of FM configurations needed for FCA inputs, we proposed a method to extract a set of feature implications directly from the set of compact relationships expressed in an FM. The obtained implicative system permits to build a lattice structure, called a feature closed set lattice, that is isomorphic to the concept lattice associated with the FM, but that only contains the SPL features. Then, we proposed a second method that consists in introducing feature-labels inspired from FM relationships in the implicative system to help identify the set of valid configurations. Therefore, we obtained the same information as in concept lattices but without having to build a formal context, without suffering from the combinatorial explosion of the number of SPL configurations. Step-by-step algorithms can then be applied on the labelled implicative system to construct parts of the final structure, thus avoiding scaling issues related to FCA output. We empirically determined that labelled implicative systems have a size which grows linearly with the number of features, contrarily to formal contexts whose size grows exponentially with the number of features.

In the future, we will study algorithms for on-demand generation of FCA structures, and their application to closure operators as labelled implicative systems. 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 analyse their properties when used for more complex representations, for instance in the context of system-of-systems or ultra-large scale systems.

References

[1] K. Pohl, G. Böckle, F. J. van der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer Science & Business Media, 2005.

[2] K. C. Kang, J. Lee, P. Donohoe, Feature-oriented product line engineering, IEEE software 19 (4) (2002) 58–65.

[3] S. Apel, D. S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines - Concepts and Implementation, Springer, 2013.

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

[5] D. Benavides, P. Trinidad, A. R. Cortés, Using Constraint Programming to Reason on Feature Models, in: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), 2005, pp. 677–682.

[6] B. Ganter, R. Wille, Formal concept analysis - mathematical foundations, Springer, 1999.

[7] F. Loesch, E. Ploedereder, Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), 2007, pp. 159–170.

[8] J. Carbonnel, M. Huchard, A. Miralles, C. Nebut, Feature Model Composition Assisted by Formal Concept Analysis, in: Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'17), 2017, pp. 27–37.
[9] A. Bazin, J. Carbonnel, G. Kahn, On-demand generation of aoc-posets: Reducing the complexity of conceptual navigation, in: Proceedings of the 23rd International Symposium on Foundations of Intelligent Systems (ISMIS'17), 2017, pp. 611–621.
[10] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, L. Lakhal, Computing iceberg concept lattices with Titanic, Data Knowledge Engineering (DKE) 42 (2) (2002) 189–222.
[11] U. Ryssel, J. Ploennigs, K. Kabitzsch, Extraction of feature models from formal contexts, in: Proceedings of the 15th International Conference on Software Product Lines (SPLC'11), 2011, p. 4.
[12] R. Al-Msie'deen, M. Huchard, A. Seriai, C. Urtado, S. Vauttier, 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), 2014, pp. 95–106.
[13] J. Carbonnel, M. Huchard, C. Nebut, Modelling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions, Journal of Systems and Software 152 (2019) 1 – 23.
[14] J. Carbonnel, K. Bertet, M. Huchard, C. Nebut, FCA for Software Product Lines Representation: Mixing Product and Characteristic Relationships in a Unique Canonical Representation, in: Proceedings of the 13th International Conference on Concept Lattices and Their Applications (CLA'16), 2016, pp. 109–122.
[15] K. Czarnecki, C. H. P. Kim, K. T. Kalleberg, Feature models are views on ontologies, in: Proceedings of the 10th International Software Product Line Conference (SPLC'06), 2006, pp. 41–51.
[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA): Feasibility Study.
[17] M. Acher, P. Collet, P. Lahire, R. B. France, Composing feature models, in: Revised Selected Papers of the 2nd International Conference on Software Language Engineering (SLE'09), Vol. 5969 of LNCS, Springer, 2010, pp. 62–81.
[18] M. Acher, P. Collet, P. Lahire, R. B. France, Comparing approaches to implement feature model composition, in: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'10), 2010, pp. 3–19.
[19] A. Jansen, R. Smedinga, J. van Gurp, J. Bosch, First class feature abstractions for product derivation, IEE Proceedings - Software 151 (4) (2004) 187–198.
[20] J. Van Gurp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001, pp. 45–54.
[21] K. Schmid, I. John, A customizable approach to full lifecycle variability management, Science of Computer Programming 53 (3) (2004) 259–284.
[22] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches, in: Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12), ACM, 2012, pp. 173–182.
[23] J. Poelmans, P. Elzinga, S. Viaene, G. Dedene, Formal concept analysis in knowledge discovery: a survey, in: Proceedings of the 8th International Conference on Conceptual Structures (ICCS'10), Vol. 6208 of Lecture Notes in Computer Science, Springer, 2010, pp. 139–153.
[24] R. Godin, E. Saunders, J. Gecsei, Lattice model of browsable data spaces, Information Sciences 40 (2) (1986) 89–116.
[25] S. Ferré, Reconciling expressivity and usability in information access : from file systems to the semantic web, Habilitation à diriger des recherches en informatique, Université de Rennes 1 (2014).
[26] T. Tilley, R. Cole, P. Becker, P. W. Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities, in: Formal Concept Analysis, Foundations and Applications, Vol. 3626 of Lecture Notes in Computer Science, Springer, 2005, pp. 250–271.
[27] A. M. Amja, A. Obaid, H. Mili, P. Valtchev, Linking Relational Concept Analysis and Variability Model within Context Modeling of Context-Aware Applications, in: Proceedings of the 2nd IEEE International Symposium on Systems Engineering (ISSE'16), 2016, pp. 413–420.
[28] M. R. Hacene, M. Huchard, A. Napoli, P. Valtchev, Relational concept analysis: mining concept lattices from multi-relational data, Annals of Mathematics and Artificial Intelligence 67 (1) (2013) 81–108.
[29] R. Al-Msie 'deen, A.-D. Seriai, M. Huchard, C. Urtado, S. Vauttier, A. Al-Khlifat, Concept lattices: A representation space to structure software variability, in: Proceedings of the 5th International Conference on Information and Communication Systems (ICICS'14), 2014, pp. 1–6.
[30] J. Carbonnel, M. Huchard, A. Gutierrez, Variability Representation in Product Lines using Concept Lattices: Feasibility Study with Descriptions from Wikipedia's Product Comparison Matrices, in: Proceedings of the 1st International Workshop on Formal Concept Analysis and Applications (FCA&A'15) co-located with the 13th International Conference on Formal Concept Analysis (ICFCA'15), 2015, pp. 93–108.
[31] N. Niu, S. M. Easterbrook, Concept analysis for product line requirements, in: Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09), 2009, pp. 137–148.
[32] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, 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: Proceedings of the 25th Conference on Software Engineering and Knowledge Engineering (SEKE'13), 2013, pp. 244–249.
[33] Y. Xue, Z. Xing, S. Jarzabek, Feature location in a collection of product variants, in: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12), 2012, pp. 145–154.
[34] H. E. Salman, A. Seriai, C. Dony, Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval, in: Proceedings of the 14th Conference on Information Reuse and Integration (IRI'13), 2013, pp. 209–216.

[35] T. Eisenbarth, R. Koschke, D. Simon, Locating Features in Source Code, IEEE Transactions on Software Engineering 29 (3) (2003) 210–224.
[36] Y. Yang, X. Peng, W. Zhao, Domain Feature Model Recovery from Multiple Applications Using Data Access Semantics and Formal Concept Analysis, in: Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09), 2009, pp. 215–224.
[37] A. Shatnawi, A.-D. Seriai, H. Sahraoui, Recovering Architectural Variability of a Family of Product Variants, in: Proceedings of the 14th International Conference on Software Reuse (ICSR'15), 2015, pp. 17–33.
[38] J. Carbonnel, M. Huchard, C. Nebut, Analyzing Variability in Product Families through Canonical Feature Diagrams, in: Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17), 2017, pp. 185–190.
[39] M. Acher, P. Collet, P. Lahire, R. B. France, FAMILIAR: A domain-specific language for large scale management of feature models, Science of Computer Programming 78 (6) (2013) 657–681.
[40] M. Mendonça, M. Branco, D. D. Cowan, 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'09), 2009, pp. 761–762.
[41] J. Carbonnel, D. Delahaye, M. Huchard, C. Nebut, Graph-based variability modelling: Towards a classification of existing formalisms, in: Proceedings of the 17th International Conference on Conceptual Structures (ICCS'19), 2019, pp. 1–14.
[42] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.
[43] D. Benavides, S. Segura, P. Trinidad, A. Ruiz-Cortés, A first step towards a framework for the automated analysis of feature models, Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms (2006) 39–47.
[44] M. Acher, B. Baudry, P. Heymans, A. Cleve, J. Hainaut, Support for reverse engineering and maintaining feature models, in: Proceedings of the 7th Internaional Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13), 2013, pp. 20:1–20:8.
[45] S. She, U. Ryssel, N. Andersen, A. Wasowski, K. Czarnecki, Efficient synthesis of feature models, Information & Software Technology 56 (9) (2014) 1122–1143.
[46] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, Reverse engineering feature models, in: Proceedings of the 33rd International Conference on Software Engineering, (ICSE'11), 2011, pp. 461–470.
[47] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, P. Lahire, On extracting feature models from product descriptions, in: Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12), 2012, pp. 45–54.
[48] J. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, P. Heymans, 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), 2013, pp. 290–300.
[49] D. S. Batory, Feature Models, Grammars, and Propositional Formulas, in: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05), 2005, pp. 7–20.
[50] D. Benavides, P. T. Martín-Arroyo, A. R. Cortés, Automated Reasoning on Feature Models, in: Proceedings of the 17th International Conference of Advanced Information Systems Engineering (CAISE'05), 2005, pp. 491–503.
[51] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, M. Toro, Automated error analysis for the agilization of feature modeling, Journal of Systems and Software 81 (6) (2008) 883 – 896.
[52] R. Mazo, C. Salinesi, D. Diaz, O. Djebbi, A. Lora-Michiels, Constraints: The heart of domain and application engineering in the product lines engineering strategy, IJISMD 3 (2) (2012) 33–68.
[53] J. P. Bordat, Calcul pratique du treillis de Galois d'une correspondance, Mathématiques et Sciences Humaines 96 (1986) 31–47.
[54] K. Bertet, C. Demko, J.-F. Viaud, C. Gurin, Lattices, closures systems and implication bases: A survey of structural aspects and algorithms, Theoretical Computer Science.
[55] K. Bertet, Structure de treillis. Contributions structurelles et algorithmiques. Quelques usages pour des données images, Habilitation à diriger des recherches en informatique, Université de la Rochelle, 196 pages (June 2011).
[56] B. Davey, H. Priestley, Introduction to Lattices and Order, second edition, Cambridge University Press, 2002.
[57] M. Barbut, B. Monjardet, Ordre et Classification, Hachette, 1970.