Feature Diagrams: A Survey and a Formal Semantics

Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux
University of Namur, Computer Science Department
5000 Namur, Belgium
fpys,phe,jtrg@info.fundp.ac.be

Yves Bontemps
SMALS MvM / eGOV
1050 Brussels, Belgium
Yves.Bontemps@smals-mvm.be

Abstract

Feature Diagrams (FD) are a family of popular modelling languages used for engineering requirements in software product lines. FD were first introduced by Kang as part of the FODA (Feature Oriented Domain Analysis) method back in 1990. Since then, various extensions of FODA FD were devised to compensate for a purported ambiguity and lack of precision and expressiveness. However, they never received a proper formal semantics, which is the hallmark of precision and unambiguity as well as a prerequisite for efficient and safe tool automation.

In this paper, we first survey FD variants. Subsequently, we generalize the various syntaxes through a generic construction called Free Feature Diagrams (FFD). Formal semantics is defined at the FFD level, which provides unambiguous definition for all the surveyed FD variants in one shot. All formalization choices found a clear answer in the original FODA FD definition, which proved that although informal and scattered throughout many pages, it suffered no ambiguity problem.

Our definition has several additional advantages: it is formal, concise and generic. We thus argue that it contributes to improve the definition, understanding, comparison and reliable implementation of FD languages.

1. Introduction

Software Product Line (PL) engineering is *"a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation"* [17]. Central to the PL paradigm is the modelling and management of *variability*, i.e. *the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts* [17]. At all those levels, but especially at the requirement level, variability is commonly modelled through *Feature Diagrams* (FD).

In the last 15 years or so, research and industry have developed several FD languages. The first and seminal proposal was introduced as part of the FODA method back in 1990 [12]. An example of a FODA FD is given in Fig. 1. It is inspired from a case study defined in [4] and indicates the allowed combinations of features for a family of systems intended to monitor the engine of a car. As is illustrated, FODA features are nodes of a graph represented by strings and related by various types of edges. On top of the figure, the feature `Monitor Engine System` is called the *root feature*, or *concept*. The edges are used to progressively decompose it into more detailed features. In FODA, there are *and-* and *xor-* decompositions (where edges are linked by a line segment, as between `Measures` and its sons in Fig. 1). The exact meaning of these and other constructs will be discussed extensively throughout the paper.

In the sequel, we will refer to FODA FD as Original Feature Trees or OFT for short. "Tree" is because in OFT, FD are structured as trees *vs.* single-rooted directed acyclic graphs (DAG) as in FORM [13].

Since Kang *et al.*'s initial proposal, several extensions of OFT have been devised as part of the following methods: FORM [13], FeatureRSEB [10], Generative Programming [7], PLUSS [8], and in the work of the following authors: Riebisch *et al.* [19, 18], van Gurp *et al.* [24], van Deursen *et al.* [23], Czarnecki *et al.* [5, 6], Batory [1] and Benavides *et al.* [2]. While few authors have recently started to better define their semantics [23, 6, 1, 2], most proponents of FD [13, 10, 7, 19, 18, 24, 8] have not. Still, most of them have argued for an "improved expressiveness". However, without a formal semantics, they have

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE

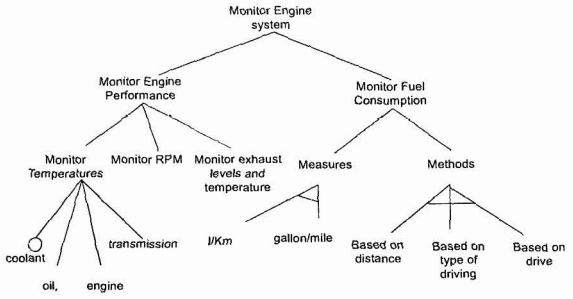

Figure 1. FODA (OFT): Monitor Engine System

failed to demonstrate it, unlike [3], a preliminary version of this study. Furthermore, some authors [19, 18] have claimed formality but have actually addressed semantic issues only superficially. The present paper offers the means for a rigorous assessment of such claims.

Formal semantics is not an issue to be taken lightly. As remarkably argued in [11], formal semantics is the best way to avoid ambiguities and to start building safe automated reasoning tools. For example, we may want to ensure that a FD excludes all the forbidden feature combinations and admits all the valid ones. If this is not the case, harmful feature interactions (a.k.a. interferences) are likely to take place, or the PL will be unnecessarily restricted and thus less competitive. A tool for assisting stakeholders in selecting features therefore must be based on formal semantics.

Without a formal semantics, there is a risk that new FD languages and constructs will continue to proliferate on the basis of shallow or even erroneous motivations, leading to interpretation and interoperability problems. For example, if two PL that use two different modelling languages need to be merged, their respective teams will have to make extra efforts to understand each other's FD, thereby increasing the likelihood of misunderstandings. If these languages are poorly defined, the risk is even higher. Moreover, a new FD will have to be created from the existing ones to account for the new merged PL. A tool can surely help in that but only if it is based on clearcut language definitions and proved correct semantic-preserving translations.

This paper delivers solutions to the aforementioned problems. Its content is organized as follows. First, in Section 2, we survey OFT and its extensions, generically called FD. In Section 3, we propose a formal framework designed to easily provide FD languages with formal semantics. We take care to make this framework generic in order to account for extant FD variants (section 3.3) and possibly others to come. Section 4 discusses the benefits of this formalization. Finally, Section 5 compares our results with related works, especially other formal definition endeavours [23, 6, 1, 2], while Section 6 concludes the paper.

2. Survey

In this section, we survey the informal OFT extensions enumerated in Section 1. For convenience, we have developed a language naming scheme. FD languages without graphical constraints (see below) are named with three or four letters acronyms. The first or first two letters are representative of the original name of the language, the method it comes from or its authors. Then comes an 'F' (for feature). The last letter is either 'D' (for Directed Acyclic Graph, or DAG) or 'T' (for trees). The name "OFT" introduced in the previous section already follows this scheme.

2.1 FODA (OFT)

OFT, the first ever FD, were introduced as part of the Feature Oriented Domain Analysis (FODA) method [12]. Their main purpose was to capture commonalities and variabilities at the requirements level. As depicted in Fig. 1, OFT are composed of:

1. A *concept*, a.k.a *root node*, which refers to the complete system.
2. *Features* which can be *mandatory* (by default) or *optional* (with a hollow circle above, e.g. `coolant`) and which are subject to decomposition (see below). The concept is always mandatory.
3. *Relations* between nodes materialized by:
   (a) *decomposition edges* (or *consists-of*): FODA further distinguishes:
      i. *and-decomposition* e.g. between `Monitor Fuel Consumption` and its sons, `Measures` and `Methods`, indicating that they should both be present in all feature combinations where `Monitor Fuel Consumption` is present.
      ii. and *xor-decomposition* e.g. between `Measures` and its sons, `l/km` and `Miles/gallon`, indicating that only one of them should be present in combinations where `Measures` is.

   Decomposition edges form a tree.

   (b) *textual constraints*:
      i. *requires*, e.g. one could complete Fig. 1 with the constraint: `Based on drive` *requires* `Monitor RPM`. This indicate that "to monitor fuel consumption using a method based

on drive, we need to monitor the engine's RPM". Thus, the former feature cannot be present if the latter is not.

ii. *mutex*, an abbreviation for "mutually exclusive with", indicates that two features cannot be present simultaneously.

2.2 FORM (OFD)

Kang *et al.* have proposed the Feature-Oriented Reuse Method (FORM) [13] as an extension of FODA. Their main motivation was to enlarge the scope of feature modelling. They argue that feature modelling is not only relevant for requirements engineering but also for software design. In terms of FD, several changes occurred:

1. FD are not necessarily trees but can be DAG.
2. Each feature pertains to one of four *layers*: the *capability layer*, the *operating environment layer*, the *domain technology layer* or the *implementation technique layer*.
3. In addition to the *composed-of* relationship (equivalent to *consists-of* in OFT), two types of graphical relationships were added: *generalization/specialization* and *implemented-by*.
4. Feature names appear in boxes.

We call the abstract syntax of such diagrams "Original Feature Diagrams" (OFD).

In this paper, we only consider the constructs which have an influence on the feature combinations allowed in the final products (see Def. 3.5). Therefore, only the first change (trees becomes DAG) is relevant for us. In our running example, the FD is a tree, so the FORM version of it is similar to Fig. 1, except that features would be boxed.

2.3 FeatuRSEB (RFD)

The FeatuRSEB method [10] is a combination of FODA and the Reuse-Driven Software Engineering Business (RSEB) method. RSEB is a use-case driven systematic reuse process, where variability is captured by structuring use cases and object models with *variation points* and *variants*. FeatuRSEB FD have the following characteristics:

1. They are DAG.
2. Decomposition operator *or* (black diamond) is added to *xor* (white diamond) and *and*. Features which sons are decomposed through *or* or *xor* are called *variation points* and those sons are called *v*ariants.
3. They possess a graphical representation (dashed arrows) for the constraints *requires* and *mutex*.

We call such diagrams "RSEB Feature Diagrams" (RFD). An example is given in Fig. 2.

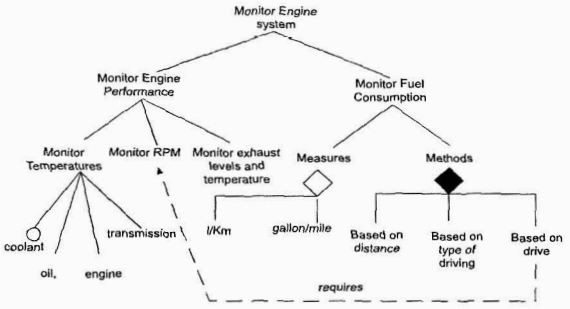

Figure 2. FeatuRSEB (RFD): Monitor Engine System

2.4 van Gurp *et al.* (VBFD)

van Gurp, Bosch and Svahnberg define another FD language in [24]. This language extends FeatuRSEB (RFD) to deal with *binding times*, indicating *when* features can be selected, and *external features*, which are technical possibilities offered by the target platform of the system. Their FD have the following characteristics:

1. *Binding times* are used to annotate relationships between features.
2. Features are boxed, as in FORM (OFD).
3. External features are represented in dashed boxes.
4. *xor* and *or* variation points are represented as white and black triangles, respectively.

We call such diagrams "van Gurp and Bosch Feature Diagrams" (VBFD). These changes mainly concern concrete syntax, and certainly have no influence on feature combinations. Thus we consider them equal to RFD.

2.5 Generative Programming (GPFT)

Czarnecki and Eisenecker [7] have studied and adapted FD in the context of Generative Programming (GP), a new programming paradigm that aims to automate the software development process for product families. Their FD are simply OFT with the addition of *or*-decomposition.

We call such diagrams "Generative Programming Feature Trees" (GPFT).

Recently, the authors have further augmented their approach with concepts such as *staged configuration* [5], distinguishing between *group* and *feature cardinalities* (see

Section 2.6) and formalizing their language [6]. These latter works will be discussed in Section 5.

2.6 Riebisch *et al.* (EFD)

Riebisch *et al.* claim that multiplicities (a.k.a. cardinalities) are partially represented with the previous notations [19]. Moreover, they argue that *"combinations of mandatory and optional features with alternatives, or and xor relations could lead to ambiguities"* [18, p.4]. As we will see, those "ambiguities" are due to a different conception of what "mandatory" and "optional" mean (see Example 4.3), and better termed redundancy (see Section 4.2 and [3]).

In order to limit these drawbacks, the authors replace *or*, *xor* and *and* by UML-like *multiplicities* (a.k.a group cardinalities [6]) and *mandatory* and *optional edges* [19]. Multiplicities consist of two integers: a lower and an upper bound. Multiplicities are illustrated in Fig. 3 for the decomposition of features `Measures` and `Methods`. In fact, they are used for all decompositions but not mentioned when both the lower and upper bound equal the number of sons, that is, for the equivalent of *and*-decomposition.

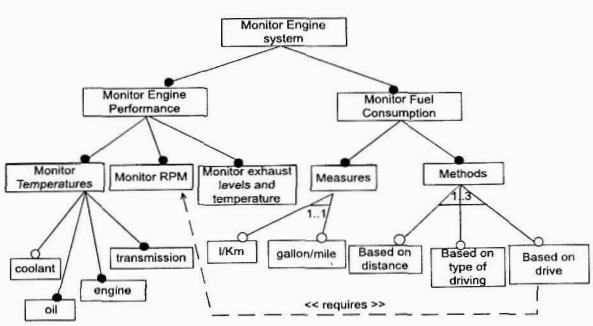

Figure 3. EFD: Monitor Engine System

In Fig. 3, we can also observe mandatory (resp. optional) edges which are marked at the lower end with a black (resp. hollow) circle. This is rather original as the other FD languages prefer to mark the optionality on the node. Optional edges are useful in DAG (and [19] indeed use DAG) since a feature can be optional on one side and mandatory on another as Fig.4 illustrates.

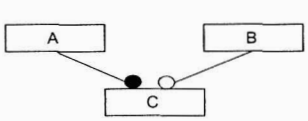

Figure 4. EFD: optional and mandatory edges [19]

We call these diagrams "Extended Feature Diagrams" (EFD).

2.7 PLUSS (PFT)

The Product Line Use case modelling for System and Software engineering (PLUSS) approach [8] is based on FeatuRSEB. It combines FD and use case diagrams to depict the high level view of a product family. FD are used to instantiate the abstract use case family model into product family models. The main characteristics of PLUSS FD are:

1. The usual FD representation conventions are changed: the type of decomposition operator is not found in the decomposed feature node anymore, nor on the departing edges but in the operand nodes: *single adaptors* (nodes with a circled 'S') represent a *xor*-decomposition of their father while *multiple adaptors* (nodes with a circled 'M') represent *or*-decomposition.
2. PLUSS FD are trees.
3. They have graphical but no mention of textual constraints.
4. Mandatory nodes are filled in black while optional ones are hollow.

An example is given in Fig. 5.

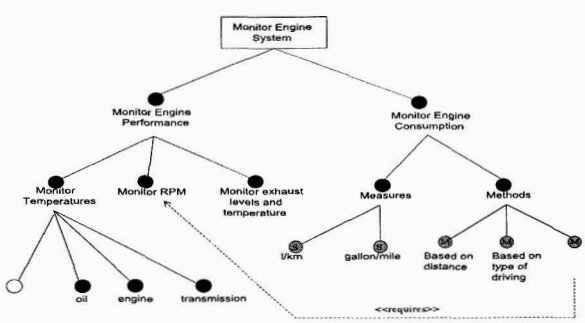

Figure 5. PFT: Monitor Engine System

We call these diagrams "PLUSS Feature Trees" (PFT).

3. Generic formal definition

3.1. From concrete to abstract syntax

The literature surveyed in the previous section mainly focuses on concrete syntax, i.e. their diagrams are mainly presented as pictures, assuming that the reader's intuition will do the rest. But actually, each reader's intuition is potentially different. Moreover, providing computer support for

these notations requires to make everything explicit. The first step in this chain is to have a bidirectional conversion from the *concrete syntax* (what the user sees), to the data structures stored in the computer, called *abstract syntax*[1].

The abstract syntax is usually a graph. It is thus essential to specify exactly what are the allowed graphs. There are two common ways to provide this information: (1) *mathematical notation* (set theory) or (2) *meta-model* (usually, a UML Class Diagram complemented with OCL constraints). In this paper, we follow [11] and adopt the former for its improved conciseness, rigour and suitability for performing mathematical proofs (see [20]). The latter might be more readable and facilitate some implementation tasks such as building a repository.

All FD are graphs whose nodes are features. They were drawn as (boxed) strings or filled circles in the concrete FD languages. Many authors use the word "feature" ambiguously, sometimes to mean a node, sometimes a leaf (a node that is not further decomposed), and sometimes as a real-world feature. Here, we use the term "feature" as a synonym of "node".

We further distinguish "primitive" and "compound features".[2][3] *Primitive features* are "features" that are of interest *per se*, and that will influence the final product. On the contrary, compound features are just intermediate nodes used for decomposition. For generality, we leave it to the modeler to define which nodes in the FD have such a purpose. Primitive features are thus not necessarily equivalent to leaves, though it is the most common case.

What we call edges is an abstraction for the arrows and line segments we find in concrete FD. Edges always relate two nodes. We further distinguish between *decomposition edges* and *constraint edges*. The former are the edges used to relate a node bearing an operator such as *or*, *xor* or *and* to one of its operand nodes. The latter are used to represent "graphical constraints" between two nodes, such as "requires" or "mutex". To determine whether a FD is a tree or a DAG, only decomposition edges are taken into account. More edge types can be found in the literature, but only edges that influence the allowed combinations of features are considered in this work.

Abstract syntax is thus about the structure that is really behind pictures. Most authors consider it so obvious that they never make it precise. Consider Fig.6 (a), a very basic FD. OFT [13] seem to hint that it should be abstracted to (b), while EFD [19] to (c). Because we want to account for both, we have to take the finer decomposition (d), adding an *opt*-node (see Section 3.2) under node R. The *opt*-node must have S as a son, thus we also add a new edge. Similar issues had to be solved to deal with filled circles (mandatory features[4]) and multiple operators attached to the same father feature as in PFT. They were also addressed by the addition of nodes and edges. As we will show in Example 4.1, this will let us highlight subtle interpretation differences between languages.

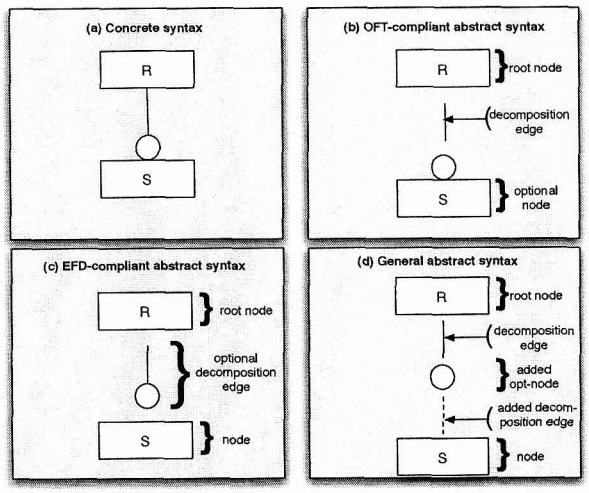

Figure 6. Three possible abstract syntaxes

3.2. Generic abstract syntax

In this section, we introduce *Free Feature Diagrams* (FFD), a parametric construction that generalizes of the syntax of all the FD variants.

FD languages vary in several respects: (1) do they consider FD as trees or DAG, (2) what are the allowed types of operators on nodes, (3) what are the allowed types of graphical constraints, (4) what are the allowed textual constraints. These are the parameters (GT,NT,GCT,TCL) of FFD:

- GT (Graph Type) is either DAG or TREE.
- NT (Node Type) is a set of boolean functions (operators). E.g.: *and* is the set of operators and_s, one for each arity s, that return true iff all their s arguments are true. Similarly, *or* (resp. *xor*) is the set of operators or_s (resp. xor_s) that return true iff some (resp. exactly one) of their s arguments is true. Operators opt_s in *opt* always return true. Operators $vp_s(i..j)$ in *card* return true iff at least i and at most j of their arguments are true. NT is usually some combination of those sets.
- GCT (Graphical Constraint Type) is a binary boolean operator. E.g.: *Requires* ($\Rightarrow$) or *Mutex* (|).

[1] If we omit implementation details such as pointer values, etc.

[2] We adopt the terminology of [1].

[3] "Compound features" are also called "decomposable features".

[4] We see them as and_1 nodes, see below.

- TCL (Textual Constraint Language) is a subset of the language of boolean formulae where the predicates are the nodes of the FD. The sublanguage used in OFT, "Composition rules" [12, p.71], is: $CR ::= p_1(requires \mid mutex)p_2$ where p_1, p_2 are primitive features.

Each FD language can be defined simply by providing values for these parameters (see Section 3.3). Semantics, on the other hand, is given once for FFD (see Section 3.4). The formal semantics of a particular FD language defined through FFD thus comes for free.

Definition 3.1 (Free Feature Diagram, or FFD) *A* FFD $d \in FFD(GT, NT, GCT, TCL) = (N, P, r, \lambda, DE, CE, \Phi)$ *where:*

- N *is its set of* nodes*;*
- $P \subseteq N$ *is its set of* primitive nodes*;*
- $r \in N$ *is the* root *of the FD, also called the* concept*;*
- $\lambda : N \rightarrow NT$ *labels each node with an operator from* NT*;*
- $DE \subseteq N \times N$ *is the set of* decomposition edges*;* $(n, n') \in DE$ *will rather be noted* $n \rightarrow n'$*;*
- $CE \subseteq N \times GCT \times N$ *is the set of* constraint edges*;*
- $\Phi \subseteq TCL$ *are the textual constraints.*

FFD collect whatever can be drawn, like a UML metamodel. FD have additional minimal well-formedness conditions.

Definition 3.2 (Feature Diagram, or FD) *A* FD *is a FFD where:*

1. *Only r has no parent:* $\forall n \in N.(\nexists n' \in N.n' \rightarrow n) \Leftrightarrow n = r$.
2. *DE is acyclic:* $\nexists n_1, ..., n_k \in N.n_1 \rightarrow \ldots \rightarrow n_k \rightarrow n_1$.
3. *If* $GT = \mathsf{TREE}$*, DE is a tree:* $\nexists n_1, n_2, n_3 \in N.n_1 \rightarrow n_2 \wedge n_3 \rightarrow n_2 \wedge n_1 \neq n_3$.
4. *Nodes are labelled with operators of the appropriate arity:* $\forall n \in N.\lambda(n) = op_k \wedge k = \sharp\{(n, n') | n \rightarrow n'\}$

The reader should keep in mind that FFD are not a user notation, that is, a notation meant to be used by analysts to draw FD. FFD are a formal framework to be used by method engineers and scientists to formally define, study or compare FD languages. FFD are intended to make these tasks much more easy and precise than before.

3.3. Specific abstract syntaxes

It is now easy to define all of the surveyed FD languages by simply providing the right parameters of FFD. The definitions are gathered in Table 1. Defining a language boils down to filling in a row of the table. The last entry (VFD) will be discussed in Section 4.

Short Name	GT	NT	GCT	TCL
OFT	TREE	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
OFD	DAG	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
RFD=VBFD	DAG	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
EFD	DAG	$card \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
GPFT	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	CR
PFT	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	$\emptyset$
VFD	DAG	$card$	$\emptyset$	$\emptyset$

Table 1. FD variants defined on top of FFD

Theorem 3.1 *We observe the following syntactical inclusions: (1)* $OFT \subset OFD \subset RFD$*, (2)* $OFT \subset GPFT \subset RFD$ *and (3)* $PFT \subset RFD$.

It is important to note that Theorem 3.1 does not say anything about the expressiveness of the languages. For this, we have to consider semantics.

3.4. Formal semantics

A formal semantics is given by a function from the *syntactic domain* of a language to a *semantic domain* [11]. The syntactic domain was given in Def. 3.1 and 3.2. Here, after some preliminary definitions, we define the semantic domain as the set of product lines (Def. 3.5, point 3) and then the semantic function (Def. 3.5, point 4).

The notion of model for a FD was introduced in [12, p.64], with the examples of models of X10 terminals.

Definition 3.3 (Model) *A* model *of a FD is a subset of its nodes:* $M = \mathcal{P}N$.

Definition 3.4 (Valid model) *[12, p.70] A model* $m \in M$ is valid for *a* $d \in FD$*, noted* $m \vDash d$*, iff:*

1. *The concept is in:* $r \in m$
2. *The meaning of nodes is satisfied: If a node* $n \in m$*, and* n *has sons* $s_1, \ldots, s_k$ *and* $\lambda(n) = op_k$*, then* $op_k(s_1 \in m, \ldots, s_k \in m)$ *must evaluate to true.*
3. *The model must satisfy all textual constraints:* $\forall \phi \in \Phi, m \vDash \phi$*, where* $m \vDash \phi$ *means that we replace each node name* n *in* ϕ *by the truth value of* $n \in m$*, evaluate*

ϕ and get true. For example, if we call ϕ_1 the CR constraint p_1 requires p_2, we say that $m \models \phi_1$ when $p_1 \in m \Rightarrow p_2 \in m$ is true.

4. *The model must satisfy all graphical constraints: $\forall(n_1, op_2, n_2,) \in CE$, $op_2(n_1 \in m, n_2 \in m)$ must be true.*

5. *If s is in the model and s is not the root, one of its parents n, called its* justification, *must be too: $\forall s \in N.s \in m \wedge s \neq r$: $\exists n \in N : n \in m \wedge n \rightarrow s$.*

Note that in [20] we also give a simplified version of the above definition for the specific case of tree languages. It has the advantage of being compositional, which highly facilitates implementation. However, it is less generic since it does not account for DAG.

Definition 3.5 (Product and Product Line, or PL) *We define:*

1. *A* product *c is a set of primitive nodes: $c \in \mathcal{P}P$.*
2. *The* product named by *a model m, noted $[\![m]\!]$, is $m \cap P$.*
3. *A* product line *(PL) pl is a set of products: $pl \in \mathcal{P}\mathcal{P}P$.*
4. *The* product line of *a FD d consists of the products named by its valid models: $[\![d]\!] = \{m \cap P | m \models d\}$*

4. Discussion

Defining a formal semantics, as we just did, provides several significant benefits.

4.1. Shared understanding of FD

First, it gives the opportunity to uncover some important issues that might well remain unnoticed with only an informal semantics. Here are a few examples:

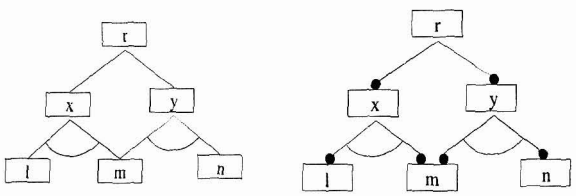

Figure 7. Node- *vs.* **edge-based semantics**

Example 4.1 *According to our semantics, the two FD in Fig. 7 are not equivalent*[5]*. The left one (in OFT syntax) has models and products $\{r, x, y, l, n\}$ and $\{r, x, y, m\}$ while the right one (in EFD syntax) has the additional models and products $\{r, x, y, l, m\}$ and $\{r, x, y, m, n\}$ which is the "edge-based" semantics hinted in [19].*

Example 4.2 *In our semantics, Fig. 3 admits a model with neither* `l/km` *nor* `gallon/mile`*. This is clearly not the semantics intended in [19]. We simply suggest to remove the harmful hollow circles.*

Example 4.3 *Our semantics also contradicts the sentence: "All mandatory features are part of all [models]"[22, p.2]. Our notion of "mandatory" is only relative to the incoming edge(s), thus a mandatory feature (node) will not be part of the model if none of its fathers is. However, in the same paper, the authors seem to agree with our interpretation. Indeed, in Fig. 8, extracted from [22, Fig. 3],* `Email` *is mandatory, but they make clear that it is not included in models where* `Net` *is not present.*

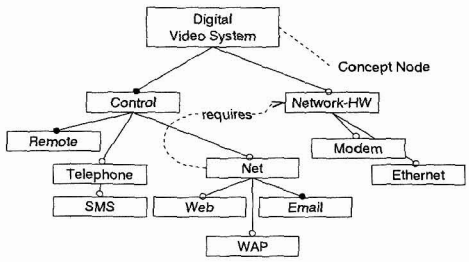

Figure 8. RFD example from [22, Fig.3]

If such issues are not made explicit and solved before a language is made public (resp. building tools), interpretations of the user (resp. developer) might differ from the intended one, possibly leading to misunderstandings and faulty developments, as already pointed out in the introduction. It is important to note, however, that we do not claim to have provided the *right* semantics for all languages. We have followed OFT, which turned out to be free of ambiguities. OFT's variants are not that precise nor ambiguity-free. We therefore had to make decisions which, maybe, are not the intended ones. But at least, we have exposed them precisely, concisely and hopefully without ambiguity. Now, they can thus be discussed constructively by their proponents and their respective user communities.

4.2. Comparing FD languages

A second advantage of formal semantics is that it makes it possible to compare languages according to formally de-

[5] Assuming all features are primitive.

fined criteria instead of loosely defined ones. In this section and in Section 4.3, we mention some of the results that were obtained on the basis of our formalization. The full developments, including mathematical proofs, can be found in [20] but are skipped here for sake of space.

One important criteria is *expressiveness*. Unlike most of the surveyed authors, we use the formal, well established definition of expressiveness of a language as the part of its semantic domain that it can express.

Definition 4.1 (Expressiveness) *The* expressiveness *of a language $\mathcal{L}$ is the set $E(\mathcal{L}) = \{[\![d]\!] | d \in \mathcal{L}\}$, also noted $[\![\mathcal{L}]\!]$. A language $\mathcal{L}_1$ is more expressive than a language $\mathcal{L}_2$ if $E(\mathcal{L}_1) \supset E(\mathcal{L}_2)$. A language $\mathcal{L}$ with semantic domain $\mathbb{M}$ (i.e. its semantics is $[\![\cdot]\!] : \mathcal{L} \rightarrow \mathbb{M}$) is* fully expressive *if $E(\mathcal{L}) = \mathbb{M}$.*

The usual way to prove that a language $\mathcal{L}_2$ is at least as expressive as $\mathcal{L}_1$ is to provide a translation from $\mathcal{L}_1$ to $\mathcal{L}_2$:

Definition 4.2 (Translation) *A* translation *is a total function $T : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ that is* correct, *i.e. preserves semantics: $[\![T(d)]\!] = [\![d]\!]$.*

The semantic domain of FD are product lines (Def. 3.5). Thus, every FD expresses a PL. Now, we can ask the converse question: Can every PL be expressed by a FD of a given kind? Stated otherwise: is this language fully expressive?

In [20], we have proved formally that this is usually false for tree languages (OFT, GPFT, PFT), and true when sharing is allowed, that is in DAG (OFD, EFD, RFD). This important point has been largely overlooked in literature. More precisely, our result shows that the disjunction of features cannot be expressed in OFT. In [10], Griss *et al.* have proposed to solve this problem by (1) adding *or*-nodes, and (2) considering FD as single-rooted DAG rather than trees. We proved that the second extension alone guarantees full expressiveness while adding *or*-nodes only does not. It should also be noted that the limited expressiveness of tree languages can be seen as a weakness of the constraint language. If nesting of constraints were allowed, Sheffer [21] has shown that *mutex* alone is complete, even on primitive features only. So, using propositional logic with feature trees as in [1] provides full expressiveness.

As we just saw, DAG are fully expressive. We thus need a finer yardstick than expressiveness to compare them. The situation is similar for programming languages, that are almost all Turing-complete and therefore have the same expressiveness. Two finer criteria are well established: *succinctness* and *naturalness* (a.k.a. *embeddability*). Intuitively, a language $\mathcal{L}_1$ is *embeddable* into $\mathcal{L}_2$ iff a semantic- and structure-preserving translation exists from $\mathcal{L}_1$ to $\mathcal{L}_2$ [9, 14]. When a language $\mathcal{L}_1$ is embeddable into a sublanguage $\mathcal{L}_2 \subset \mathcal{L}_1$ with the same expressiveness, $\mathcal{L}_1$ is said to be harmully redundant. Stated otherwise: $\mathcal{L}_1$ is unnecessarily complex because it contains at least one construct that is easily definable using only a strict subset of its own constructs. In [20], we have shown that OFD have harmfully redundant optional nodes and textual mutual exclusion constraints. Similarly, EFD have redundant optional edges and textual mutual exclusion constraints. Also, *and*-, *or*-, *xor*- and *opt*-nodes as well as *mutex* constraints are harmfully redundant wrt *vp*-nodes (see Table 2).

Instead of ...	write ...
an opt_1-node	$vp_1(0 \ldots 1)$
a xor_s-node	a $vp_s(1 \ldots 1)$
an or_s-node	a $vp_s(1 \ldots *)$
an and_s-node	a $vp_s(s \ldots s)$

Table 2. Embedding RFD into VFD

With succinctness, we formally measure the blow-up caused by a change of notation. The most important result here is that a notation that uses *vp* as node operators (e.g. EFD) is more succinct than OFD, RFD or VBFD.

These results have lead us to propose a new language, called VFD (Varied Feature Diagrams) defined so as to have the best ranking on all criteria: its is fully expressive, harmfully irredundant and can embed all other known variants and is at the same level of succinctness as EFD. It is defined on the last line of Table 1 and further discussed in the conclusion.

4.3. Implementing decision procedures

Finally, formalization also allows to clearly define some decision problems that need to be solved to automate FD related activies and to study their computational complexity. Three recurrent PL engineering activities are: (1) *modelling a PL*, (2) *determining a product* to be built and (3) *managing PL evolution*.

During activities 1 and 2, a FD is edited. In particular, in activity 2, the FD is "pruned" by adding constraints that eventually will lead to a single product. In both cases, from time to time, it is important to check whether the modified FD can still lead to some product. This problem is called checking *satisfiability* (i.e. whether $[\![d]\!] \neq \emptyset$). The problem is NP-complete for all useful fully expressively notations, but with linear-time algorithms for some degenerated cases e.g. for trees with *and*-, *xor*-, *vp*-, *or*-nodes.

For activity 2, it is also possible for the customer or market expert to select in advance the features of the product and to check whether it is allowed or not in the PL, i.e. whether it contains feature interferences. This problem ($c \in [\![d]\!]$) is called *product checking* and is also NP-complete.

The evolution of a PL (activity 3) can also be supported by tools. For example, one might want to refactor an overly complex FD into a simpler one. Or, two (versions of the same) FD representing the same PL that coexist must now be merged. In both cases, it is important to be able to check the equivalence of two FD ($[\![d_1]\!] = [\![d_2]\!]$). Deciding it turned out to be Π_1-complete[6].

Evolution also makes it necessary to be able to *merge* PL but this has a rather broad meaning. More rigorously, and depending on how conservative the engineers want to be about the result, one of three operations must be chosen: *intersection*, *union* (a.k.a. disjunction) or *reduced product* of two FD. Intersection ($[\![d_1]\!] \cap [\![d_2]\!]$) excludes all the feature interactions already excluded in each of the source FD. It is the safest option and can be used, for example, to combine the contributions of two feature interference engineers who have worked in parallel. Conversely, union ($[\![d_1]\!] \cup [\![d_2]\!]$) creates a FD that encompasses all the products of the two originating PL. It can be judged safe in some cases but may require interference checking in critical settings. A third option is the reduced product ($\{c_1 \cup c_2 | c_1 \in [\![d_1]\!], c_2 \in [\![d2]\!]\}$): it delivers a FD that not only accepts all the products from each source PL but it also generates all possible new feature combinations. The validity of these new combinations must of course be checked by feature interference engineers. The remaining combinations will allow offering the customer a wider choice. All three operations (intersection, union and reduced product) produce a FD and were proved to be computable in linear time and size.

Still in the context of activity 3, it may be decided at some point to change the FD language used for the PL project, for convenience or for merging it with another project that uses another language. Algorithms to translate between FD languages can be derived immediately from our embeddability and succinctness results (see e.g. Table 2). It is also important to note that these can also be applied prior to equivalence checking or merging which implies that these latter operations can be easily carried out between two FD written in different languages.

5. Related work

In this paper, we have studied the formal underpinnings of a family of languages in the line of the original feature trees [12], that have not yet received a formal definition, but where a product line semantics is clearly intended. More recently, however, a few more formally defined notations have surfaced. We briefly examine them.

van Deursen *et al.* [23] define the semantics of a textual FD language by coding rewriting rules in the ASF+SDF meta-environment. Hence, contrary to ours, their semantics is not tool-independent, nor self-contained. Another major difference is that they preserve the order of features.

Batory [1] provides a translation of FD to both grammars and propositional formulae. His goal is to use off-the-shelf Logic-Truth Maintenance Systems and SAT solvers in feature modelling tools. The semantics of grammars is a set of strings, and thus order and repetition are kept. The semantics of propositional formulae is closer to our products but [1] differs in two respects: (i) decomposable features are not eliminated, and (ii) the translation of operators by an equivalence leads to (we suspect) a counter-intuitive semantics.

In [6], Czarnecki *et al.* define a new FD language to account for staged configuration. They introduce *feature cardinality* (the number of times a feature can be repeated in a product) in addition to the more usual (*group*) cardinality. Foremost, a new semantics is proposed where the full shape of the unordered tree is important, including repetition and decomposable features. The semantics is defined in a 4-stage process where FD are translated in turn into an extended abstract syntax, a context-free grammar and an algebra. In [5], the authors provide an even richer syntax. The semantics of the latter is yet to be defined, but is intended to be similar to [6].

Benavides *et al.* [2] propose to use constraint programming to reason on feature models. They extend the FD language of [6] with extra-functional features, attributes and relations between attributes. Subsequently, they describe tool-support based on mapping those FD to Constraint Satisfaction Problems (CSP).

In general, we can argue that related approaches do not rank as good as ours on generality, abstraction, conciseness and intuitiveness. For some approaches [23, 1, 2] the semantics is tool-driven, while on the contrary tools should be built according to a carefully chosen and well-studied semantics. For the others [6, 5], we could not find a justification. Our approach is justified by our goals: make fundamental semantic issues of FD languages explicit in order to study their properties and rigorously evaluate them *before* adopting them or implement CASE tools. A finer comparison of the semantic options taken in the aforementioned related approaches wrt ours is a topic for future work.

A current limit of our endeavour is the scope of formalization. As mentioned repeatedly, we focussed on PL semantics i.e. what are the allowed and forbidden feature combinations. Further formalization of aspects such as layers [13] or binding times [24] should be further investigated.

Moreover, formality is only one specific quality of languages. For example, Krogstie [15] proposes a comprehensive quality framework covering such notions as *domain appropriateness* (i.e. is the language considered adequate to convey the relevant information on the subject domain). A complete evaluation and comparison of FD languages should also take such aspects into account. To the best of

[6] Π_1 is the first level of the polynomial hierarchy, just above NP.

our knowledge, this remains to be done.

6. Conclusion

Throughout this paper, we have tried to clarify feature diagrams. Feature diagrams are a popular and valuable tool to engineer software product lines. In particular, they are central to the requirements engineering process and are used by a variety of stakeholders. However, after surveying past research, we have concluded that most of it had not looked enough at the foundations necessary to avoid ambiguities and to build safe and efficient tools. We have thus proposed a generic formalization of the syntax and semantics of feature diagrams which, we hope, can be the basis for more rigorous investigations in the area. We have started such investigations ourselves by formally comparing languages and studying decision procedures. On these grounds, a natural further step is the development of tools. Such tools will be able to operate on multiple feature diagram languages and support a large spectrum of activities. They will be centered around VFD (Varied Feature Diagrams), suggested in this paper, but which need more work in several respects including a concrete syntax and empirical validation.

References

[1] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In Obbink and Pohl [16], pages 7–20.

[2] D. Benavides, P. Trinidad, and A. R. Cortés. Automated Reasoning on Feature Models. In O. Pastor and J. F. e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2005.

[3] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux. Semantics of feature diagrams. In T. Männistö and J. Bosch, editors, *Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support)*, Boston, August 2004.

[4] S. Cohen, B. Tekinerdogan, and K. Czarnecki. A case study on requirement specification: Driver Monitor. In *Workshop on Techniques for Exploiting Commonality Through Variability Management at the Second International Conference on Software Product Lines (SPLC2)*, 2002.

[5] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. *Software Process Improvement and Practice, special issue on Software Variability: Process and Management*, 10(2):143 – 169, 2005.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[7] U. W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[8] M. Eriksson, J. Börstler, and K. Borg. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In Obbink and Pohl [16], pages 33–44.

[9] M. Felleisen. On the expressive power of programming languages. In N. D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151. Springer Verlag, 1990.

[10] M. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, June 1998.

[11] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[13] K. C. Kang, S. Kim, J. Lee, and K. Kim. FORM: A Feature-Oriented Reuse Method. In *Annals of Software Engineering 5*, pages 143–168, 1998.

[14] S. C. Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, Amsterdam, 1952.

[15] J. Krogstie. A Semiotic Approach to Quality in Requirements Specifications. In *Organizational Semiotics*, pages 231–249, 2001.

[16] J. H. Obbink and K. Pohl, editors. *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, july 2005.

[18] M. Riebisch. Towards a More Precise Definition of Feature Models. *Position Paper. In: M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines*, 2003.

[19] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA, June 2002.

[20] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic Semantics of Feature Diagrams. Technical report, Computer Science Institute, University of Namur, 2005.

[21] H. M. Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Trans. AMS*, 14:481–488, 1913.

[22] D. Streitferdt, M. Riebisch, and I. Philippow. Formal Details of Relations in Feature Models. In *Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03), Huntsville, Alabama, USA, April 7-11, 2003*, pages 297–304. IEEE Computer Society Press, 2003.

[23] A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[24] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.