Comparative Semantics of Feature Diagrams: FFD vs vDFD
Trigaux, Jean-Christophe; Heymans, Patrick; Schobbens, Pierre-Yves; Classen, Andreas

Published in:
Proceedings of the Workshop on Comparative Evaluation in Requirements Engineering (CERE'06) held in conjunction with the 14th IEEE International Requirements Engineering Conference (RE'06)

Publication date:
2006

Document Version
Early version, also known as pre-print

Link to publication
Citation for published version (HARVARD):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. May. 2021
Comparative semantics of Feature Diagrams: FFD vs. vDFD

Jean-Christophe Trigaux, Patrick Heymans, Pierre-Yves Schobbens, Andreas Classen
University of Namur, Computer Science Department
5000 Namur, Belgium
{jtr,phe,pys,aclassen}@info.fundp.ac.be

Abstract

Feature Diagrams are a popular family of modelling languages used for engineering requirements in software product lines. In our previous research, we advocated the use of formal semantics as an indispensable means to clarify discussions about feature diagrams and to facilitate safe and efficient tool automation. We presented a generic formal semantics for feature diagram languages and criteria to compare them. However, other formal semantics exist. We already informally argued in favour of our semantics which, we think, is more abstract, more concise and not tool dependent. However, some of these claims needed to be further objectified. The purpose of this paper is to compare the semantics proposed by van Deursen and Klint with our own following the methodology of comparative semantics. To be made amenable to comparison, van Deursen and Klint’s tool-based definition is first recalled and redefined by correcting some minor mistakes. Their semantics is then mapped to ours through an abstraction function. We then proceed to compare the expressiveness, embeddability and succinctness of both approaches. The study tends to confirm our semantic choices as well as our tool-independent methodology. It also demonstrates that van Deursen and Klint’s language is fully expressive and provides various results likely to help tool developers, especially for implementing model transformations.

1 Introduction

Central to the Product Line (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 artifacts [21]. Variability at the requirement level is commonly modelled through Feature Diagrams (FD). In the last decade, research and industry have developed several FD languages. The first and seminal proposal was introduced as part of the FODA method back in 1990 [16]. An example of a FODA FD is given in Fig. 1. It is inspired from a case study defined in [6] 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 Fig. 1, the node Monitor Engine System is called the root, or concept. The nodes can be mandatory or optional. Optional nodes are represented with a hollow circle above their name, e.g. Coolant. In FODA, mandatory nodes are the ones without a hollow circle (in some other syntaxes [12, 23, 22, 7, 10, 8], they are represented with filled circles). The edges are used to progressively decompose nodes into more detailed features. In FODA, there were only and- and xor- decompositions like illustrated in Fig. 1:

1. 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.

2. xor-decomposition where edges are linked by a line segment, as between Measures and its sons, 1/km and Miles/gallon, indicating that only one of them should be present in combinations where Measures is.

Figure 1. FODA: Monitor Engine System
Since Kang et al.’s initial proposal, several extensions have been devised as part of the following methods: FORM [17], FeatureRSEB [12], Generative Programming [10], PLUSS [11], and in the work of the following authors: Riebisch et al. [23, 22], van Gurp et al. [28], van Deursen and Klint [27], Czarnecki et al. [7, 8], Batory [1] and Benavides et al. [2].

Most of these authors [17, 12, 10, 23, 22, 28, 11] present their semantics by the way of examples. Still, most of them have argued for an “improved expressiveness”. However, without a formal semantics, they have failed to demonstrate it.

In previous publications [4, 26, 25], we have developed and applied a rigorous framework to assess those claims. We have first carried out a comprehensive survey of the informal FD variants. We have generalized their various syntaxes through a generic construction called Free Feature Diagrams (FFD). We gave a formal semantics to FFD, thus providing a (hopefully) unambiguous and very concise definition for all the surveyed FD variants. 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. However, having a proper formal semantics remains extremely important. As remarkably argued in [13], formal semantics is the best way to avoid ambiguities and to start building safe automated reasoning tools. Without a formal semantics, new FD languages might continue to proliferate on the basis of shallow or erroneous motivations, leading to interpretation and interoperability problems.

In [4, 26, 25], we argued that FFD contribute to improve the definition, understanding, comparison and reliable implementation of FD languages. In particular, we have highlighted some subtle points in the interpretation of FD. Additionally, we have defined the main decision problems that a FD tool should implement, i.e. we gave a specification of such tools, and subsequently, we studied their minimal execution time.

Some authors have also started to better define their semantics [27, 8, 7, 1]. However, we found that these semantics are less general, abstract and concise. These approaches typically transform FD to other formalisms (for which tools exist). This naturally gives a more complex transformation and a less abstract semantics. It has the dubious advantage that the transformation is correct by definition. On the contrary, we believe that tools should be built or chosen according to a natural, carefully chosen and well-studied semantics. 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.

We are now in position to proceed to our next phase of study sketched in [26, 25]: given well-defined FD languages, we can start a meaningful discussion of their merits, following the well-established scientific method called comparative semantics. In this paper, we compare the semantics of FFD with the one defined by van Deursen and Klint in [27] which is apparently the first first formal semantics of FD to have been published. For brevity, we call their variant of FD, vDFD (van Deursen and Klint’s Feature Diagrams).

This paper is structured as follows. In Sec. 2, we will describe the method that we use to compare formal semantics. FFD are then briefly presented in Sec. 3. In Sec. 4, we will recall the semantics of vDFD given in [27] and compare it with our own. The comparison of both formalisms appears in Sec. 5. Finally, related works are examined in Sec. 6 and conclusions are given in Sec. 7.

2 Research method

A proper definition of a formal semantics is preferably done in several steps. The first step in this chain is to have a bidirectional conversion from the concrete syntax (what the user sees), to abstract data structures, called abstract syntax. Indeed, the concrete syntax is usually too cumbersome to be reasoned on efficiently, be it by humans or tools.

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 [13] and adopt the former for its improved conciseness, rigour and suitability for performing mathematical proofs. The latter might be more readable and facilitate some implementation tasks such as building a repository.

The second step is to provide a formal semantics, i.e. a function from the graphs above to a mathematical structure chosen for being as close as possible to our intuitive understanding. This structure forms the semantic domain. In [3] (and in Def. 3.5 of this paper) we proposed as semantic domain the one of Product Lines (PL) defined as set of products, where a product is characterized by the primitive features it includes.

The works to which we compare often do not follow this methodology, but are amenable to it. For instance, [1] defines a transformation to grammars and propositional formulae. Fortunately, these two formalisms are provided with a standard semantics, so that we can obtain a semantics by composing the transformation followed by the standard semantics. In all similar approaches, the semantic domain of the formalisms used were not designed for features. They are thus usually less abstract: they keep too much syntactic
information, so that fewer diagrams are considered equivalent. To discard this syntactic information, we must introduce abstraction functions between semantic domains (see Fig. 2). As we progress in our comparative semantics study, we will construct a category of semantic domains linked by abstraction functions. The comparative semantics of specification languages [19] of logic programming, of concurrent programming [9] or of coordination languages [5], for instance, is already well developed and integrates developments and tools from different languages.

Given this, we can then evaluate which FD languages are more expressive, succinct, or natural [3] more rigorously. The technical tool for this study are translations or embeddings (see Fig. 2), i.e. transformations (functions between abstract syntaxes) that preserve semantics.

As illustrated in Fig. 2, for each FD language, say X, with an already existing formal semantics, their semantic domain (XSD) and abstract syntax (XFD) should be compared with our semantic domain (PL, see Def. 3.5) and the abstract syntax of FFD (Def. 3.1), respectively, in order to derive abstraction functions and embeddings. Moreover, these translations compose, so that it is useful to have our category of semantic domains, and look at its shape so that most results follow by composition.

![Figure 2. Comparative Semantics Approach](image)

We further distinguish “primitive” and “compound features”\(^3\). 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.

**Decomposition edges** relate a father node f to a son node s and are noted f → s.

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), at most one per arity. 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\(_i\) in opt always return true. Operators \(v_{p_i(i...j)}\) 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 (⇒) or Mutex (\(\land\)).
- **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 FODA FD, “Composition rules” [16, p.71] is: \(CR := p_1(rquires | mutex)p_2\) where \(p_1, p_2\) are primitive features.

The syntactic domain of a particular FD language can be defined simply by providing values for these parameters. For example, the abstract syntax of FODA FD is defined as \(FFD(TREE, and \cup xor \cup \{opt\}_1, 0, CR)\). In [4], the abstract syntax of other FD variants, including [17, 12, 10, 23, 22, 28, 11] is defined similarly. As we will see in Sec. 4.2, van Deursen and Klint’s abstract syntax is defined as \(FFD(DAG, and \cup xor \cup or \cup \{opt\}_1, 0, CR′)\) where \(CR′ := p_1(rquires | excludes)p_2 | (include | exclude)p\).

The semantics is defined only once for FFD [26, 25], reproduced in Sec. 3.2. The formal semantics of a particular FD language defined through FFD thus comes for free.

---

\(^2\)We adopt the terminology of [1].

\(^3\)“Compound features” are also called “decomposable features”.

---

3 Free Feature Diagrams: FFD

In this section, for the paper to be self-contained, we recall Free Feature Diagrams (FFD) from [26, 25], a parametric construction that generalizes the syntax of FD languages and represents a family of FD languages. We define it according to our research method.

3.1 Abstract syntax

All FD are graphs whose nodes are features. They were drawn as (boxed) strings 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”. 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.

**Decomposition edges** relate a father node f to a son node s and are noted f → s.

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), at most one per arity. 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\(_i\) in opt always return true. Operators \(v_{p_i(i...j)}\) 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 (⇒) or Mutex (\(\land\)).
- **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 FODA FD, “Composition rules” [16, p.71] is: \(CR := p_1(rquires | mutex)p_2\) where \(p_1, p_2\) are primitive features.

The syntactic domain of a particular FD language can be defined simply by providing values for these parameters. For example, the abstract syntax of FODA FD is defined as \(FFD(TREE, and \cup xor \cup \{opt\}_1, 0, CR)\). In [4], the abstract syntax of other FD variants, including [17, 12, 10, 23, 22, 28, 11] is defined similarly. As we will see in Sec. 4.2, van Deursen and Klint’s abstract syntax is defined as \(FFD(DAG, and \cup xor \cup or \cup \{opt\}_1, 0, CR′)\) where \(CR′ := p_1(rquires | excludes)p_2 | (include | exclude)p\).

The semantics is defined only once for FFD [26, 25], reproduced in Sec. 3.2. 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;
- \( \lambda : N \to NT \) 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 \( n \to n' \);
- \( CE \subseteq N \times GCT \times N \) is the set of constraint edges;
- \( \Phi \subseteq TCL \) are the textual constraints.

FD collect whatever can be drawn. So, 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: \( \forall n \in N. (\exists n' \in N.n' \to n) \iff n = r. \)
2. \( DE \) is acyclic: \( \forall n_1, ..., n_k \in N.n_1 \to \ldots \to n_k \to n_1. \)
3. If \( GT = TREE \), \( DE \) is a tree: \( \forall n_1, n_2, n_3 \in N.n_1 \to n_2 \land n_3 \to n_2 \land n_1 \neq n_3. \)
4. Nodes are labelled with operators of the appropriate arity: \( \forall n \in N. \lambda(n) = op_k \land k = \sharp((n,n')|n \to n'). \)

3. Semantics

A formal semantics is given by a function from the syntactic domain of a language to a semantic domain [13]. 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 (PL)(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 [16, 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: \( M \subseteq N. \)

Definition 3.4 (Valid model) [16, p.70] A model \( m \in M \) is valid for a \( d \in FD \), noted \( m \models d \), 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.

4 van Deursen and Klint’s Feature Diagrams: vDFD

van Deursen and Klint have formalized FD, by providing [27]:

1. A syntax presented as a feature description language that we will call vDFD. The authors also provided a feature definition (Def. 4.1) and a grammar for vDFD (Def. 4.3);
2. A semantics presented as a feature diagram algebra defining various sets of rules manipulating vDFD:
   (a) Normalization rules (\( \mathcal{N} \)) to eliminate duplicate features and degenerate cases of the various constructs (Def. 4.5);
   (b) Variability rules to count the number of products allowed in a FD;
   (c) Expansion rules (\( \mathcal{E} \)) to expand a normalized feature expression into a disjunctive normal form (Def. 4.6);
   (d) Satisfaction rules (\( \mathcal{S} \)) to determine which feature expressions in disjunctive normal form satisfy the feature constraints (Def. 4.7);
In Fig. 3, we show the sequence of these transformations \((N,E,S)\) on vDFD as proposed in [27]. An alternative sequence of transformations \((N',E',S')\) on vDFD is shown which refers to the small corrections we propose for each of them. In the next sections, we will give the reader more details on this. First, we will present the formalization for vDFD proposed in [27] (Sec. 4.1), then we will discuss their abstract syntax (Sec. 4.2) and revisit their semantics (Sec. 4.3). Finally, we compare the revisited semantics with our own (Sec. 5).

![Figure 3. van Deursen and Klint's semantics](image)

### 4.1 van Deursen and Klint’s original definition

The primary objective of van Deursen and Klint was to reason on FD using a textual representation rather than a graphical one. Further requirements for this representation were [27]:

1. to contain all the information contained in the graphical form,
2. to be suited for automatic reasoning.

To satisfy these requirements, the authors first produced a feature definition (Def. 4.1). Then they proposed a grammar for their textual FD (Def. 4.3). Finally, they proposed rules to reason on this representation. Rules are used to check the consistency of the representation. \(N\) and \(E\) are used to generate a normal form (syntactic consistency). Rules for computing variability are also defined but are not relevant here as they do not influence the semantics. All these rules are defined and justified in [27]. To present these definitions we reuse the variable naming convention proposed by van Deursen and Klint (Table 1). All the reasoning proposed by the authors is based on their disjunctive normal form (Def. 4.4).

#### Definition 4.1 (Feature definition)

A feature definition [27, p.4] is a feature name followed by “:” and a feature expression (Def. 4.2).

#### Definition 4.2 (Feature expression)

A feature expression [27, p.4] can consist of

1. an atomic feature,
2. a composite feature: a named feature whose definition appears elsewhere,
3. an optional feature: a feature expression followed by ?,
4. mandatory features: a list of feature expressions enclosed in all( ),
5. alternative features: a list of feature expressions enclosed in one-of( ),
6. non-exclusive selection of features: a list of feature expressions enclosed in more-of( ),
7. a default feature value: default = followed by an atomic feature,
8. features of the form ..., indicating that a given set is not completely specified.

#### Definition 4.3 (vDFD Grammar)

A vDFD Grammar [27, p.6] is defined by:

\[
\begin{align*}
\text{FeatureName} & \rightarrow [A-Z][a-zA-Z0-9]* \\
\text{AtomicFeature} & \rightarrow \text{FeatureName} \\
\text{FeatureDefinition} & \rightarrow \text{FeatureExpr} \\
\text{Constraint} & \rightarrow \text{DiagramConstraint} \\
\text{FeatureExpr} & \rightarrow (\text{FeatureExpr} , )+ \\
\text{FeatureDiagram} & \rightarrow \text{all(FeatureList)} \\
\text{FeatureList} & \rightarrow \text{one-of(FeatureList)} \\
\text{one-of} & \rightarrow \text{FeatureExpr} \\
\text{FeatureName} & \rightarrow \text{FeatureExpr} \\
\text{FeatureExpression} & \rightarrow \text{FeatureExpr} \\
\text{FeatureRequire} & \rightarrow \text{FeatureExpr} \\
\text{default = AtomicFeature} & \rightarrow \text{FeatureExpr} \\
\text{...} & \rightarrow \text{AtomicFeature} \\
\text{Constraint} & \rightarrow \text{DiagramConstraint} \\
\text{UserConstraint} & \rightarrow \text{constraint} \\
\text{include AtomicFeature} & \rightarrow \text{UserConstraint} \\
\text{exclude AtomicFeature} & \rightarrow \text{UserConstraint} \\
\end{align*}
\]

#### Definition 4.4 (Disjunctive normal form)

A disjunctive normal form [27, p.9] is a one-of feature expression with only all feature expressions as arguments themselves with only atomic features as arguments. A disjunctive normal form is an expression of the form: one-of(all(A1, ..., An), ..., all(Am1, ..., Amn))
This disjunctive normal form indicates explicitly all possible feature combinations. It is obtained by applying the normalization \( N \) and expansion rules \( E \). For instance, \( (N_2) \) removes duplicate features; \( (N_3) \) transforms a one-of expression containing one optional feature into an optional one-of expression; \( (E_4) \) translates an all( ) expression containing a more-of expression into three cases: one with the first alternative, one with the first alternative and the remaining more-of expression, and one with only the remaining more-of expression.

**Definition 4.5 (Normalization rules)** The set of normalization rules [27, p.7] is \( N = \{N_1, \ldots, N_{12}\} \):

\[
\begin{align*}
(N_1) & \quad Fs, F, Fs', F?; Fs'' = Fs, F, Fs', Fs' \\
(N_2) & \quad Fs, F, Fs'; Fs'' = Fs, F, Fs', Fs'' \\
(N_3) & \quad F?? = F? \\
(N_4) & \quad all(F) = F \\
(N_5) & \quad all(Fs, all(Ft), Fs') = all(Fs, Ft, Fs') \\
(N_6) & \quad one-of(F) = F \\
(N_7) & \quad one-of(Fs, one-of(Ft), Fs') = one-of(Fs, Ft, Fs') \\
(N_8) & \quad one-of(Fs, F?, Fs') = one-of(Fs, F, Fs')? \\
(N_9) & \quad more-of(F) = F \\
(N_{10}) & \quad more-of(Fs, more-of(Ft), Fs') = more-of(Fs, Ft, Fs') \\
(N_{11}) & \quad more-of(Fs, F?, Fs') = more-of(Fs, F, Fs')? \\
(N_{12}) & \quad default = A = A
\end{align*}
\]

**Definition 4.6 (Expansion rules)** The set of expansion rules [27, p.9] is \( E = \{E_1, \ldots, E_4\} \):

\[
\begin{align*}
(E_1) & \quad all(Fs, F?, Ft) = \\
& \quad one-of(all(Fs, F, Ft), all(Fs, Ft)) \\
(E_2) & \quad all(Ft, F?, Fs) = \\
& \quad one-of(all(Ft, F, Fs), all(Ft, Fs)) \\
(E_3) & \quad all(Fs, one-of(Ft, Fs') = \\
& \quad one-of(all(Fs, F, Fs'), all(Fs, one-of(Ft, Fs'))) \\
(E_4) & \quad all(Fs, more-of(Ft, Fs') = \\
& \quad one-of(all(Fs, F, Fs'), all(Fs, F more-of(Ft, Fs'), all(Fs, more-of(Ft, Fs'))) \\
\end{align*}
\]

On this disjunctive normal form, satisfaction rules (Def. 4.7) are applied to eliminate products that do not satisfy the constraints. These satisfaction rules use the two following functions (not explicitly defined in [27]):

- \( isElement : AtomicFeature \times FeatureExpr \rightarrow \mathcal{B} \) which determines whether the AtomicFeature is contained in the FeatureExpr or not.
- \( sat : FeatureExpr \times Constraints \rightarrow \mathcal{B} \) which determines whether the FeatureExpr satisfies the Constraints or not.

If no constraint is applicable to the feature expression then \( (S_9) \) is used. Otherwise, binary constraints (requires, excludes) and unary constraints (include, exclude) are respectively handled by:

- \( (S_1) \) and \( (S_2) \) for excludes;
- \( (S_3) \) and \( (S_4) \) for requires;
- \( (S_5) \) and \( (S_6) \) for include;
- \( (S_7) \) and \( (S_8) \) for exclude.

For instance, \( (S_6) \) defines that the \( sat \) function must return \( false \) if the constraint \( Include \ A \) is applicable and if the atomic feature \( A \) is not an element of the FeatureExpr \( Ft \).

**Definition 4.7 (Satisfaction rules)** The set of satisfaction rules [27, p.13] is \( S = \{S_1, \ldots, S_{19}\} \), where \( \mid \) means \( OR \):

\[
\begin{align*}
(S_1) & \quad isElement(A_2, F) \mid isElement(A_2, F\ s') = true \\
& \quad sat(all(Fs, A_1, F\ s'), Cs A_1 excludes A_2 Cs\ s') = false \\
(S_2) & \quad isElement(A_2, F) \mid isElement(A_2, F\ s') = false \\
& \quad sat(all(Fs, A_1, F\ s'), Cs A_1 excludes A_2 Cs\ s') = \\
& \quad sat(all(Fs, A_1, F\ s'), Cs Cs\ s') = \\
(S_3) & \quad isElement(A_2, F) \mid isElement(A_2, F\ s') = false \\
& \quad sat(all(Fs, A_1, F\ s'), Cs A_1 requires A_2 Cs\ s') = false \\
(S_4) & \quad isElement(A_2, F) \mid isElement(A_2, F\ s') = true \\
& \quad sat(all(Fs, A_1, F\ s'), Cs A_1 requires A_2 Cs\ s') = \\
& \quad sat(all(Fs, A_1, F\ s'), Cs Cs\ s') = \\
(S_5) & \quad isElement(A, Ft) = true \\
& \quad sat(all(Ft), Cs include A Cs\ s') = sat(all(Ft), Cs Cs\ s')
\end{align*}
\]
\[
\text{(S}_6\text{)} \quad \text{orElse}(A, F, C) = \text{false} \\
\text{sat}(\text{all}(F), C, s) = \text{true} \\
\text{orElse}(A, F, C) = \text{true} \\
\text{sat}(\text{all}(F), C, s) = \text{false}
\]

\text{Definition 4.9 (vDFD Semantics)} The semantics of a vDFD \( \text{vDFD}(vdfg) \) is a function \( \mathcal{L} : \text{vDFD} \rightarrow \mathcal{O}(\mathcal{P}) \) where \( \mathcal{L}(\text{vDFD}) = \mathcal{S}(\mathcal{G}(\text{N}(vdfg))) \).

Nevertheless, we have discovered undesirable semantics due to missing rules. Consequently, we provide some additional rules and justify them:

- The rule in \( N_1 \) is not sufficient to avoid feature lists that combine mandatory and optional features. Indeed, a feature list such as \( F_s, F_s', F_s'', \ldots \) (where \( F \) and \( F \) are switched wrt the rule \( N_1 \)) would be considered normalized. The set of normalization rules should be corrected by adding one simple rule (Def. 4.10);

- The set of expansion rules is not sufficient to produce a correct disjunctive normal form. Indeed, terms of the form \( a \) or \( \text{one-of}(F_s) \) or \( \text{all}(F_s) \) are allowed. In order to respect the intentions of the authors we extend \( \mathcal{E} \) (Def. 4.11);

- The satisfaction function (sat) is never explicitly called. Consequently, we propose one rule (Def. 4.12) that calls this function and eliminates invalid products (products which do not satisfy the constraints).

\text{Definition 4.10 (Normalization rules)} The normalization rules are a set of rules \( N' = N \cup \{N_{13}\} \) where

\[
(N_{13}) \quad F_s, F_s', F_s'', \ldots = F_s, F_s', F_s''
\]

\text{Definition 4.11 (Expansion rules)} The expansion rules are a set of rules \( \mathcal{E}' = \mathcal{E} \cup \{\mathcal{E}_5, \mathcal{E}_6\} \) where

\[
\text{(E}_5\text{)} \quad A = \text{all}(A) \\
\text{(E}_6\text{)} \quad \text{all}(F_s) = \text{one-of}(\text{all}(F_s))
\]

\text{Definition 4.12 (Satisfaction call rule)} The satisfaction rules are a set of rules \( \mathcal{S}' = \mathcal{S} \cup \{\mathcal{S}_{10}\} \) where

\[
\text{(S}_{10}\text{)} \quad \text{sat}(\text{all}(F_s), C_s) = \text{sat}(\text{one-of}(F_s'), \text{all}(F_s), C_s)
\]

Having revisited van Deursen and Klint’s rewriting rules we need to redefine the semantics function (Def. 4.13).

\text{Definition 4.13 (Revisited vDFD Semantics)} The redefined semantics function of van Deursen and Klint is defined as:

\[
\mathcal{L}' : \text{vDFD} \rightarrow \mathcal{O}(\mathcal{P}) \text{ where } \mathcal{L}'(\text{vDFD}) = \mathcal{S}(\mathcal{G}(\text{N}(\text{vDFD})))
\]

\section{Comparison}

We will now compare the redefined semantics of vDFD with the one of FFD (Section 5.1) and subsequently define further comparison criteria and examine the languages with respect to those criteria (Section 5.2).
5.1 Comparative Semantics

Even with the modifications of vDFD that we proposed, when we compare their semantics (Def. 4.13) with the one of FFD (Sec. 3.2) two points of divergence appear:

- First, the semantic domains are different (see Fig. 3):
  1. On the one hand, the semantics of vDFD translates a vDFD expression into another expression in disjunctive normal form which is an ordered list of ordered lists of atomic features $O(O(P))$.
  2. On the other hand, the semantic domain of FFD is a set of sets of primitive features $\mathcal{P}(\mathcal{P}(P))$.

- Second, van Deursen and Klint’s semantics always gives preference to inclusion in terms (not in constraints) and thus behaves like a semantics based on edges rather than on nodes.

Let us examine these two points in turn. Although the semantic domains are different, they can be easily related by an abstraction function. Indeed, in [27], atomic features directly correspond to primitive features. Hence, the remaining difference is the one between the mathematical structures list and set. The order of atomic features is important in van Deursen and Klint’s semantics. For instance, the textual vDFD expression one-of(a11(A, B), a11(B, A)) contains two products: a11(A, B) and a11(B, A). In FFD’s semantics, only one product would be part of PL: {A, B}. Consequently, we can define an obvious abstraction function $R$ (definition 5.1) that simply abstracts away order from van Deursen and Klint’s semantic domain and directly maps it to our semantic domain. We do not know if this notion of order between features was deliberate or not, but intuitively we consider that two products with the same features should be identical. However, ordering features could be relevant, for example, when each feature corresponds to one transformation (on code or models) [24] and these transformations do not produce the same result if they are applied in a different order.

**Definition 5.1 (Semantic Domain Abstraction $R$)**

$R : O(O(P)) \rightarrow \mathcal{P}(\mathcal{P}(P))$

$R(a11(f_n,\ldots,f_m),\ldots,a11(f'_n,\ldots,f'_m)) = \{\{f_n,\ldots,f_m\},\ldots,\{f'_n,\ldots,f'_m\}\}$

Nevertheless, this abstraction function is not sufficient to find an equivalence between our semantics and van Deursen and Klint’s semantics. Indeed, the latter implicitly always gives preference to inclusion in terms, which is also a characteristic of edge-based semantics (as opposed to node-based semantics). For instance, contrary to a node-based semantics and classical intuition, an edge-based semantics for the FD illustrated in Fig. 4 will consider as valid the products \{A,B,C,D,E\} and \{A,B,C,E,F\}. The solution to find a semantic equivalence between both semantics (Theorem 5.1) is to apply a preliminary transformation $T$ on the FFD representation of a textual vDFD. The idea behind the transformation $T$ is to replace each atomic feature shared by several fathers with one and-node for each incoming edge, each of these and-nodes having only one son which is the corresponding atomic feature. In our concrete notation, to obtain an edge-based semantics, we add one mandatory circle (and-node with one son) for each incoming edge of a shared feature (see Fig. 4).

![Figure 4. From node-based to edge-based Semantics](image)

**Theorem 5.1**

$\forall t \in vDFD : \|T(t)\| = R(L'(t))$

5.2 Comparison Criteria

Now that we have aligned the definitions of the two languages, we can start to compare them based on various formal criteria. Three important criteria that we have already studied for other notations [26] are expressiveness, embeddability and succinctness.

FFD and vDFD can not be compared directly with these comparison criteria. Indeed, every possible FD language defined through FFD potentially evaluates differently. Ideally, every FFD-based language should be compared with vDFD separately, which is by far out of the scope of this paper. Hence, in the following, we will introduce the criteria and, for each of them, discuss the comparison between vDFD and one or more representative members from the FD language family that we already studied. More studies could come in the future to complement these results.

5.2.1 Expressiveness

We use the formal, well established, definition of expressiveness of a language, as the part of its semantic domain that it can express.
Definition 5.2 (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 \( \mathcal{M} \) (i.e., its semantics is \( \{ [.] : \mathcal{L} \to \mathcal{M} \} \)) is expressively complete if \( E(\mathcal{L}) = \mathcal{M} \).

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 5.3 (Translation) A translation is a total function \( T : \mathcal{L}_1 \to \mathcal{L}_2 \) that is correct, i.e., preserves semantics: \( \| T(D_1) \| = \| D_1 \| \).

Given that we have the domain abstraction function \( \mathcal{R} \), we can consider that the semantic domain of vDFD are also product lines (Def. 3.5). Thus, every vDFD expresses a PL. Now, we can ask the converse question: can every PL be expressed by a vDFD? Stated otherwise: are vDFD fully expressive?

In [26], we have examined this question for several FD languages [16, 17, 12, 10, 11, 23, 22, 28]. The definitions of those languages are recalled in Table 3.4.

<table>
<thead>
<tr>
<th>Name</th>
<th>GT</th>
<th>NT</th>
<th>GCT</th>
<th>TCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFT</td>
<td>TREE</td>
<td>and ( \lor ) xor ( { \text{opt} } )</td>
<td>0</td>
<td>CR</td>
</tr>
<tr>
<td>OFD</td>
<td>DAG</td>
<td>and ( \lor ) xor ( { \text{opt} } )</td>
<td>0</td>
<td>CR</td>
</tr>
<tr>
<td>PFD</td>
<td>DAG</td>
<td>card ( \lor ) { \text{opt} }</td>
<td>{=, =}</td>
<td>CR</td>
</tr>
<tr>
<td>EPFD</td>
<td>DAG</td>
<td>card ( \lor ) { \text{opt} }</td>
<td>{=, =}</td>
<td>CR</td>
</tr>
<tr>
<td>GPFT</td>
<td>TREE</td>
<td>and ( \lor ) xor ( \lor ) \lor { \text{opt} }</td>
<td>0</td>
<td>CR</td>
</tr>
<tr>
<td>PFT</td>
<td>TREE</td>
<td>and ( \lor ) xor ( \lor ) \lor { \text{opt} }</td>
<td>{=, =}</td>
<td>0</td>
</tr>
<tr>
<td>vDFD</td>
<td>DAG</td>
<td>and ( \lor ) xor ( \lor ) \lor { \text{opt} }</td>
<td>0</td>
<td>CR</td>
</tr>
</tbody>
</table>

Table 3. FD variants defined on top of FFD

If we ignore the graphical and textual constraints, that is, the last two parameters in FFD, we can prove formally [26] that tree languages (OFT [16], GPFT [10], PFT [11]) are not fully expressive. However, DAG languages (ODF [17], EFD [23, 22], RFD [12]) are fully expressive. More precisely, our results show that the disjunction of features cannot be expressed in OFT. In [12], 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. In [26], we proved that the second extension alone guarantees full expressiveness while adding or-nodes only does not.

In vDFD, we have or-nodes but we do not have DAGs, or at least just a restricted form of DAG where only the sharing of leaf nodes is allowed. Studying the expressiveness of vDFD thus requires specific treatment.

The operators that manipulate the vDFD expressions must always have at least one operand. Therefore, vDFD expressions are expressively incomplete with respect to PL as the empty PL (i.e., the PL containing no product i.e. \{ \}) and the base PL (i.e., the PL containing one product in which no feature has been selected i.e. \{\}) can not be expressed in their disjunctive normal form. If we add the vDFD textual constraints, these two products can be expressed since preference is given to the constraints. An empty PL can be expressed by: a normal form one-of (\{a1\(A\}) and a constraint “exclude A”, where A is an atomic feature. A base PL can be expressed by: a normal form one-of (\{a1\(?\}) and a constraint “exclude A” where A is an atomic feature.

The consequence of this result is that we now know that vDFD equipped with constraints (at least exclude constraints) can be used to express any PL. This is interesting because vDFD are supported by a tool environment and so in theory all FD languages with PL semantics can also be supported by this environment, provided that forward and backward translations between vDFD and the other languages are implemented. We now discuss the practical feasibility of these translations with the two remaining criteria.

5.2.2 Embeddability

In [26], we have proposed a definition of graphical embeddability (Def. 5.4) which generalizes the definition of embeddability for context-free languages (here, simplified):

Definition 5.4 (Graphical embeddability) A graphical language \( \mathcal{L}_1 \) is embeddable into \( \mathcal{L}_2 \) if there is a translation \( T : \mathcal{L}_1 \to \mathcal{L}_2 \) that is node-controlled [15]: \( T \) is expressed as a set of rules of the form \( D_1 \to D_2 \), where \( D_1 \) is a diagram containing a defined node or edge \( n \), and all possible connections with this node or edge. Its translation \( D_2 \) is a subgraph in \( \mathcal{L}_2 \), plus how the existing relations should be connected to nodes of this new subgraph.

Given this definition, we only need to look at the (graph-based) abstract syntax of FD languages to study their embeddability.

All tree FD languages are clearly embeddable into vDFD. For DAG languages, this is not the case because we can have sharing of intermediate nodes in the graph. However, if we consider sublanguages that restrict the sharing to leaves, we just have to apply the linear transformation \( T \) (to guarantee edge-based semantics) and from Theorem 5.1 we can directly infer that we have an embedding. Finally, vDFD are embeddable into RFD.

Embeddings are of practical relevance because they ensure that there exists a transformation from one language to the other which preserves the whole shape of the models. This way, traceability between the two models is greatly facilitated and tool interoperability is made more transparent.
Embedding results must however be completed by examining the blow-up caused by the change of notation. This is what is measured by succinctness.

5.2.3 Succinctness

Succinctness (Def. 5.5) actually allows to compare the size of the diagram before and after translation.

Definition 5.5 (Succinctness) Let \( G \) be a set of functions from \( \mathbb{N} \rightarrow \mathbb{N} \). A language \( L_1 \) is \( G \)-as succinct as \( L_2 \), noted \( L_2 \leq G(L_1) \), iff there is a translation \( T : L_1 \rightarrow L_2 \) that is within \( G \): \( \exists g \in G, \forall n \in \mathbb{N}, \forall l_1 \in L_1, |l_1| \leq n \Rightarrow |T(l_1)| \leq g(n) \). Common values for \( G \) are “identically” = \( \{ n \} \), “thrice” = \( \{ 3n \} \), “linearly” = \( O(n) \), “cubically” = \( O(n^3) \), “exponentially” = \( O(2^n) \).

By definition, wherever there is an embedding, there also exists a linear translation. In our case, a vDFD produced from a tree-shaped FD is identically as succinct as the tree, and a vDFD produced from a restricted DAG is linearly as succinct as the latter (because intermediate nodes and edges need to be added). Also, the translation from a vDFD to an RFD is linear. In all those cases, the transformation engines do not face tractability issues. However, for turning unrestricted DAG into vDFD, we need to precompute all shared cases that vDFD will treat as independent. This will cause an exponential blow-up. In practice, this means that one will be able to apply such transformations only to small diagrams.

6 Related work

Besides vDFD, a few more formally defined FD languages exist and need to be compared with FFD (and also possibly between themselves) following the same methodology:

1. 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.

2. In [8, 7], 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 [7], the authors provide an even richer syntax. The semantics of the latter is yet to be defined, but is intended to be similar to [8].


4. Wang et al. [29] propose a semantics of FD using ontologies. A semantic web environment is used to model and verify FD with OWL DL. The RACER reasoner is used to check inconsistencies during feature configuration. Their semantics slightly differ from ours, since (1) they omit justifications and (2) they did not eliminate auxiliary symbols. Missing justifications yield strongly counter-intuitive results, but keeping auxiliary symbols is harmless for consistency checking (but incorrect for other tasks) as shown in [4].

As mentioned in Sec. 1, we think that these approaches do not rank as good as ours on generality, abstraction and intuitiveness. However, a finer comparison is required in order to further justify these claims just as we have done in this paper for vDFD. This a topic for future work.

7 Conclusion

This work is a first step towards the comparative semantics of feature diagrams. We have recalled FFD, a generic formalization of feature diagrams. We have also recalled and revisited van Deursen and Klint’s definition of FD, which we called vDFD. We have then compared the two by applying the principles of comparative semantics. We have defined an abstraction function that relates van Deursen and Klint’s semantics to our own and then studied 3 properties of vDFD: expressiveness, embeddability and succinctness.

We can summarize the conclusions and practical implications of our investigations as follows:

- By being able to abstract vDFD’s semantics to ours and assuming that the abstracted information (the order of features) is irrelevant, we further validated the appropriateness of our semantic domain and semantic function for representing product lines. This gives us more assurance that FFD can be used as a reference semantics for implementing future tools for engineering the requirements of software product lines.
• We have discovered a few minor errors in the original formalization of vDFD which can cause an automatic reasoner based on it to yield erroneous results. These findings can help the developers to improve their tool and future developers to avoid the same problems.

• More fundamentally, we analyse the lack of abstraction and the presence of errors in the original formalization as a result of the bias imposed by the tool. This tends to confirm the advantages of our methodology [26, 25] where we recommend tool-independent formalization prior to the adoption or development of any supporting tool. The generic semantic framework of FFD greatly facilitates such a definition (which can boil down to filling a simple 4-entry template as shown in Table 3). From there on, the available results of comparative analyses can guide the adopters in the selection or development of their supporting tools.

• We have proved that vDFD are expressively complete with respect to the semantic domain of product lines, just as some other members of the FFD family. From this, we have concluded that the vDFD language and its supporting tools can be used to model, and reason on, product lines without the fear of being limited in expressiveness. However, this conclusion has to be mitigated by the preceding conclusions concerning the lack of abstraction and presence of errors in the original tool-driven formalization.

• By looking at translations, we have assessed the feasibility of transforming vDFD models into other kinds of feature models and conversely. Based on these results, developers can start writing model transformation scripts to enable tool interoperability. For example, if some reasoning facilities are not supported by tool A but they are supported by tool B, embeddability of A's notation into B's guarantees that a transformation from A to B is not only possible but also that B preserves the shape of the original model so that traceability between the two models is greatly facilitated. Succinctness measures the cost of the translations. Regarding translations that have vDFD as a target, we have observed that, roughly stated, the translation from tree-shaped FD are easily tractable whereas those from DAG-shaped FD are only tractable for small diagrams. Finally, vDFD have been found embeddable into RFD.

Comparative semantics should be obtained for all variants of feature diagrams, so that we know precisely what they are, what are their respective qualities: expressiveness, succinctness, embeddability, but also axiomatizability, complexity of the reasoning tasks, etc. These objective data can then serve to guide the selection of a common standard for feature diagrams. Such a standard could hopefully accelerate their adoption by industry, increase the competition between tool providers, and help to establish efficient and safe requirements and software product line engineering methods.

8 Threats to validity

The main threats to the validity of our method to compare FD languages is that it considers only the formal aspects of these languages. Hence, we are faced with the usual advantages and drawbacks of any formalization endeavour [13]. On the one hand, a mathematically defined semantics tends (i) to eliminate ambiguity in the interpretation of languages, (ii) to facilitate safe and efficient tool automation and (iii) to allow the definition of objective criteria to evaluate and compare languages. On the other hand, a formal semantics can never guarantee by itself that it captures enough and the right information about the domain being modeled.

More general language evaluation frameworks exist that take into account a wider spectrum of qualities and criteria. For example, Krogstie [18] proposes a comprehensive language and model 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. These are topics for future work which need to be addressed by empirical validation when we will have developed a concrete (user-oriented) syntax and supporting tools based on FFD. Then, we will be able to evaluate whether there is a good match between the intended (“real world”) and the formal semantic domains and functions of our language. Yet, it should be noted that even if we obtain a good assurance that there is such a match, we will still not be able to guarantee that our language will always be used to represent the relevant information about the requirements of the system at hand for no formal, nor informal, language can ever guarantee this by itself [14].

Another threat to the validity of our results is that all the formal definitions and reasoning in this paper have been carried out by humans without the assistance of tools (except text editing tools). Therefore, we cannot guarantee that human errors are completely absent from our comparative analysis.

Finally, it should be noted that we have only assessed the language defined by van Deursen and Klint from the description that is made of it in [27]. We have not assessed the tool itself, which might well have been improved to address the issues that we mention.
References


