Graph-Based Variability Modelling: Towards a Classification of Existing Formalisms

Jessie Carbonnel, David Delahaye, Marianne Huchard, Clémentine Nebut

Graph-Based Variability Modelling: Towards a Classification of Existing Formalisms

Jessie Carbonnel[1], David Delahaye[1],
Marianne Huchard[1], and Clémentine Nebut[1]

LIRMM, Université de Montpellier & CNRS, Montpellier, France
firstname.lastname@lirmm.fr

Abstract. Software product line engineering is a reuse-driven paradigm for developing families of similar products from a generic product backbone with identified options. A customised product is then derived by combining the artefacts implementing the backbone with the ones implementing the chosen options. Variability analysis and representation is a central task of this paradigm: it consists in suitably defining and structuring the scope, the commonalities, and the differences between the derivable products. Several formalisms have been proposed: some are textual, such as propositional logic or constraint programming, while others are based on annotated graph representations. In this paper, we aim to survey and compare existing graph-based variability representations. Among them, conceptual structures have been used rather early and occasionally employed: this survey highlights their original position, which is due to some of their properties, including canonicity and dual view, that they provide on product configurations versus their features.

Keywords: Variability · Product Lines · Formal Concept Analysis.

1 Introduction

Product Line Engineering (PLE) is a paradigm for developing families of similar products while lowering costs and time to market, and improving product quality and diversity of supply. PLE receives a lot of attention in many domains, like mobile phone or car manufacturing, and it encounters a growing success in the domain of software engineering where software product construction can be quite fully automated [13]. Software Product Line (SPL) Engineering is a widespread methodology, with two well identified processes. *Domain Engineering* consists in defining: (1) a model of the scope and of the variability of the product line, in particular a model of product options, (2) a generic product backbone, and (3) a set of assets (or artefacts) that implement the possible options. During *Application Engineering*, a customised product is derived by combining the artefacts of the backbone with the ones implementing the chosen options. When the process is completely automated, the resulting product is an executable software, otherwise a significant part is generated.

Variability modelling is a primary task that consists in representing the common and variable aspects of products belonging to a same family. It is usually

expressed in terms of *features*, where a feature is a distinguishable and visible characteristic or behaviour of a product. For instance, an e-commerce application (a product) owns a `catalog`, may implement different `payment_methods` and manage a `basket`. In this context, each product can be associated with the set of features that it possesses. A combination of features therefore represents an abstract description of the product, also called a *configuration*. A configuration set is usually presented in a tabular view depicting products against their features. Tab. 1 represents a set of ten configurations describing a potential family of e-commerce applications, depending on nine features, which we will use throughout the paper. A cross states that the configuration (column) possesses the feature (row). For example, $conf_1$ describes an e-commerce application proposing only a catalog depicted in a grid, and which does not implement any payment method.

Table 1. Configuration Set of a Product Line About E-Commerce Applications

features	$conf_1$	$conf_2$	$conf_3$	$conf_4$	$conf_5$	$conf_6$	$conf_7$	$conf_8$	$conf_9$	$conf_{10}$
e_commerce	x	x	x	x	x	x	x	x	x	x
catalog	x	x	x	x	x	x	x	x	x	x
grid	x		x	x	x				x	
list		x				x	x	x		x
payment_method			x	x	x	x	x	x	x	x
credit_card			x		x	x		x	x	x
check				x	x		x	x		
basket			x	x	x	x	x	x	x	x
quick_purchase									x	x

Complying with domain and/or development constraints, all feature combinations may not be possible: for instance, two features may be incompatible. *Feature-oriented variability models* aim to document the existing features found in an SPL, as well as constraints, that they can be combined to form a *valid configuration*, i.e., corresponding to a functional derivable product. In other words, variability models represent constraints between features to describe a configuration set delimiting the scope of an SPL. Several formalisms have been proposed for modelling and managing variability of an existing configuration set as the one of Tab. 1. Some are textual, such as propositional logic or constraint programming, while others are based on annotated graph representations, which give complementary and sometimes overlapping views of variability [3,10]. Among these variability representations, conceptual structures associated to formal concept analysis [9] have been used rather early and occasionally employed, e.g., in [14,5]. A common issue faced by these graph-based models is their limited expressiveness, which may prevent them to exactly represent a given configuration set and encourage industries to use textual formalisms [12]. In this paper, we survey and compare in-use graph-based variability representations found in papers studying variability model synthesis from configuration sets [17,16,14,8,7,6,1].

Our objective is to identify their expressiveness limits to help practitioners choose a graphical variability representation. We highlight the particular place of conceptual structures, which are the only ones to give a canonical feature- and configuration-oriented perspective of an SPL.

The paper is organised as follows. In Sec. 2, we present the basics of the seminal in-use graph-based feature diagram representation, and we introduce feature models, which have emerged as a standard. In Sec. 3, we then establish relations between feature diagrams and propositional logic, allowing us to characterise the logical semantics of graph-based representations. In Sec. 4, we outline the different graph-based representations for variability, which are then classified, compared, and analysed in Sec. 5.

2 Feature Models and Feature Diagrams

Feature Models (FMs) and Feature Diagrams (FDs) are a family of descriptive languages that aim to document variability of an SPL in terms of features and interaction between these features. For instance, they define which features require other features, or which ones are incompatible. The constraints are represented graphically by structuring the set of features in a refinement hierarchy with decorated edges to specify their dependencies, and by cross-tree constraints. The literature on PLE makes a focus on specific relations between features having in mind that FDs and FMs are used to support several tasks (e.g., defining the product line scope, and guiding its evolution and maintenance).

Fig. 1 presents an FD about e-commerce applications. Feature `e_commerce` is the root feature. Feature `catalog` is mandatory, and it owns a xor-group composed of features `grid` and `list` (exactly one feature has to be present in a valid configuration). Features `payment_method` and `basket` are optional child-features of the root, and they require each other. Features `credit_card` and `check` form an or-group under `payment_method` (at least one of them has to be selected). Feature `quick_purchase` is an optional child feature of `basket`, and is mutually exclusive with `check`.

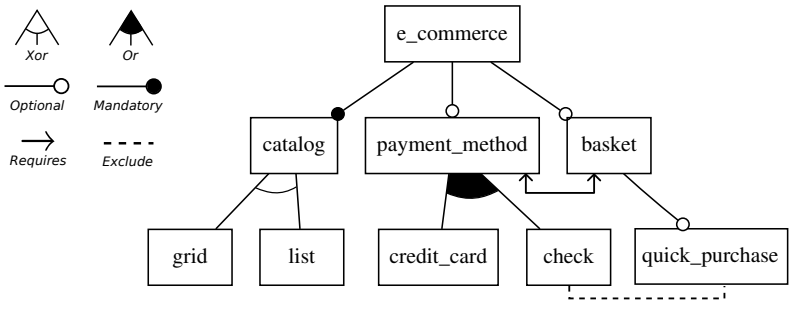


Fig. 1. Example of an FD About E-Commerce Applications

A feature combination that verifies all the constraints expressed by the FD is called a valid configuration. The set of all valid configurations is the FD *configuration semantics*. Tab. 1 is the configuration semantics of the FD of Fig. 1. Features that are present in all valid configurations are called *core-features* (e.g., `e_commerce` and `catalog`), and features that are present in none are called *dead-features*. Moreover, the feature hierarchy gives ontological information: for instance, a child feature may represent a refinement, a *part-of* or even a *use* relationship. Here, features are seen as domain concepts: this information is called the FD *ontological semantics*. A known problem in FD construction is the fact that a given configuration semantics may be represented by different ontological relationships. FDs are therefore non-canonical representations, as different FDs (i.e., presenting different ontological semantics) may be equivalent (i.e., having the same configuration semantics). Moreover, FDs are not logically complete, as some configuration sets may not be represented by this formalism (see [15]), and some authors therefore add a propositional formula to an FD.

As a consequence, She et al. [17] propose to differentiate the terms "feature diagram" and "feature model" (which is a feature diagram completed with a propositional formula), and we will use this terminology in the rest of this paper. More formally, the syntax of FDs and FMs is defined as follows:

Definition 1 (Feature Diagram [17]). *A feature diagram is defined as a tuple* $(F, E, (E_m, E_i, E_x), (G_o, G_x))$*, where* F *is a finite set of features,* $E \subseteq F \times F$ *is a set of directed child-parent edges, and* (F, E) *is a rooted tree connecting all features from* F*.* $E_m \subseteq E$ *is a set of mandatory edges,* $E_i \subseteq F \times F$ *is a set of cross-tree require edges such that* $E_i \cap E = \varnothing$*, and* $E_x \subseteq F \times F$ *is a set of cross-tree exclude edges such that* $E_x \cap E = \varnothing$*. The two sets* G_o *and* G_x *contain subsets of* E*, representing edges involved in or-groups and xor-groups, respectively. Two distinct subsets of* $G_o \cup G_x$ *are disjoint, and all edges in a subset have the same parent-feature.*

Definition 2 (Feature Model [17]). *A feature model is defined as a pair* (FD, φ)*, where FD is a feature diagram, and* φ *is a propositional formula where the propositional variables are the features of the feature set* F *of FD.*

Now, consider that we have the same set of configurations as given in Tab. 1 without the $conf_5$ showing, among other things, that both child features of `payment_method` can be selected with a catalog displayed in a `grid`. To respect this new configuration semantics, the FD should state that selecting both `check` and `credit_card` implies to select `list`. As it is not possible to express such complex implication only by means of an FD, we must add the following formula $\varphi' = (\texttt{check} \wedge \texttt{credit\_card}) \Rightarrow \texttt{list}$ to the previous FD. The FM composed of the FD of Fig. 1 and φ' therefore represents the new configuration semantics.

3 Semantics of FDs and FMs Using Propositional Logic

Mannion [11] was the first to build relationships between FDs and propositional logic. Ever since, FD ontological relationships were translated in the form of a

propositional formula, where features are represented by propositional variables and relationships are defined using logical connectives. Therefore, a *logical semantics* is also associated with an FD through a propositional formula that has for models the FD valid configurations. Its use is very popular in work aiming to carry out automated analysis or reasoning over product line variability [2]. As for FMs, the logical semantics of an FM defined as $FM = (FD, \varphi)$ is $\varphi_{FD} \wedge \varphi$, where φ_{FD} is the propositional formula corresponding to the logical semantics of FD.

Tab. 2 presents how the ontological semantics of an FD is defined in propositional logic, giving its *logical semantics*. Two logical forms are given. The first one (column 2) is commonly used in the SPL domain and uses the following logical connectives: $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), $\vee$ (or), and $\oplus$ (exclusive or). The second one (column 4) shows the equivalent formula in Conjunctive Normal Form (CNF), i.e., expressed as sets of clauses (a clause is a disjunction of literals, and a literal corresponds to a feature or the negation of a feature). Following the tracks of [7], and because the nonempty clauses represent implications, the logical semantics written in CNF (column 4) is appropriate to highlight the expression power of the graph-based representations.

Table 2. Logical Semantics of FD Constraints

Constraints	**Logical Semantics**	**Formula Name**	**CNF Form**
optional	$c \Rightarrow p$	Binary	$\neg c \vee p$
mandatory	$p \Leftrightarrow c$	Implication	$\neg c \vee p$, $\neg p \vee c$
requires	$f_1 \Rightarrow f_2$	(BI)	$\neg f_1 \vee f_2$
exclude	$f_1 \Rightarrow \neg f_2$	Mutex (MX)	$\neg f_1 \vee \neg f_2$
or-group	$p \Leftrightarrow (c_1 \vee \ldots \vee c_n)$	OR	$\neg p \vee c_1 \vee .... \vee c_n$ $\neg c_i \vee p, 1 \leq i \leq n$
xor-group	$p \Leftrightarrow (c_1 \oplus \ldots \oplus c_n)$	XOR	$\neg p \vee c_1 \vee .... \vee c_n$ $\neg c_i \vee p, 1 \leq i \leq n$ $(\neg c_i \vee \neg c_j)$, $1 \leq i, j \leq n, i \neq j$
core-feature	$\top \Rightarrow f_1$	CF	f_1
dead-feature	$f_1 \Rightarrow \bot$	DF	$\neg f_1$

p represents a feature in a parent position, c, c_i, $1 \leq i \leq n$, features in a child position, and f_1, f_2 any feature. Columns 2 and 4 show patterns of clause sets corresponding to the FD constraints. For a clause G corresponding to the OR or XOR pattern, its associated child set is $C_G = \{c_1, \ldots, c_n\}$.

To build the formula corresponding to the logical semantics of an FD, the algorithm consists in following the FD from the root to its leaves, and applying the constructs of Tab. 2 to produce the clauses. The obtained formula is therefore a conjunction of expressions conforming to the patterns of clause sets of Tab. 2, i.e., BI, MX, OR, XOR, CF, and DF. In addition, this formula verifies two properties that follow directly from the definition of an FD: (1) presence of a

root and (2) the fact that the child sets of distinct feature groups are disjoint. These properties can be formally expressed as follows:

Property 1 (FD-to-CNF). Given a feature diagram FD, if φ is the logical semantics of FD then φ verifies the following properties: (a) φ is equal to a conjunction of clauses that conform to the BI, MX, OR, XOR, CF, or DF patterns; (b) φ contains at least one clause r that conforms to the CF pattern (existence of a root, which is a core-feature, i.e., present in all valid configurations); (c) for any G_i, G_j clauses of φ that conform to the OR or XOR patterns, with respective child sets C_{G_i}, C_{G_j}, $C_{G_i} \cap C_{G_j} = \emptyset$ (distinct feature group child sets are disjoint).

Conversely, any formula that can be expressed as a CNF verifying some of the properties mentioned above in Prop. 1 can be represented by an FD:

Property 2 (F-to-FD). Any propositional logic formula φ that can be expressed as a CNF φ_{FD} such that (a) φ_{FD} is a conjunction of clauses conform to the BI, MX, OR, XOR, CF, or DF patterns, and (b) for all clauses G_i, G_j of φ_{FD} that conform to the OR or XOR patterns, with respective child sets C_{G_i}, C_{G_j}, we have $C_{G_i} \cap C_{G_j} = \emptyset$, can be represented by an FD.

It should be noted that the property (b) of Prop. 1 has been relaxed in Prop. 2. If no root can be identified in the formula (no clause that conforms to the CF pattern in the CNF representation of the formula) then it is always possible to complete the corresponding FD with a root feature to get a fully connected graph.

As said previously in Sec. 2, FMs have been introduced to alleviate the logical incompleteness of FDs. As a consequence, FMs are logically complete, which can be expressed by the following property:

Property 3 (F-to-FM). Any propositional logic formula can be represented by an FM.

As FDs are not logically complete (which may require the addition of complementary formulas), we may wonder which kind of formulas cannot be represented as FDs, and in particular, what are the different forms of such formulas. In the following, to characterise these formulas, we will use some notations inspired by regular expressions: n represents any negative literal, p any positive literal, l^{k+} at least k times the literal l (if k is 0, l is omitted).

Any propositional formula in CNF can have three types of nonempty[1] clauses: only positive literals (p^+), only negative literals (n^+), and mixed, namely containing positive and negative literals (p^+n^+). To identify the formulas that cannot be represented by FDs, the principle is to look at the patterns of clause sets of Tab. 2 and identify the missing patterns. In Tab. 2, the BI pattern corresponds to np, the MX pattern corresponds to n^2, the OR and XOR patterns include np, np^{2+}, and n^2 (the latter for XOR), CF corresponds to p, and DF to n. We can observe that the following forms are not captured by the patterns of Tab. 2:

[1] When representing FDs, empty clauses are of little interest as any set of clauses containing an empty clause is insatisfiable, and it corresponds to an FD with no valid configuration.

- A mixed clause with at least two negative literals (denoted by p^+n^{2+});
- A mixed clause with at least two positive literals (denoted by n^+p^{2+}), with the case of the clause np^{2+}, when it is not included in an OR or XOR pattern;
- A clause with at least two and only positive literals (denoted by p^{2+});
- A clause with at least three and only negative literals (denoted by n^{3+}); these generalised exclusion clauses will be called NAT (*Not All Together*).

We do not provide a proof that the set of these patterns is complete in the sense that the set of formulas that can be represented by FDs together with the set of formulas that can be represented using these patterns allows us to represent any formula. However, our approach is purely syntactical (trying to match the patterns of representable clauses and the patterns of Tab. 2), and the reader should be easily convinced of the completeness of this set of patterns.

These missing clause patterns[2] may be needed for expressing a configuration set. The formula that completes an FD (if needed), to obtain an FM, will therefore be mainly composed of clauses that conform to these patterns.

4 Graph-Based Formalisms for Variability Structuring

In this section, we consider a product line variability information through a propositional formula φ defined over a set of variables $F = \{f_1, \ldots, f_k\}$. F represents the product line feature set and the models of φ the product line configuration set. In what follows, we survey graph-based variability representations used in work about FM synthesis, and we compare the variability information that they express with the one expressed by φ.

Binary Decision Graphs [7,1,17]. A Binary Decision Tree (BDT) is a canonical tree-like graph used to depict the truth table of a Boolean function of the form $\{0,1\}^k \to \{0,1\}$, which can represent a propositional formula in k variables. Each internal node represents a variable and has two outgoing edges: a low edge and a high edge. A path from the root to a leaf corresponds to a variable assignment (i.e., a configuration): a low edge assigns the variable to 0, and a high edge assigns the variable to 1. The value of an assignment is given by the leaf (terminal node), which is equal to either 1 (valid configuration) or 0 (invalid configuration). It is an extensional representation of the configuration set. The BDT representation has redundancies, which can be avoided by node sharing, which results in a graph called **Binary Decision Diagram** (BDD) [4,7]. The term BDD usually refers to ROBDD (for Reduced Ordered Binary Decision Diagram), which is unique for a given propositional formula. A ROBDD therefore represents the set of valid configurations (models of φ), but it does not represent feature or configuration interactions. However, ROBDDs are logically complete. Fig. 2 presents the ROBDD associated with the part of Tab. 1 restricted to features `e_commerce`, `catalog`, `grid`, and `list`.

[2] Let us notice that they can appear in a formula satisfying Prop. 2 provided that, combined with other clauses, they can disappear to the benefit of emergence of clauses of the admitted patterns BI, MX, OR, XOR, CF, or DF.

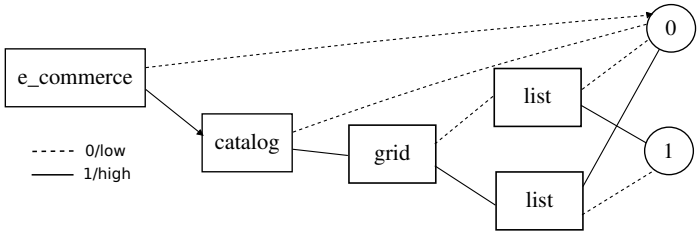


Fig. 2. ROBDD Associated with the First Four Features of Tab. 1

Binary Implication Graphs [7,16,1,8,17]. A Binary Implication Graph (BIG) is a directed graph denoted by $G(V, E)$ where $V = F = \{f_1, \ldots, f_k\}$ and $E = \{(f_i, f_j) \mid \varphi \wedge (f_i \rightarrow f_j)\}$ representing binary implications (BI) between features. A feature f_i *implies* a feature f_j when each valid configuration having f_i also has f_j. BIGs are intensional representations structuring the product line feature set. This type of representation is not canonical, but its transitive closure and transitive reduction are. Fig. 3 (left-hand side) presents the transitive reduction of the BIG associated with the first four features of Tab. 1.

Directed Hypergraphs [7]. A directed hypergraph is a directed graph generalisation where an arc can connect more than two vertices: such arcs are called hyperarcs. Vertices correspond to variables (features of F) and Boolean constants (0 and 1). A directed hypergraph can represent all types of nonempty clauses. A clause having both negative and positive literals is represented by a hyperarc $A \rightarrow B$ with $A, B \subseteq F$, A being the conjunction of the clause negative literals and B the disjunction of the clause positive literals. A clause having only negative literals is represented by a hyperarc $A \rightarrow 0$, and a clause with only positive literals is represented by a hyperarc $1 \rightarrow A$. As BIGs, their transitive reduction/closure are canonical representations, and they structure the feature set in an intensional representation. Fig. 3 (right-hand side) presents the directed hypergraph transitive reduction of the first four features of Tab. 1.

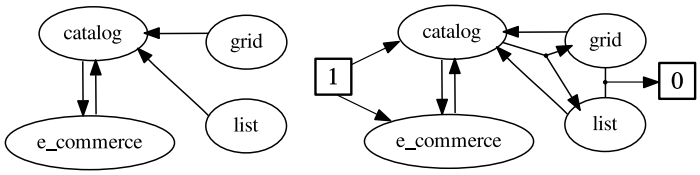


Fig. 3. Transitive Reduction of the Binary Implication Graph (Left-Hand Side) and the Directed Hypergraph (Right-Hand Side) Associated with the First Four Features of Tab. 1

Mutex Graphs [16,17]. A mutex graph is an undirected graph denoted by $G(V, E)$ where $V = F = \{f_1, \ldots, f_k\}$ and $E = \{\{f_i, f_j\} \mid \varphi \wedge \neg(f_i \wedge f_j)\}$ rep-

resenting mutual exclusions, also called mutex (MX) between features. Features are mutually exclusive if they cannot appear together in any valid configuration. There is a unique mutex graph associated with a propositional formula. Note that a clique in the graph represents incompatibilities inside each pair of involved features. Fig. 4 (left-hand side) presents the mutex graph extracted from Tab. 1.

Feature Diagram Generalised Notation and Feature Graphs [7,17]. Czarnecki and Wasowski [7] propose an FD generalised notation where the feature tree may be replaced by a directed acyclic graph (encompassing require cross-tree constraints as optional relationships), feature groups may overlap, and co-occurrent features are visualised in a single node. Exclude cross-tree constraints are not represented in this formalism. This generalised notation is a canonical and intensional representation that covers the same information found in usual FDs except mutual exclusions (i.e., BI, OR and, XOR) but without the structural constraints that require expert decisions during the synthesis. An FD in generalised notation therefore represents several FDs. She et al. [17] extend this notation with mutex groups and mutual exclusions, and called this extension a feature graph. A mutex group is a set of features such that each pair is a mutex (i.e., corresponding to a clique in the mutex graph). Feature graphs depict MX in addition to FD generalised notation relationships (see the right-hand side of Fig. 4 for an example).

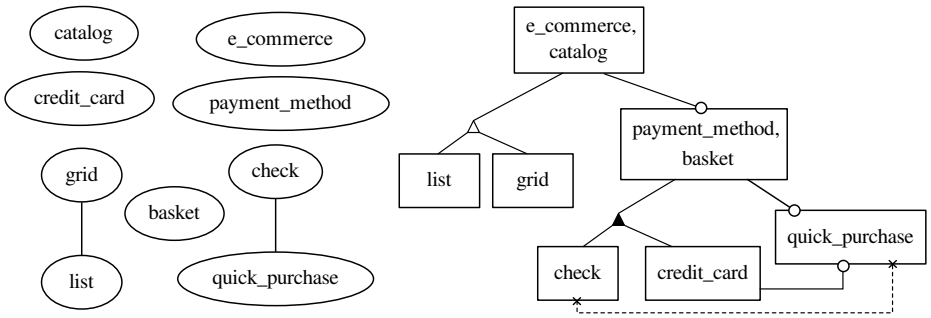
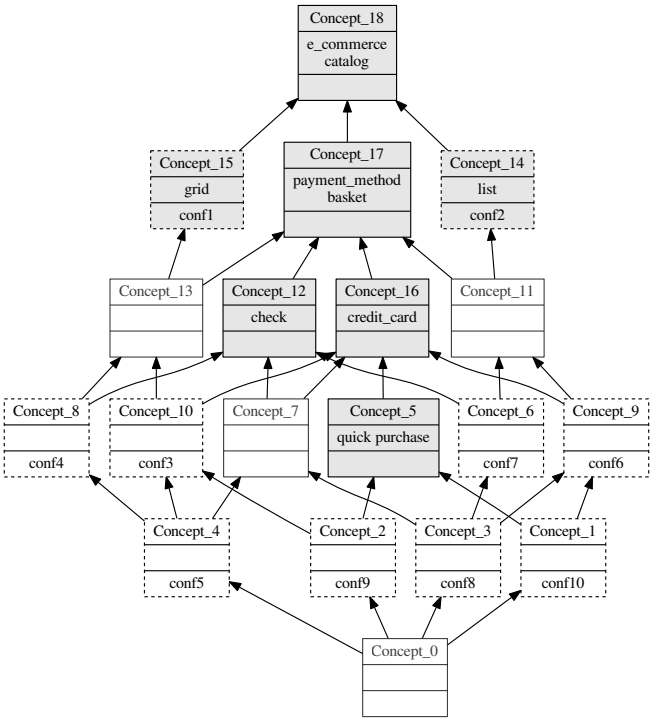


Fig. 4. Mutex Graph (Left-Hand Side) and Feature Graph (Right-Hand Side) of Tab. 1

Conceptual Structures [9]. A set of configurations (e.g., Tab. 1) naturally underlies a formal context $K = (G, M, J)$ composed of configurations (G), features (M), and a binary relation (J) stating which configuration has which feature. Formal Concept Analysis (FCA) is a natural framework for variability representation: each concept $C = (E, I)$ gathers a maximal set of configurations E (extent) sharing a maximal set of features I (intent). For example, $Concept\_5$ gathers the configurations $conf9$ and $conf10$, together with their shared features $e\_commerce$, $catalog$, $payment\_method$, $basket$, $credit\_card$, and $quick\ purchase$. The concept lattice (see Fig. 5) provides the concept set with a specialisation order $\leq$, where we have $C_1 = (E_1, I_1) \leq C_2 = (E_2, I_2)$ if



Attribute-concepts are coloured in grey and object-concepts have a dashed border.

Fig. 5. Concept Lattice Associated with Tab. 1

$E_1 \subseteq E_2$. For instance, we have $Concept\_5 \leq Concept\_16$, the latter adding to $Concept\_5$'s extent the configurations $conf_3$, $conf_5$, and $conf_6$, which eliminates *quick purchase* from the shared features. An attribute-concept $\mu(f)$ (resp. an object-concept $\gamma(c)$) introduces a feature f (resp. a configuration c), if it is the highest (resp. lowest) concept where f (resp. c) appears. For example, $Concept\_5$ is an object-introducer (introducing *quick purchase*), in grey in the figure, $Concept\_6$ is an attribute-introducer (introducing $conf_7$), within a dashed border concept in the figure. In the representation, the features (resp. configurations) are only written in their introducer concept and inherited top-down (resp. bottom-up). Several sub-structures are of interest for the SPL domain. The AC-poset (resp. OC-poset) is the sub-order of the concept lattice restricted to attribute-concepts (resp. object-concepts), while the AOC-poset contains both types of introducers. FCA conceptual structures are the only graph-based variability representations being both intensional and extensional, and structuring both products and features. They are also canonical and logically complete.

Equivalence Class Feature Diagrams [6]. An Equivalence Class Feature Diagram (ECFD) is derived from an AC-poset. It has been introduced in [6] as an intermediate canonical structure for analysing variability. It graphically

represents all the feature co-occurrences (equivalent features, like *e_commerce* and *catalog*), and all the BI (with ECFD arrows), MX, OR, and XOR that can be extracted from the concept lattice. It may also contain generalised exclusions (*NAT*, standing for *Not All Together*), i.e., of the form n^{3+}. The OR and XOR groups may overlap and the ECFD structure corresponds to an acyclic graph. Fig. 6 (right-hand side) shows the ECFD extracted from the AC-poset of Fig. 6 (left-hand side). All the feature diagrams that have the same configuration semantics can be embedded in the ECFD built on the formal context associated to the configuration set. It is an intensional and canonical representation structuring the feature set, but it is not logically complete, as it aims to represent FD variability information.

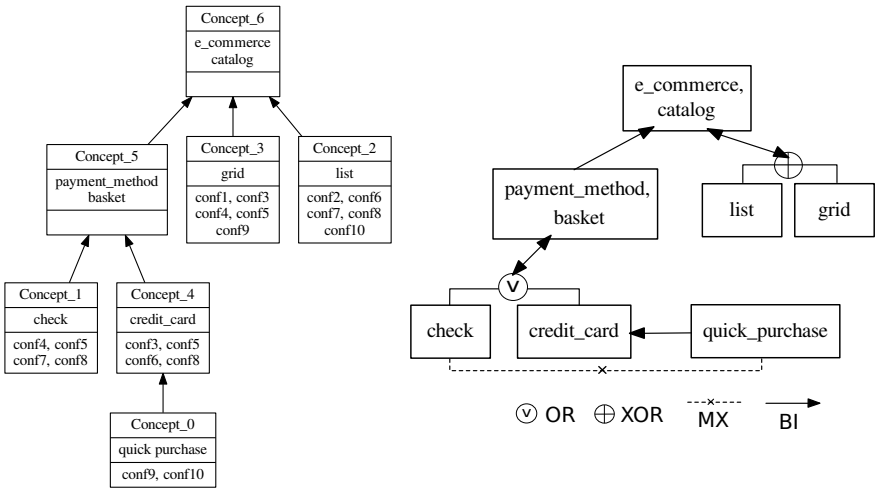


Fig. 6. AC-Poset (Left-Hand Side) and Extracted ECFD (Right-Hand Side) Associated with Tab. 1

5 Towards a Classification of the Graph-Based Formalisms for Variability

In this section, we first compare the studied formalisms through two points of view: the covered propositional logical expressions and the properties related to the graph-based representation. We then give examples of possible uses of this comparison by designers.

Tab. 3 gathers information about the formalisms presented in the previous sections. The first two parts of the table present the logical properties that are defined in Sec. 3. The first four columns state if a formalism is able to express the clause set patterns found in traditional FDs. The next four columns present the four disjunctive clause patterns that cannot be diagrammatically represented (if

they are not combined with other clause patterns) in an FD. They characterise the propositional formulas that should complete the FD semantics to represent all possible configuration sets. The third part of the table gives representation properties, i.e., the canonicity of the formalisms, if they structure the feature set or the product set, and if they are intensional or extensional representations. The last two columns state if the formalisms need to own a root feature, and if their feature groups are distinct (i.e., do not overlap). They correspond to the meta-properties expressed in Sec. 3. The `rooted` column is not applicable (*na*) for formalisms that do not represent binary implications, and the `distinct groups` column is not applicable for formalisms without feature groups. The formalisms that can express the ten types of clause set patterns are therefore logically complete. Aside from FMs, which are logically complete thanks to their complementary propositional formulas, three types of graph-based representations can depict all propositional formulas. ROBDDs are complete and canonical, but they do not provide feature organisation in an intensional representation of variability, which is one of the main goal of variability modelling. Directed hypergraphs have the same advantages as ROBDDs, but in addition they structure the feature set to help visualising variability information. Concept lattices and AOC-posets appear as the most complete variability representations, as they also depict and structure the set of valid configurations. These structures are the only ones gathering all logical and representation properties. Their main drawback is their size: the number of nodes in a concept lattice may grow exponentially with the size of the data input, as they organise both feature and product sets, making it difficult to compute and use to handle large product lines. Fortunately, AOC-, AC- and OC-posets have a node number limited respectively by the number of products plus features, the feature number, or the product number.

Thanks to this table, we are able to characterise CNFs that can be represented by each formalism, as we have done for FDs in Sec. 3. We characterise a CNF by 1) a conjunction of clause set patterns of certain types and 2) some meta-properties. For a given formalism, the CNFs that can be written as a conjunction of the clause set patterns associated to this formalism (amongst the types in columns [2-11]) and that respect the corresponding meta-properties (amongst those in the last two columns) can be represented diagrammatically by this formalism. Note that our analysis reveals that the two meta-properties are only necessary to FDs and FMs. Conversely, the comparison can be used to detect logic formulas that cannot be expressed by the existing graphical formalisms.

Our comparison may also be used to assist a designer who aims to represent the variability of a product line structured in a formalism A (for instance, which can be automatically extracted) with another formalism B (for instance, which needs an expert intervention during the synthesis). If the target formalism represents the same set or a subset of clause patterns of the ones represented by the source formalism, the transformation can be done without logical information loss. If the source formalism represents a subset of clauses of the ones represented by the target formalism, our analysis helps the designer identify the type of logical relationships that will be lost in the transformation.

Table 3. Comparison of the Different Graph-Based Variability Representations Depending on Logical and Representation Properties

Formalism	np (BI)	n^2 (MX)	OR	XOR	p (CF)	n (DF)	n^{3+} (NAT)	p^+n^{2+}	n^+p^{2+}	p^{2+}	Canonical	Feature Struct.	Product Struct.	Intension	Extension	Rooted	Distinct Groups
FD	x	x	x	x	x	x						x		x		x	x
FM	x	x	x	x	x	x	x	x	x	x		x		x		x	x
ROBDD	x	x	x	x	x	x	x	x	x	x	x				x		
BIG (Trans. Red)	x				x						x	x		x			*na*
Mutex Graph		x									x	x		x		*na*	*na*
FM Gen. Nota.	x		x	x	x	x					x	x		x			
Feature Graph	x	x	x	x	x	x					x	x		x			
Dir. Hypergraph	x	x	x	x	x	x	x	x	x	x	x	x		x			
Concept Lattice	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
AOC-Poset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
AC-Poset	x	x	x	x	x	x	x	x	x	x	x	x		x	x		
OC-Poset	x	x	x	x	x	x	x	x	x	x	x		x	x	x		
ECFD	x	x	x	x	x	x					x	x		x			
ECFD+NAT	x	x	x	x	x	x	x				x	x		x			

6 Conclusion

Variability modelling is a central aspect in the spreading paradigm of product lines in product design and construction. Disposing of a variety of formalisms, a clear understanding of their scope and applicability, and embedding methodologies and algorithms is of major importance. In this paper, we give keys to move in these directions. The point is not to oppose the different formalisms, but being able to use the right one at the right moment. We survey the most popular and in-use formalisms in SPL engineering research where most of the current advances are made, ranging from feature diagrams to conceptual structures. We compare and discuss them through disjunctive clauses categories that highlight their expressiveness, and through other properties such as canonicity, and whether they emphasise the relations between features, configurations, and both features and configurations. Conceptual structures play a particular role in this variety of formalisms, providing canonical representations, configuration-oriented as well as feature-oriented views, while being logically complete.

As future work, the comparison can be extended to non-Boolean formalisms including FDs with attributes or with references. FCA owns extensions, namely pattern structures and relational concept analysis, which could be used to capture new aspects of variability modelling. New graphical operators could also be imagined as a result of these studies. For identified useful transformations,

we could investigate how one formalism embeds into another one, with which possible precision loss and algorithmic complexity.

References

1. Acher, M., Baudry, B., Heymans, P., Cleve, A., Hainaut, J.: Support for Reverse Engineering and Maintaining Feature Models. In: Proc. of the 7th Int. Workshop on Variability Modelling of Software-intensive Systems. pp. 20:1–20:8 (2013)
2. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems **35**(6), 615–636 (2010)
3. Bontemps, Y., Heymans, P., Schobbens, P., Trigaux, J.: Generic Semantics of Feature Diagrams Variants. In: Feature Interactions in Telecommunications and Software Systems VIII. pp. 58–77 (2005)
4. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986)
5. Carbonnel, J., Huchard, M., Miralles, A., Nebut, C.: Feature Model Composition Assisted by Formal Concept Analysis. In: Proc. of the 12th Int. Conf. on Evaluation of Novel Approaches to Software Engineering. pp. 27–37 (2017)
6. Carbonnel, J., Huchard, M., Nebut, C.: Analyzing Variability in Product Families through Canonical Feature Diagrams. In: Proc. of the 29th Int. Conf. on Software Engineering and Knowledge Engineering. pp. 185–190 (2017)
7. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: Proc. of the 11th Int. Software Product Line Conference. pp. 23–34 (2007)
8. Davril, J., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., Heymans, P.: Feature Model Extraction from Large Collections of Informal Product Descriptions. In: Proc. of the 9th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 290–300 (2013)
9. Ganter, B., Stumme, G., Wille, R.: Formal Concept Analysis: Foundations and Applications, vol. 3626. Springer (2005)
10. Knüppel, A.: The Role of Complex Constraints in Feature Modeling. Master's thesis, Institute of Software Engineering and Automotive Informatics, Technische Universität Carolo-Wilhelmina zu Braunschweig (2016)
11. Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Proc. of the 2nd Int. Software Product Line Conf. pp. 176–187 (2002)
12. Mazo, R., Salinesi, C., Diaz, D., Djebbi, O., Lora-Michiels, A.: Constraints: The Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. IJISMD **3**(2), 33–68 (2012)
13. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
14. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of Feature Models from Formal Contexts. In: Work. Proc. (2) of the 15th Int. Conf. on Software Product Lines. pp. 4:1–4:8 (2011)
15. Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y.: Generic Semantics of Feature Diagrams. Computer Networks **51**(2), 456–479 (2007)
16. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse Engineering Feature Models. In: Proc. of the 33rd Int. Conf. on Software Engineering. pp. 461–470 (2011)
17. She, S., Ryssel, U., Andersen, N., Wasowski, A., Czarnecki, K.: Efficient Synthesis of Feature Models. Information & Software Technology **56**(9), 1122–1143 (2014)