

Generic semantics of feature diagrams [☆]

Pierre-Yves Schobbens ^a, Patrick Heymans ^a, Jean-Christophe Trigaux ^{a,*},
Yves Bontemps ^b

^a *Computer Science Department, University of Namur, 5000 Namur, Belgium*

^b *SMALS MvM/leGOV, 1000 Brussels, Belgium*

Available online 20 September 2006

Responsible Editor: H. Rudin

Abstract

Feature Diagrams (FDs) are a family of popular modelling languages used to address the feature interaction problem, particularly in software product lines. FDs were first introduced by Kang as part of the FODA (Feature-Oriented Domain Analysis) method back in 1990. Afterwards, various extensions of FODA FDs were introduced to compensate for a purported ambiguity and lack of precision and expressiveness. However, they never received a formal semantics, which is the hallmark of precision and unambiguity and a prerequisite for efficient and safe tool automation.

The reported work is intended to contribute a more rigorous approach to the definition, understanding, evaluation, selection and implementation of FD languages. First, we provide a survey of FD variants. Then, we give them a formal semantics, thanks to a generic construction that we call Free Feature Diagrams (FFDs). This demonstrates that FDs can be precise and unambiguous. This also defines their expressiveness. Many variants are expressively complete, and thus the endless quest for extensions actually cannot be justified by expressiveness. A finer notion is thus needed to compare these expressively complete languages. Two solutions are well-established: succinctness and embeddability, that express the naturalness of a language. We show that the expressively complete FDs fall into two succinctness classes, of which we of course recommend the most succinct. Among the succinct expressively complete languages, we suggest a new, simple one that is not harmfully redundant: Varied FD (VFD). Finally, we study the execution time that tools will need to solve useful problems in these languages.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Feature diagram; Survey; Formal semantics; Feature interaction; Software product lines

1. Introduction

Features have been defined as “a distinguishable characteristic of a concept (e.g., system, component and so on) that is relevant to some stakeholder of the concept” [9]. This broad definition encompasses most commonly used meanings, for example: new services or options added to an existing base system.

[☆] Work supported by Walloon Region and FSE under FIRST Europe Objective 3 Project PLENTY EPH3310300R0462/215315.

* Corresponding author. Tel.: +32 81 72 49 85; fax: +32 81 72 49 67.

E-mail addresses: pys@info.fundp.ac.be (P.-Y. Schobbens), phe@info.fundp.ac.be (P. Heymans), jtr@info.fundp.ac.be (J.-C. Trigaux), Yves.Bontemps@smals-mvm.be (Y. Bontemps).

The description of systems in terms of features is popular in many domains such as telecommunication or embedded systems. Features are not used for one-shot developments, but for systems that must have multiple incarnations because (1) they need to evolve through time or (2) variants must be deployed in many contexts. A typical example of the former are telephony systems which, to face harsh competition, need to adapt quickly to satisfy demand. An example of the second are *product lines* (also known as product families) in domains such as home appliances or the automotive industry.

Two perspectives are used in turn:

- When the base system (or system family) is conceived, features are “units of evolution” (or change) that adapt it to an optional user requirement [12]. A recurrent problem is the one of *feature interaction*: adding new features may modify the operation of already implemented ones. When this modification is undesirable, it is called a *feature interference*. This problem is difficult because “individual features do not typically trace directly to an individual component” [15]. With the increasing size and complexity of current systems and associated product lines, dealing with feature interaction becomes challenging. To guarantee that systems will deliver the desirable behaviour, interactions must be detected, resolved if possible, or the combination of features must be forbidden.
- When a specific system (or family member or product) must be designed, “[the] product is defined by selecting a group of features, [for which] a carefully coordinated and complicated mixture of parts of different components are involved” [15]. Hence, it is essential that features and their interactions be well-identified. This knowledge about their valid and forbidden combinations must be packaged in a form that is usable by the customers selecting their options, and by the engineers deploying them.

In software product lines, the former activity is often referred to as *domain engineering* and the second as *application engineering* [5,26].

To express the knowledge about allowed feature combinations, practitioners use feature diagrams (FDs). FDs are an essential means of communication between domain and application engineers as well as customers and other stakeholders such as marketing representatives, managers, etc. In partic-

ular, FDs play an essential role during requirements engineering [26], which is known to be the most critical phase of software development. FDs provide a concise and explicit way to:

- describe allowed variabilities between products of the same line/family,
- represent feature dependencies,
- guide the selection of features allowing the construction of a specific product,
- facilitate the reuse and the evolution of software components implementing these features.

These benefits of FD notations are deemed particularly valuable in emergent application domains which, above all, are characterized by an increasing complexity: they undergo a multitude of constantly changing requirements and must adapt quickly to changing technologies, they are pervasive (they have to interact with many kinds of users and systems), they are composed of many communicating, heterogeneous and often legacy components. The fact that such systems exist in many, possibly co-existing variants brings an additional dimension to this complexity. Therefore, variability management techniques have been explored and used in domains such as smart homes [26], electronic commerce and embedded systems [13] or robotics [21]. Such techniques are central in the emergent product line engineering paradigm. Features have also been advocated to be first-class citizens in Model-Driven Engineering (MDE) [30] approaches such as Fire Works [29].

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 [19]. 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 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.

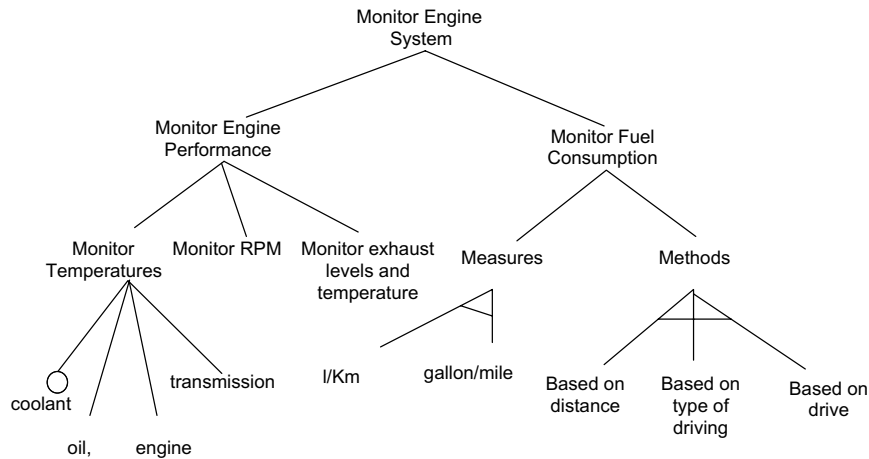


Fig. 1. FODA (OFT): monitor engine system.

In the sequel, we will refer to FODA FD as Original Feature Trees or OFTs for short. “Original” is because OFTs were the first FD language ever proposed and because they were the starting point for the work of many researchers. “Tree” is because in OFT, FDs are structured as trees vs. single-rooted directed acyclic graphs (DAGs) as in FORM [20].

Since Kang et al.’s initial proposal, several extensions of OFT have been devised as part of the following methods: FORM [20], FeatureRSEB [14], Generative Programming [9], PLUSS [10], and in the work of the following authors: Riebisch et al. [28,27], van Gurp et al. [36], van Deursen et al. [35], Czarnecki et al. [7,8] and Batory [1]. While some authors have recently started to better define their semantics [35,3,8,1], most proponents of FD [20,14,9,28,27,36,10] have not. Still, they have argued for an “improved expressiveness”. In this paper, we adopt a formal approach to check the validity of such claims.

Formal semantics is not an issue to be taken lightly. As remarkably argued in [16,17], formal semantics is the best way (1) to avoid ambiguities (see Definition 22) and (2) to start building safe automated reasoning tools for a variety of purposes including verification, transformation and code generation. More specifically, for FD, we must be sure that a model excludes all the forbidden feature combinations and admits all the valid ones. If this is not the case, harmful feature interactions are likely to take place, or the product line will be unnecessarily restricted and thus less competitive. A tool for assisting stakeholders in selecting features therefore must be based on formal semantics.

Our formal approach is designed to introduce more rigour in the motivation and definition of FD languages. This should make the discussion of their qualities more focused and productive. In the end, we hope for a better convergence of research efforts in this area. A new, rigorously defined and motivated FD language, called VFD, is introduced in this paper as a first step in this direction. VFD uses a single construct instead of all the proposed extensions. Indeed, the proliferation of constructs and languages is an additional source of interpretation and interoperability problems. For example, if two product lines that use 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. Moreover, a new FD will have to be created from the existing ones to account for the new merged product line. It is important that rigorous criteria are used to choose the new FD language and that proved correct semantic-preserving translations are used.

In the context of emergent and complex systems, it is important that automation of product line and model-driven engineering activities such as those that we just described are made as available, as safe and as efficient as possible. The present paper tries to advance the state of the art in feature modelling towards this target. 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 and possibly others to come. Sections 3.1 and 3.2 describe our proposal of a common abstract syntax. The semantics then given in Section 3.4. In Section 3.3, we apply this framework to give a formal semantics to OFT and several of its extensions. Surprisingly, although extremely concise and translated almost literally from [19], a formal definition of OFT had never been published before.¹ Since we restricted the scope of this paper to FD notations that lack a formal semantics, formal definitions for the analysed extensions are also missing. The formal comparison of FD languages is addressed in Section 4. Contrary to folklore, we prove that all these extensions add no expressiveness to the first extension FORM (OFD) [20], since OFDs are expressively complete (Theorem 4.3). We do not question the fact that extended FDs have advantages over the original ones, but these are *succinctness* (Section 4.3) and *naturalness* (Section 4.2) rather than *expressiveness*. In most extensions, there are synonymous constructs, i.e., they are *redundant* (Definition 23). Thus we propose a new, natural, succinct, and non-redundant variant of FDs: VFD. We conclude Section 4 with a study of the complexity of the decision procedures supporting an envisioned FD-based tool environment (Section 4.5). Finally, Section 5 compares our results with related works, especially other formal definition endeavours [35,8,1], while Section 6 concludes the paper.

2. Survey

In this section, we survey the informal OFT extensions enumerated in Section 1. We illustrate them with examples from the same case study (the one already used in Section 1) adapted from [6] and concerning an engine system monitor product line.

For convenience, we have developed a naming scheme to designate the reviewed languages. FD languages without graphical constraints 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 one of its authors. Then comes an ‘F’ (for feature). The last letter is either ‘D’ (for DAG) or ‘T’ (for trees). The names “OFD” and “OFT” introduced in the previous section already follow this scheme.

2.1. FODA (OFT)

OFT, the first ever FD, was introduced as part of the Feature-Oriented Domain Analysis (FODA) method [19]. Its main purpose was to capture commonalities and variabilities at the requirement level. As depicted in Fig. 1, OFT is 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 `gallon/mile`, 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 indicates 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) [20] 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

¹ Except of course in abstracts of this paper [3,4].

engineering but also for software design. In terms of FD, several changes occurred:

1. FDs are not necessarily trees but can be DAGs.
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” (OFDs).

In this paper, we only consider the constructs which have an influence on the feature combinations allowed in the final products (see Definition 13). Therefore, only the first change (trees become DAGs) 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 [14] 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 FDs have the following characteristics:

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

We call such diagrams “RSEB Feature Diagrams” (RFDs). An example is given in Fig. 2.

2.4. van Gorp et al. (VBFD)

van Gorp, Bosch and Svahnberg define another FD language in [36]. 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 FDs 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 Gorp and Bosch Feature Diagrams” (VBFDs). These changes mainly concern concrete syntax, and certainly have no influence on feature combinations. Thus we also call their abstract syntax RFD.

2.5. Generative programming (GPFT)

Czarnecki and Eisenecker [9] have studied and adapted FDs in the context of Generative Programming (GP), a new programming paradigm that aims to automate the software development process for product families. Their FDs are simply OFTs with the addition of *or* decomposition and graphical constraints.

We call such diagrams “Generative Programming Feature Trees” (GPFTs).

Recently, the authors have further augmented their approach with concepts such as *staged configuration* [7], distinguishing between *group* and *feature cardinalities* (see Section 2.6) and formalizing their language [8]. 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 [28]. Moreover, they argue that “combinations of mandatory and optional features with alternatives, or and xor relations could lead to ambiguities” [27, p. 4]. As we will see, those “ambiguities” are due to a different conception of what “mandatory” and “optional” mean (see Example 3.5), and are better termed redundancies (see Section 4.4).

In order to limit these drawbacks, the authors replace *or*, *xor* and *and* by UML-like *multiplicities*

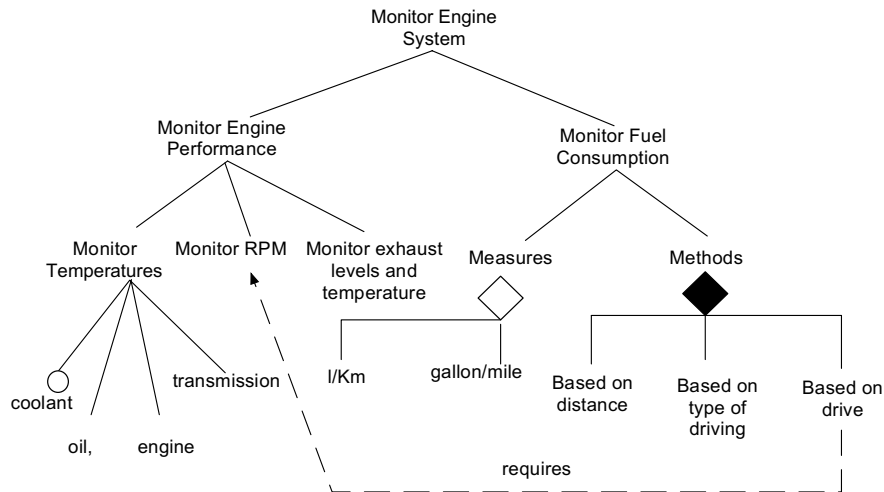


Fig. 2. FeatuRSEB (RFD) monitor engine system.

(a.k.a group cardinalities [8]) and *mandatory* and *optional edges* [28]. 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.

In Fig. 3, we can also observe mandatory and optional edges which are marked with a black or a hollow circle, respectively, at the lower end. This is rather original as the other FD languages prefer to mark the optionality on the node. Optional edges

are useful in DAG (and [28] indeed use DAG) since a feature can be optional on one side and mandatory on another as Fig. 4 illustrates.

We call these diagrams “Extended Feature Diagrams” (EFDs).

2.7. PLUSS (PFT)

The Product Line Use case modelling for System and Software engineering (PLUS) approach [10] is based on FeatuRSEB. It combines FDs and use case diagrams to depict the high level view of a product family. FDs are used to instantiate the abstract use

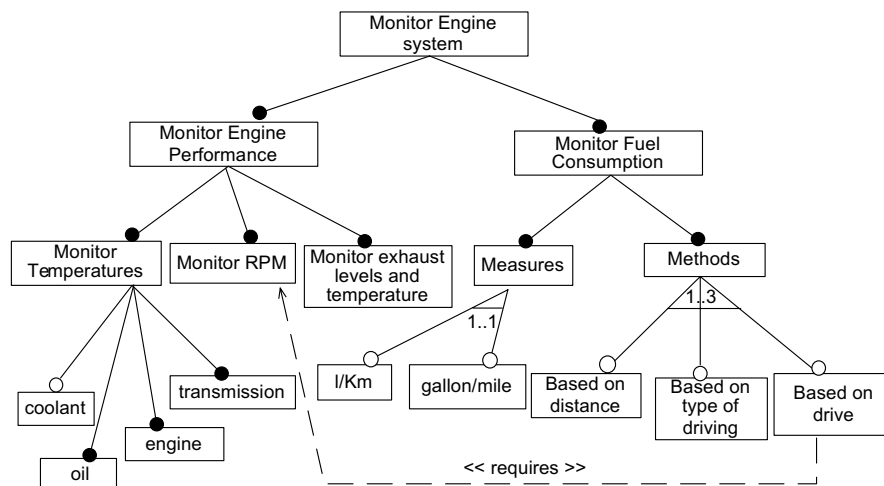


Fig. 3. EFD: monitor engine system.

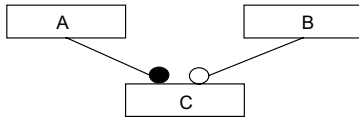


Fig. 4. EFD: optional and mandatory edges [28].

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 parent while *multiple adaptors* (nodes with a circled ‘M’) represent *or*-decomposition.
2. PLUSS FDs are trees.
3. They have graphical but no mention of textual constraints.
4. Mandatory nodes are filled in black while optional ones are hollow.

We call these diagrams “PLUSS Feature Trees” (PFTs). An example is given in Fig. 5.

A more formal definition of all these FD languages will be given in the next section. In particular, Table 1 will provide both a definition and a summary of the various notations.

Table 1

Formal definition of FD variants on top of FFD

Short name	GT	NT	GCT	TCL
OFT	1	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
OFD	0	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
RFD	0	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
EFD	0	$card \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
GPFT	1	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	CR
PFT	1	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	$\emptyset$
CFD(OP)	0	<i>OP</i>	$\emptyset$	$\emptyset$
COFD	0	$and \cup xor$	$\emptyset$	$\emptyset$
VFD	0	<i>card</i>	$\emptyset$	$\emptyset$
CFT(OP)	1	<i>OP</i>	$\emptyset$	$\emptyset$
COFT	1	$and \cup xor$	$\emptyset$	$\emptyset$
CRFT	1	$and \cup xor \cup or$	$\emptyset$	$\emptyset$

3. Generic formal definition of feature diagrams

In this section, we provide a generic definition of the syntax and semantics of FD. We do this by introducing *Free Feature Diagrams* (FFDs). FFDs are a generalization of all the variants of FDs surveyed in Section 2. To make everything explicit, we separate concrete syntax, abstract syntax and semantics.

3.1. From concrete to abstract syntax

The literature surveyed in the previous section does not clearly separate *concrete syntax* (what the user sees) from *abstract syntax*. With respect to

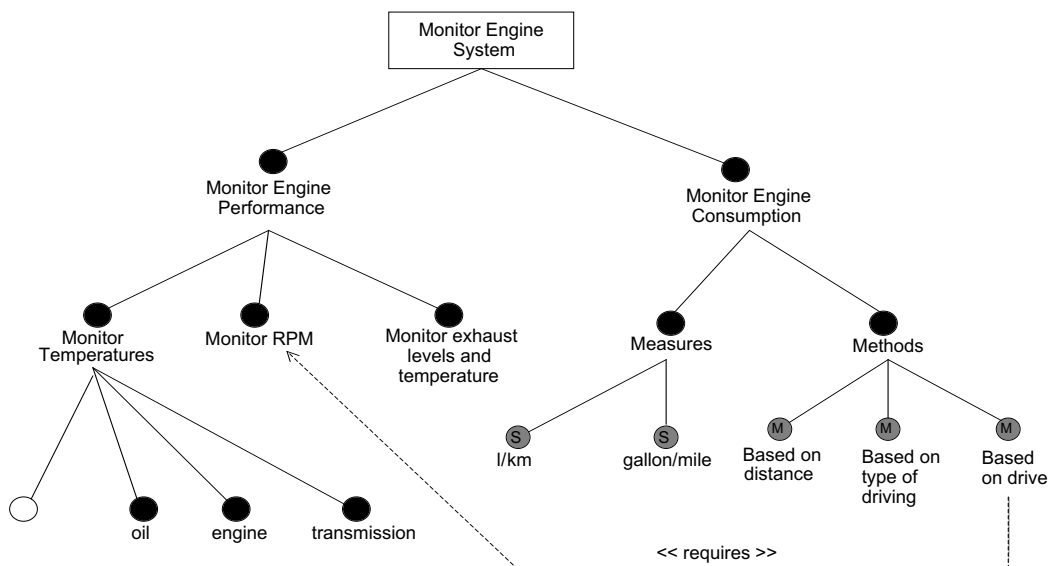


Fig. 5. PFT: monitor engine system.

the concrete syntax, the abstract syntax ignores the visual rendering information that is useless to assign a formal semantics to a diagram, e.g., whether nodes are circles or boxes, whether an operator is represented by a diamond shape or by joining the edges departing from a node, etc. From a tool developer's point of view, the abstract syntax is also an abstract view of the data structures stored in the computer which ignores irrelevant implementation details such as pointer values, etc.

Distinguishing between concrete and abstract syntax allows identifying similarities and differences between languages that might otherwise remain unnoticed. For example, as we have seen in Section 2.4, despite some visual differences, VBFD and RFD have the same abstract syntax. Instead, the approach taken in most of the surveyed languages is to discuss semantics informally, on the basis of visual representations, assuming that the reader's intuition will do the rest. But actually, each reader's intuition is potentially different. Hence, following the recommendation of [16,17], we will define semantics on the basis of the abstract syntax, which is much simpler and less error-prone.

The first step in this chain is to have a bidirectional conversion from the concrete syntax to the abstract syntax. There are two common ways to provide the latter information [16,17]: (1) *mathematical notation* (set theory) or (2) *meta-model* (usually, a UML Class Diagram complemented with OCL constraints). In this paper, we follow [16,17] and adopt the former for its improved conciseness, rigour and suitability for performing mathematical proofs.²

From our survey, it turns out that all concrete FDs are graphs whose nodes are features. Features are 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 for “node”.

We further distinguish “primitive” and “compound features”.^{3,4} *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. Note that a primitive features p bearing the operator op_s , $s > 0$ being the arity of the operator, can always become a leaf:

1. we add two new non-primitive nodes, n_1 bearing and_2 and n_2 bearing op_s ;
2. the sons of n_1 are n_2 and p ;
3. the sons of n_2 are the former sons of p .

What we call edges is an abstraction for the arrows and line segments we find in concrete FDs. 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 an 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 [20] seems to hint that it should be abstracted to (b), while EFD [28] to (c). Because we want to account for both, we have to take the finer decomposition (d).

Similar issues had to be solved to deal with filled circles⁵ (mandatory features) and multiple operators attached to the same parent feature as in PFT. They were also addressed by the addition of implicit nodes and edges.

3.2. Generic abstract syntax

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

FD languages vary in several respects: (1) Do they consider FDs as trees or DAGs? (2) What are

² However, a meta-model might be more readable and facilitate some implementation tasks such as building a repository, but these issues are not addressed in this paper.

³ We adopt the terminology of [1].

⁴ “Compound features” are also called “decomposable features”.

⁵ We see them as *and*₁ nodes, see below.

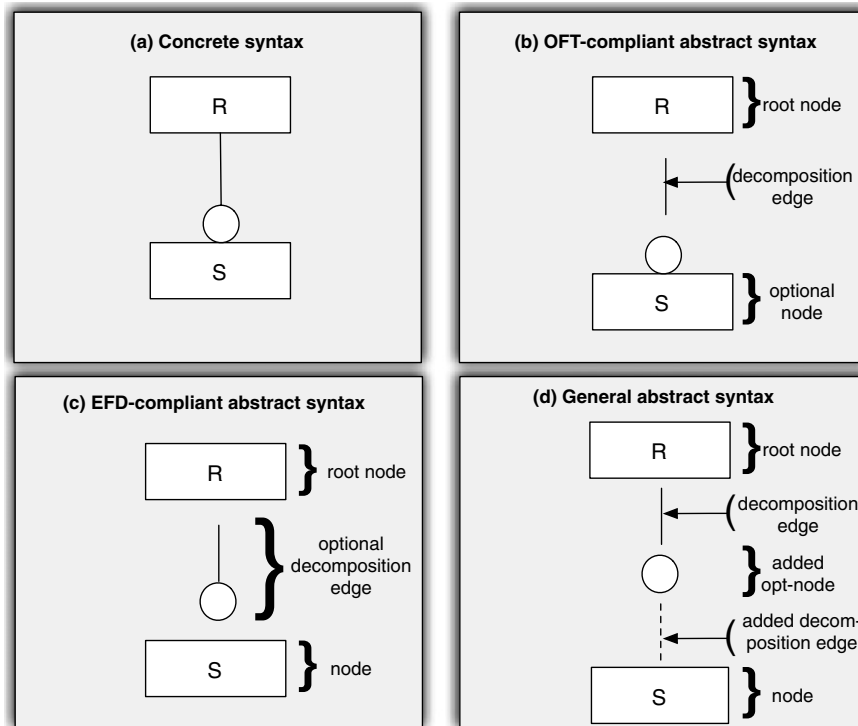


Fig. 6. Three of the possible abstract syntaxes.

the allowed types of operators on nodes? (3) what are the allowed types of graphical constraints? (4) What are the textual constraints?. These are the parameters (GT, NT, GCT, TCL) of FFD. Each FD language can be defined simply by providing values for these parameters. Semantics is then given only once, for FFD. Hence, the formal semantics of a particular FD language defined through FFD comes for free.

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

Before defining FFD, we define the structure and possible contents of their parameters.

Definition 1 (Graph Type). A graph type (GT) is a Boolean value 0 stands for DAG and 1 stands for tree.

We need to extend the well-known notion of (binary) commutativity to an arbitrary number of arguments:

Definition 2 (Commutative). A k -ary function is commutative if the result does not depend on the order of its arguments.

Definition 3 (Operator). An operator is a commutative Boolean function.

An operator can only count the number of true (or equivalently false) arguments to be commutative. Thus we can identify k -ary operators with constraints on this number. There are thus 2^{k+1} k -ary operators, closed under negation. For instance, the eight binary operators are opt_2 , and_2 , xor_2 , or_2 (see their definition just below) and their negation.

Definition 4 (Node Type). A node type (NT) is a set of operators, at most one per arity.

Example. and is the set of operators and_k , one for each arity k , that return true iff all their k arguments are true. For example, the node Monitor Engine Performance in Fig. 1 bears the operator and_j because it has three sons.

Example. xor is the set of operators xor_k that return true iff exactly one of their k arguments is true. For

example, the node `Methods` in Fig. 1 bears operator xor_3 .

Example. or is the set of operators or_k that return true iff any of their k arguments is true. For example, the node `Methods` in Fig. 2 and the node `A` in Fig. 15(1) and (2) all bear the operator or_3 .

Example. $vp(i \cdot \cdot j)$ is the set of operators of the form $vp(i \cdot \cdot j)_k$ where $i \in \mathbb{N}$ and $j \in \mathbb{N} \cup \{*\}$. $vp(i \cdot \cdot j)_k$ returns true iff at least i and at most j of its k arguments are true. The union of $vp(i \cdot \cdot j)$ is called *card*. $i \cdot \cdot j$ is called *group cardinality*⁶ as in [7]. For example, the nodes `Measures` and `Methods` in Fig. 3 bear operators $vp(1 \cdot \cdot 1)_3$ and $vp(1 \cdot \cdot 3)_3$, respectively. A vp is *reduced* if j is at most k . It is *consistent* if furthermore $i \leq j$. From this definition, it appears that some forms of vp operators are equivalent to more classical Boolean operators. For example, xor_s and $vp(1 \cdot \cdot 1)_s$ are the same. We will use these properties, most notably in Section 4.2 (see Table 2).

Example. opt is the set of operators opt_k , that always return true, irrespective of the truth value of their arguments. For example, the hollow circles above `coolant` in Figs. 1–3 and 5 are opt -nodes. They bear opt_1 .

The types of operators that we have just listed are sufficient to embed the surveyed FD languages into FFD. That is, for a given FD language, NT is usually a subset of $or \cup and \cup xor \cup card \cup opt$ but this list is by no means exhaustive. One could well create a new form of FD whose nodes bear other kinds of operators.

Similarly, we now open the set of possible constraint types that FD languages can use. Graphical constraints are represented by an edge joining two nodes in the FD.

Definition 5 (*Graphical Constraint Type*). A graphical constraint type (GCT) is a binary Boolean function.

Example. *Requires*, a.k.a. $\Rightarrow$: when a (directed) constraint edge bears this function, its source node must be out or its target node must be in.

⁶ The terms *cardinality* and *multiplicity* are also commonly used.

Table 2
Embedding RFD into VFD

Instead of...	write...
An option node op	a $vp(0 \cdot \cdot 1)$
A xor -node	a $vp(1 \cdot \cdot 1)$
An or -node	a $vp(1 \cdot \cdot *)$
An and -node with number of sons s	a $vp(s \cdot \cdot s)$

Example. *Mutex*, a.k.a. Sheffer's bar [33] or *nand*₂: when a constraint edge bears this function, if one of its nodes is in, then the other must be out. For operators, no need to orient the constraint link in the concrete syntax.

In the surveyed languages, GCT is either $\{\Rightarrow, |\}$ or $\emptyset$ but, again, this list is not exhaustive.

Definition 6 (*Textual Constraint Language*). A textual constraint language (TCL) 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” [19, p. 71], is:

$CR ::= p_1(\text{requires} | \text{mutex})p_2$,

where p_1 and p_2 are primitive features.

Definition 7 (*Free Feature Diagram*). A free FD (FFD) $d \in FFD(GT, NT, GCT, TCL)$, $d = (N, P, r, \lambda, DE, CE, \Phi)$ 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 (usually, r is drawn at the top, e.g., Monitor Engine System in Fig. 1);
- $\lambda: N \rightarrow NT$ 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 $n \rightarrow n'$; DE is acyclic and, if $GT = 1$, it is also a tree (see Definition 10 below);
- $CE \subseteq N \times GCT \times N$ is the set of constraint edges;
- $\Phi \subseteq TCL$ are the textual constraints.

In order to facilitate the formal developments to come, we introduce the following two definitions:

Definition 8 (*Son, parent*). If $n_1 \rightarrow n_2$, we call n_1 a parent of n_2 and n_2 a son of n_1 .

Definition 9 (*Leaf*). A node $n \in N$ that has no son is called a leaf. We will call L the set of leaves.

It is worth noting that only DE is considered to define the notions of son, parent and leaf, FFD collects whatever can be drawn. FDs have additional minimal well-formedness conditions.

Definition 10 (*Feature Diagram*). A feature diagram (FD) is a FFD where:

1. Only r has no parent: $\forall n \in N. (\nexists n' \in N. n' \rightarrow n) \iff n = r$.
2. DE is acyclic: $\nexists n_1, \dots, n_k \in N. n_1 \rightarrow \dots \rightarrow n_k \rightarrow n_1$.
3. If $GT = 1$, DE is a tree: $\nexists n_1, n_2, n_3 \in N. n_1 \rightarrow n_2$ and $n_3 \rightarrow n_2$.
4. Nodes are labelled with operators of the appropriate arity: $\forall n \in N. \lambda(n) = op_k$, an k -ary operator where $k = \#\{(n, n') | (n, n') \in DE\}$.

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.

Beside the languages surveyed in Section 2, Table 1 also includes the definition of four more Spartan variants (see the six last entries in Table 1):

- Constraintless Optionless Original Feature Diagrams (COFDs),
- Varied Feature Diagrams (VFDs),
- Constraintless Optionless Original Feature Trees (COFTs),
- Constraintless Optionless Feature Trees with *or*-nodes (CRFTs).

COFD and VFD are defined by instantiating OP in the parametric definition of Constraintless Optionless Feature Diagrams (CFD(OP)) defined just above in the table. COFT and CRFT are defined similarly through CFT(OP). Motivations for these new languages will be given in Section 4.

Theorem 3.1. *We observe the following syntactical inclusions:*

1. $COFD \subset OFD \subset RFD$
2. $VFD \subset EFD$
3. $COFT \subset OFT \subset OFD$
4. $COFT \subset OFT \subset GPFT$

5. $COFT \subset CRFT \subset PFT$
6. $CRFT \subset GPFT$

3.4. Semantics

The semantics of an FD is defined as a *Product Line* (PL) (see Definition 13), i.e., a set of products. A product provides a set of primitive features (see Definition 13).

The notion of model is introduced in [19, p. 64], with the examples of models of X10 terminals.

Definition 11 (*Model*). A model of an FD is a subset of its nodes: $M = \mathcal{P}N$.

Definition 12 (*Valid model*). [19, 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 $s_1, \dots, s_k$ $\lambda(n) = op_k$, then $op_k(s_1 \in m, \dots, s_k \in m)$ must evaluate to true.
3. The model must satisfy all textual constraints: $\forall \phi \in \Phi, m \models \phi$, where $m \models \phi$ means that we replace each node name n in ϕ by the value of $n \in m$, evaluate ϕ and get 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: $\forall s \in N. s \in m \wedge s \neq r: \exists n \in N: n \in m \wedge n \rightarrow s$.

Example. According to our semantics, the two FDs in Fig. 7 are not equivalent, if we take all features as primitive. The left one (in OFT syntax) has models and products $\{r, x, y, l, n\}$ and $\{r, x, y, m\}$ whereas the right one 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 [28].

Definition 13 (*Product, Product Line*). 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 $\llbracket m \rrbracket$, 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 an FD d consists of the products named by its valid models: $\llbracket d \rrbracket = \{m \cap P | m \models d\}$.

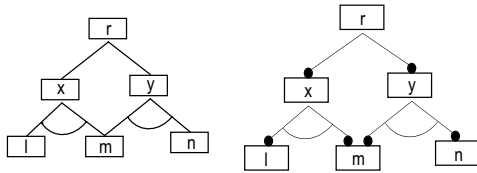


Fig. 7. Node- vs. edge-based semantics.

3.5. Discussion

Defining a formal semantics as we just did gives the opportunity to uncover some important issues that might well remain unnoticed with only an informal semantics. Here are a few examples:

Example. In our semantics, Fig. 3 admits a model with neither 1/km nor miles/gallon. This is clearly not the semantics intended in [28]. We simply suggest removing the harmful hollow circles.

Example. Our semantics also contradicts the sentence: “All mandatory features are part of all [models]” [34, 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 parents is. However, in the same papers, the authors seem to agree with our interpretation. Indeed, in Fig. 8, extracted from ([34], Fig. 3), Email is mandatory, but they make clear that it is not included in models where Net is not present.

If such issues are not made explicit and solved before a language is made public (respectively building tools), interpretations of the user (respectively 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 used 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, perhaps, are not the intended ones. But at least, we have exposed them precisely, concisely and without ambiguity. Thus now, they can be discussed constructively by their proponents and their respective user communities. Some starting points for these discussions can be found in Section 5.

Another advantage of formal semantics is that it makes it possible to compare languages according to formally defined criteria instead of loosely defined ones. This is the topic of the next section.

4. Formal comparison

In this section, we compare FD languages with respect to the formal criteria of expressiveness, embeddability, succinctness and redundancy.

4.1. Expressiveness

Habitually, the *expressiveness* of a language is the part of its semantic domain that it can express.

Definition 14 (*Expressiveness*). The expressiveness of a language $\mathcal{L}$ is the set $E(\mathcal{L}) = \{\llbracket D \rrbracket \mid D \in \mathcal{L}\}$, also noted $\llbracket \mathcal{L} \rrbracket$. 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 $\llbracket \cdot \rrbracket : \mathcal{L} \rightarrow \mathbb{M}$) is expressively complete 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 $\mathcal{L}_2$:

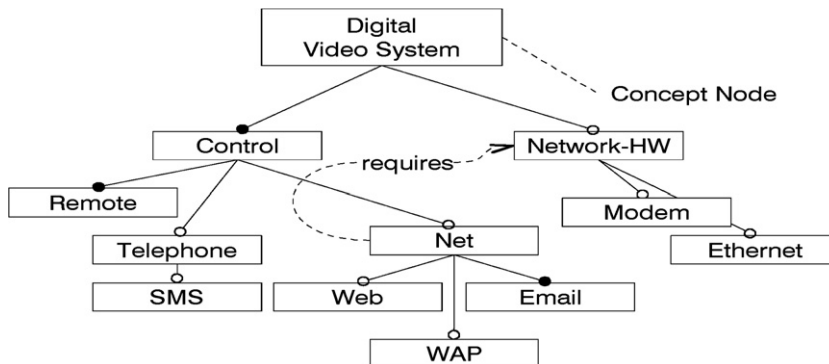


Fig. 8. RFD example from ([34], Fig. 3).

Definition 15 (Translation). A translation is a total function $T: \mathcal{L}_1 \rightarrow \mathcal{L}_2$ that is correct, i.e., preserves semantics: $\llbracket T(D_1) \rrbracket = \llbracket D_1 \rrbracket$.

The semantic domain of FD is product lines. In other words: every FD expresses a PL. Now, we can ask the converse question: Can every PL be expressed by an FD of a given kind? In other words: is this language expressively complete? We will see that this is usually false for tree languages, and true when sharing is allowed. This important point has often been overlooked.

4.1.1. Tree variants

The main FD variants [19,9] as well as several others [10,1,35,7] use trees only, instead of DAGs with a single root as allowed by our definition. We called them “feature trees”.

Trees bring several simplifications to our semantics:

1. A textual variant of the notation is now easily obtained by using this tree as an abstract syntax tree, see, e.g., [35]. Here, we use the notation $op(s_1, \dots, s_n)$ where op is the Boolean operator and s_i is the text representing the i th subtree. Optional nodes will be noted as $\dot{s}$. These notations should however not be confused with their logical equivalents: for instance $and_2(f_1, f_2)$ (where f_1 and f_2 are primitive features) designates a feature tree whose semantics is $\{\{f_1, f_2\}\}$, while $f_1 \wedge f_2$ is a propositional formula whose semantics in presence of another primitive feature, say f_3 , is $\{\{f_1, f_2\}, \{f_1, f_2, f_3\}\}$. More generally, formulae do not care about the presence of non-mentioned primitive features, while trees (and graphs) exclude them.
2. The justification rule (Definition 12, last point) simplifies since each node has exactly one parent.
3. We know that each subtree mentions only primitive features that do not occur in its brothers, since no sharing is allowed. We call this the signature of the subtree.
4. The difference between node-based and edge-based semantics then vanishes.
5. The semantics of feature trees can now be expressed compositionally: $\llbracket n \rrbracket$, where n is a node of a diagram, is the set of products that would be obtained by choosing n as the root of the diagram. Let op_k be the operator of n , and $s_1, \dots, s_k$ its sons.

We observe that $\llbracket op_k(s_1, \dots, s_k) \rrbracket = \{\bigcup c_i \mid \exists v_1, \dots, v_k \in \mathbb{B} \cdot op_k(v_1, \dots, v_k) = 1 \wedge \forall i \in 1, \dots, k, (v_i = 0 \Rightarrow c_i = \emptyset) \wedge (v_i = 1 \Rightarrow c_i \llbracket s_i \rrbracket)\}$.

- (a) For *and*-nodes, this simplifies to $\{\bigcup_i c_i \mid c_j \in \llbracket s_i \rrbracket\}$.
- (b) For *xor*-nodes, this simplifies to $\bigcup_i \llbracket s_i \rrbracket$.

One readily observes that:

Theorem 4.1. *OFT cannot express disjunction, i.e., the product line $\{\{A\}, \{B\}, \{A, B\}\}$ with A, B two primitive features.*

Proof. COFT admits nodes without sons, namely the *xor*-node without sons xor_0 , where $\llbracket xor_0 \rrbracket = \emptyset$, the *and*-node without sons and_0 , where $\llbracket and_0 \rrbracket = \{\emptyset\}$. They are all product lines that exist with no primitive features.

If a primitive feature occurs on a non-leaf node, we replace this node by a *and* with the primitive feature and a new node similar to the non-leaf node.

Let r be the root, s_i its sons. Any primitive feature can only occur under one s_i , since it is a tree. We eliminate any s_j that contains neither A nor B : s_i is then equivalent to either and_0 or xor_0 , that we can simplify using the equalities:

$$\begin{aligned} xor(s_1, \dots, s_{i-1}, xor_0, s_{i+1}, \dots, s_k) \\ = xor(s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_k), \end{aligned} \quad (1)$$

$$and(s_1, \dots, s_{i-1}, xor_0, s_{i+1}, \dots, s_k) = xor_0, \quad (2)$$

$$\begin{aligned} xor(s_1, \dots, s_{i-1}, and_0, s_{i+1}, \dots, s_k) \\ = (xor(s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_k))^\circ, \end{aligned} \quad (3)$$

$$\begin{aligned} and(s_1, \dots, s_{i-1}, and_0, s_{i+1}, \dots, s_k) \\ = and(s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_k), \end{aligned} \quad (4)$$

$$and(s) = s, \quad (5)$$

$$xor(s) = s. \quad (6)$$

(Note that these simplification rules are only valid for trees!) If the two primitive features A, B occur in the same s_i , we can use the last two rules to replace the root by s_i . Thus we end this step with either:

1. and_0 (a.k.a. v in Fig. 9),
2. or xor_0 (a.k.a. f in Fig. 9),
3. or with a root with a single optional son with two grandsons s_1, s_2 ,
4. or the root is a *and*-node or a *xor*-node with two sons s_1, s_2 ,

where A occurs in s_1 and B in s_2 . Thus there are only four product lines that s_1 can express, because it

uses only one primitive A : and_0 , xor_0 , A , $\hat{A}$. The first two have already been eliminated. This leaves two product lines for s_1 , and the same for s_2 . There are thus 16 possibilities for this case: 2 (has the root a single optional son, or two sons?) times 2 (is the first node with two sons a xor - or a and -node?) times 2 (is s_1 equivalent to A or $\hat{A}$?) times 2 (is s_2 equivalent to B or $\hat{B}$?). There is also and_0 and xor_0 . Thus we have 18 simplified trees containing A , B . There are 16 possible product lines to express. Unfortunately, as noted in [27], four of them express $A|B = \{\emptyset, \{B\}, \{A\}\}$, and $A \vee B$ is thus missing, see Fig. 9.

Let us now consider constraints. Constraints can only remove products from the product line expressed by the tree D , thus we must start from a product line containing $A \vee B$. The only one is $\top = \{\emptyset, \{A\}, \{B\}, \{A, B\}\}$, from which we have to remove the base version $\emptyset$. Let s_1 and s_2 be the maximal nodes whose subtrees contain only A and B respectively. They must be brothers. To express $\top$, their parent must be a and -node, otherwise $\{A, B\} \notin [D]$. Then s_1 and s_2 must express $\hat{A}$ and $\hat{B}$, respectively, otherwise $\{A\} \notin [D]$. Thus the nodes of this tree can only express A , $\hat{A}$, B , $\hat{B}$, $\top$, perhaps and_0 , xor_0 . There are thus only 98 possible constraints, none of which expresses $A \vee B$. $\square$

In [36], van Gurp et al. have proposed two extensions: add or -nodes, and consider FD as a single-rooted acyclic graph rather than a tree. We will see in the next section that the second extension alone guarantees complete expressiveness. Adding or -nodes only, on the other hand, does not give complete expressiveness:

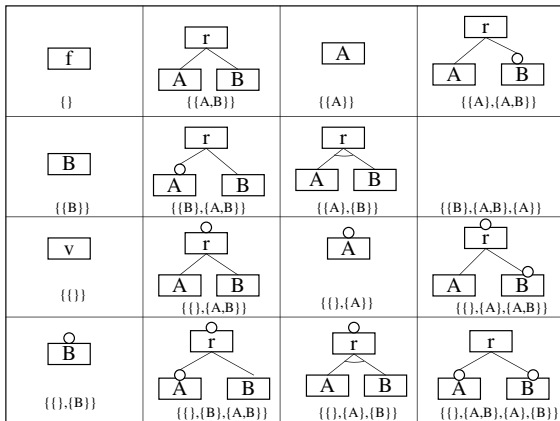


Fig. 9. Product lines on two primitive features and their OFT.

Theorem 4.2. CRFT (i.e., with or -nodes but no constraints) cannot express $vp(2 \cdots 2)$ among three features.

Proof. We note that when using trees, and , xor , or are associative. So without loss of generality, we can assume that the first node from the root that has more than one son, actually has two sons s_1 , s_2 : if it has more, we use associativity to break the list of sons in two:

$$xor(s_1, s_2, \dots, s_k) = xor_2(s_1, xor(s_2, \dots, s_k)), \quad (7)$$

$$and(s_1, s_2, \dots, s_k) = and_2(s_1, and(s_2, \dots, s_k)), \quad (8)$$

$$or(s_1, s_2, \dots, s_k) = or_2(s_1, or(s_2, \dots, s_k)). \quad (9)$$

Now each operator imposes its “shape” on the product line:

1. for and_2 , $PL = \{c_1 \cup c_2 | c_1 \in [s_1], c_2 \in [s_2]\}$,
2. for or_2 , $PL = \{c_1, c_2, c_1 \cup c_2 | c_1 \in [s_1], c_2 \in [s_2]\}$,
3. for xor_2 , $PL = [s_1] \cup [s_2]$.

For trees, $[s_1]$ and $[s_2]$ have disjoint primitive features, say, without loss of generality $\{A\}$, $\{B, C\} \cdot [vp(2 \cdots 2)]$ has none of these shapes. $\square$

The next step is then to add group cardinality (operator vp) to the language [28]. However, we guess that this does not lead to complete expressiveness for trees. Our argument goes through the complexity reasoning of Section 4.5 and must thus be postponed there. However, it is certainly possible to list all product lines to discover again the simplest one that is not expressible by vp , and then propose a new node type.

Note that one can also see these limited expressiveness results as a weakness of the constraint language. If nesting of constraints were allowed, Sheffer [33] has shown that *mutex* alone is complete, even on primitive features only. One could also use Boolean (a.k.a. propositional) logic or Boolean circuits to easily obtain complete expressiveness. For instance, [1] uses trees with optional, and , xor and or -nodes, with constraints in propositional logic. This is, of course, expressively complete.

4.1.2. Diagram variants

In this section, we examine what we call “diagrams” defined in the previous section. Unlike trees, they allow sharing of subgraphs. This small change provides complete expressiveness, i.e., every product line can be expressed by an OFD. We need a small tool to prove this.

Definition 16 (*First normal form*). The first normal form of a pl , noted $\mathcal{N}_1(pl)$, has a single *xor*-node as root, its *and*-nodes are the products of pl . It has edges from the root to each product, and from each product to its constituent primitive features: $n \rightarrow c$ iff $n = r \wedge c \in pl \vee n \in pl \wedge c \in n$, and no constraints, see, e.g., Fig. 10.

The product with no feature is represented by and_0 . The language of first normal forms is thus called $\mathcal{N}_1(PL)$.

Theorem 4.3. Every PL pl can be expressed by a COFD, e.g., by its first normal form: $\forall pl \in PL. \exists \mathcal{N}_1(pl) \in COFD. pl = \llbracket \mathcal{N}_1(pl) \rrbracket$.

Another alternative is to choose a rich, but fixed, graphical part, and to put all the information in constraints. This requires constraints on non-primitive features, that are not allowed in [1,35] but allowed in [19]:

Definition 17 (*Second normal form*). The second normal form of pl , noted $\mathcal{N}_2(pl)$ has the root as single *xor*-node, all possible combinations of features as *and*-nodes, no optional nodes. Then we use a mutex with root to exclude the combinations that should not be included in pl .

Theorem 4.4. For any set of primitive features P , there is an FD $\mathcal{N}_2(\{P\})$ such that any PL pl on P can be expressed by mutex constraints on $\mathcal{N}_2(\{P\})$.

One might ask whether COFD are the smallest expressively complete sublanguage. On one hand, we show easily that we cannot remove one of its two operators:

Theorem 4.5. $CFD(\text{and})$ is not expressively complete.

Proof. $P \in \llbracket CFD(\text{and}) \rrbracket$ iff P has exactly one model. $\square$

Theorem 4.6. $CFD(\text{xor})$ is not expressively complete.

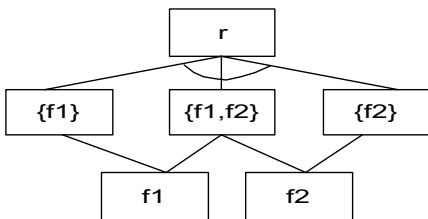


Fig. 10. First normal form of the product line $f1 \vee f2$.

Proof. $P \in \llbracket CFD(\text{xor}) \rrbracket$ iff each model of P contains exactly one path from the concept, and thus one primitive feature at most. $\square$

This section thus shows that all feature notations that allow sharing are expressively complete, while notations based on trees (i.e., forbidding sharing) are not, unless of course complemented by a very rich constraint language.

We thus need a finer yardstick than expressiveness to compare these languages. 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 even finer, naturalness (a.k.a. embeddability), that are explained in the following sections.

4.2. Embeddability

When languages have the same expressiveness, one needs finer criteria to compare them. We then know that there are two translations back and forth, and we have a closer look at them. We can compare the size of the diagram before and after translation: this is what is measured by succinctness. We can also see whether the translation is natural, where a natural translation [11,22] respects the structure of the original diagram:

Definition 18 (*Embeddability*). A context-free language $\mathcal{L}_1$ is embeddable into $\mathcal{L}_2$ iff there is an embedding, i.e., a translation $T: \mathcal{L}_1 \rightarrow \mathcal{L}_2$ that is compositional (or modular or homomorphic or natural): $T(C_1(\vec{x})) = C_2(T(\vec{x}))$, for all constructs C_1 of $\mathcal{L}_1$, where $\vec{x}$ is a vector of metavariables, adequately typed wrt. C_1 's subterms expected types, and C_2 is an expression in language $\mathcal{L}_2$ containing the same metavariables. C_2 is usually noted $T(C_1)$. A natural translation is called an embedding.

Said otherwise, an $\mathcal{L}_1$ compiler can be obtained by putting a trivial macro preprocessor in front of an $\mathcal{L}_2$ compiler.

The graphical languages considered in this paper have no context-free syntax, so that compositionality cannot apply. Therefore, we generalize it:

Definition 19 (*Graphical embeddability*). A graphical language $\mathcal{L}_1$ is embeddable into $\mathcal{L}_2$ iff there is a translation $T: \mathcal{L}_1 \rightarrow \mathcal{L}_2$ that is node-controlled [18]: T is expressed as a set of rules of the form $D_1 \rightarrow D_2$, 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.

Example. Assume a graphical language with *states* that are nodes and *transitions* that are edges. The node-controlled translation of a *state* must take the most general case, i.e., a state with an arbitrary number of incoming and outgoing *transitions*. However, D_1 will just contain the *defined state*, an incoming *transition* from some *state*, and an outgoing *transition* to some *state*. When we apply this translation to a concrete graph, we match a node, called the *application node*, with the *defined* node. All incoming *transitions* will be translated according to the translation of the incoming *transitions* in D_1 , and similarly for outgoing *transitions*.

Our second definition is a generalisation of the first one, if we view syntax trees as labeled graphs, and allow syntax trees with sharing. Due to this sharing, a graphical embedding is always linear in graph size, and thus embeddings are mainly used to compare languages of similar succinctness.

Example. For the Pascal programming language, a Pascal program will be represented here as a syntax DAG. A typical example of a textual compositional translation would be:

$$T(\text{for } i := l \text{ to } h \text{ do } S) = \text{begin } i := l; \\ \text{while } i \leq h \text{ do } S \text{ end.}$$

It is also a node-controlled translation, if we remember that *for* is the *defined* node. In D_1 , *for* is linked to nodes i, l, h, S that are to be understood as metavariables of the adequate syntactic category. In the tree form, i would be duplicated by the translation, and if i is big, this can blow up the translation. Here, in the graph form, i cannot be duplicated since it is allowed to occur only once in D_2 . But we can link the constant nodes of D_2 (*while*, etc.) twice to i . Thus we obtain the same net effect without blow-up.

Note that a syntactic inclusion is also an embedding.

Embeddability is widely considered as an adequate notion for comparing “natural expressiveness” of languages. Specially, non-trivial self-embeddability is considered as *harmful redundancy*. Said otherwise, the language is unnecessarily complex because it contains some easily definable construct C :

Definition 20 (*Harmful redundancy*). A language $\mathcal{L}$ is harmfully redundant iff there is a construct C in $\mathcal{L}$ that has a node-controlled translation in $\mathcal{L} \setminus C$.

All structuring principles are kept in the smaller language. Indeed, the replacement is purely local, and thus the new diagram has a new size linear in the size of the old diagram written in the redundant language.

OFDs have harmfully redundant optional nodes and textual mutual exclusion constraints, see Fig. 11. Similarly, EFDs have redundant optional edges and textual mutual exclusion constraints. In Fig. 11 and Table 2 we can compute the expansion factor: for instance, an optional node is replaced by three nodes and two edges, leading to an expansion factor of 5.

Similarly, *and*-, *or*-, *xor*- and optional nodes, *mutex* constraints are harmfully redundant w.r.t. *vp*-nodes, see Table 2.

4.3. Succinctness

Sometimes, languages of the same expressiveness cannot be compared by embeddings. This is to be expected since embeddings always yield a linear translation, and are thus a very refined comparison. A coarser one is provided by succinctness: this allows to measure the blow-up caused by a change of notation.

Definition 21 (*Succinctness*). Let $\mathcal{G}$ be a set of functions from $\mathbb{N} \rightarrow \mathbb{N}$. A language $\mathcal{L}_1$ is $\mathcal{G}$ -as succinct as $\mathcal{L}_2$, noted $\mathcal{L}_2 \leq \mathcal{G}(\mathcal{L}_1)$, iff there is a translation $T : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ that is within $\mathcal{G} : \exists_g \in \mathcal{G}, \forall n \in \mathbb{N}, \forall l_1 \in \mathcal{L}, |l_1| \leq n \Rightarrow |T(l_1)| \leq g(n)$. Common

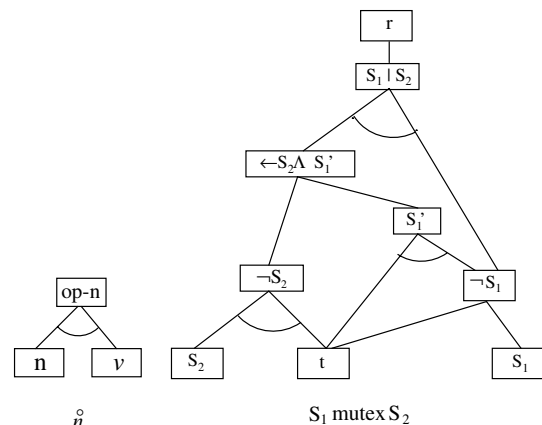


Fig. 11. Embedding for OFD.

values for $\mathcal{G}$ are “identically” = $\{n\}$, “thrice” = $\{3n\}$, “linearly” = $O(n)$, “cubically” = $O(n^3)$, “exponentially” = $O(2^n)$. We will omit “identically”.

Thus, this definition implies that $\mathcal{L}_2$ is at least as expressive as $\mathcal{L}_1$.

In Fig. 12, BC are the combinatorial Boolean electronic circuits [31,32,37]. They are acyclic graphs where nodes (called “gates”) bear a Boolean operator and they thus seem similar to FD. However, they operate differently: a gate computes the value of its operator op applied to the Boolean values e_i carried by its input links (or wires): $op_k(e_1, \dots, e_k)$. This output value is copied to all its output links. For simplicity, we only consider *nand* gates, since it is known that they are expressively complete [33].

For each notation we can define a graphical variant, where the constraints are drawn as edges labeled by the constraint. The textual variant is also a graph, since we consider its abstract syntax tree, where a constraint is coded by three edges and a node labeled by the constraint. Therefore the textual variant is considered here as four times less succinct, if the size of a graph is the number of edges and nodes. But the textual variant is linked to the root, a property that we used for embedding constraints (Fig. 11).

Syntactical inclusions are of course translations with no expansion.



Fig. 12. Translations between feature diagram languages.

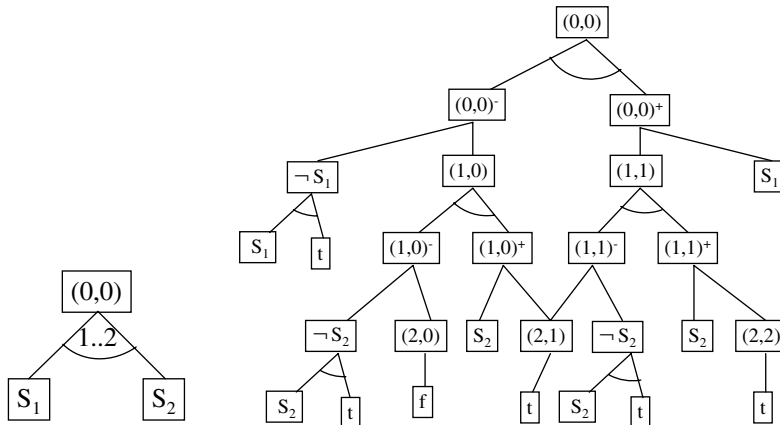


Fig. 13. Translation of a *vp*-node to COFD.

Theorem 4.7. $COFD \leq O(VFD^3)$.

Proof. We translate each variation point of multiplicity $\mu \cdots v$ by a subgraph of quadratic size in the number of sons k . Assume its sons are ordered as $s_1, \dots, s_k$. We introduce fresh *xor*₂-nodes of the form (i, j) where $0 \leq i \leq k$ is the number of sons treated, and $0 \leq j \leq i$ is the number of sons in the model among the sons treated. (We could collapse values of j above v). When $i = k$, if $\mu \leq j \leq v$, we replace it by the true node t . Else, we replace it by *xor*₀. Each (i, j) with $i < k$ has two *and*-sons, based on a case analysis of whether the son i is in the model. The first $(i, j)^+$ is *and*-node with sons s_i and $(i+1, j+1)$. The second $(i, j)^-$ is *and*-node with sons $\neg s_i$ and $(i+1, j)$. $\neg s_i$ is a *xor*₂-node with two sons, s_i and a true node. Node *xor*₀ can be simplified away. This new diagram starts from $(0, 0)$ (see Fig. 13). $\square$

Theorem 4.8. $COFD \leq O(BC)$.

Proof. We translate each wire of a Boolean circuit by two nodes: one to represent its value, the other for its negation. These two nodes are linked by a *xor*₂-node to the root, ensuring that they are indeed the negation of each other. We use thus two OFD, one for computing the result and the other for its negation. For a *nand* gate, this gives Fig. 14. This figure contains a *or*₂-node, i.e. a *vp*(1..2). It can be translated by the procedure above. The other gates can be obtained from a combination of *nand*-gates, or more efficiently translated directly. $\square$

These results compose. Further, the inclusions of [Theorem 3.1](#) can be used, as well as the linear translations between a textual and the corresponding graphical variant. Thus we have essentially five levels of succinctness:

1. Propositional logic is the standard in this domain.
2. Boolean circuits are exponentially more succinct.
3. COFD, OFD, RFD are more succinct because they are non-deterministic: checking that a product is in an FD is a non-trivial operation, while this is easy for the previous levels.
4. VFD, EFD are cubically more succinct, due to the use of vp.
5. They can all be linearly embedded into Σ_1 , the existentially quantified Boolean logic.

4.4. Redundancy

Riebisch et al. [27] criticized OFD and RFD as “ambiguous”. Let us first recall the definition of ambiguity:

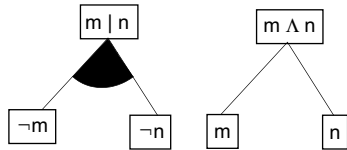


Fig. 14. Translation of a nand-gate to RFD, with xor_2 -nodes omitted.

Definition 22 (Ambiguity). A diagram (or sentence) is ambiguous iff it has two different meanings: $\llbracket D \rrbracket \neq \llbracket D \rrbracket$. A language is ambiguous iff it contains an ambiguous diagram or sentence.

This is obviously impossible with a formal semantics, since $\llbracket \cdot \rrbracket$ is a function.

Let us examine the “proof” of [27] in [Fig. 15](#).

This actually shows that $\llbracket D_1 \rrbracket = \llbracket D_2 \rrbracket = \llbracket D_3 \rrbracket$ and $\llbracket D_4 \rrbracket = \llbracket D_5 \rrbracket$. Let us name this property:

Definition 23 (Harmless redundancy. Normal form). A language $\mathcal{L}$ is harmlessly redundant iff $\exists D_1, D_2 \in \mathcal{L} \cdot D_1 \neq D_2 \wedge \llbracket D_1 \rrbracket = \llbracket D_2 \rrbracket$. Otherwise, it is a normal form (NF).

All the FDs seen here, except the two NFs ([Definitions 16 and 17](#)), are harmlessly redundant. This is the price to pay to have a decently natural language: in a normal form, all structure has to be lost.

Riebisch et al. proposed to solve the problem by adding multiplicities and optional edges to OFD, stating: “Such multiplicities cannot be expressed using the previous notations.” [27, p. 67], contradicting thus [Theorem 4.3](#). While the idea is good *per se*, it obviously cannot solve any of the alleged problems:

Theorem 4.9. If a language is ambiguous, adding constructs will keep it ambiguous.

Theorem 4.10. If a language is harmlessly redundant, adding constructs will keep it harmlessly redundant.

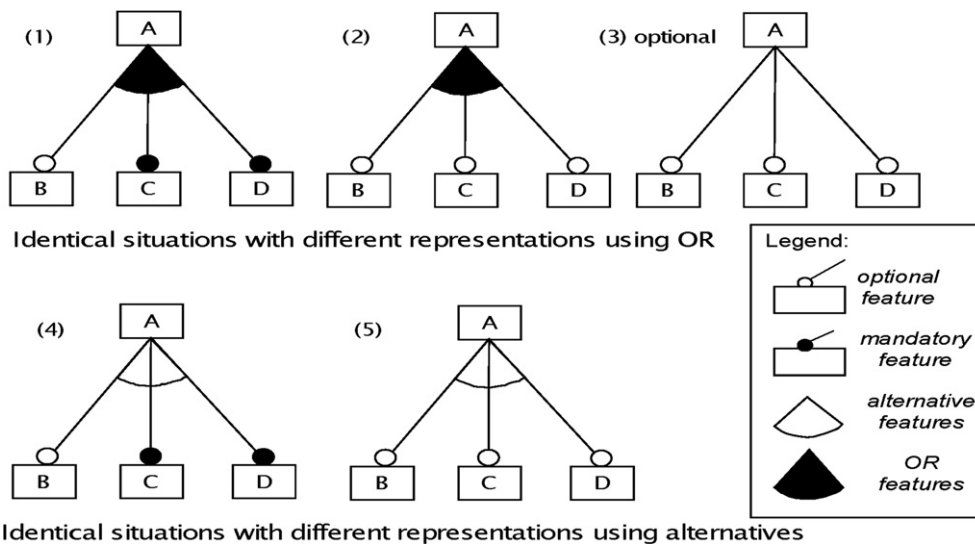


Fig. 15. “Ambiguity” example from [27].

Theorem 4.11. *If a language is harmfully redundant, adding constructs will keep it harmfully redundant.*

4.5. Decision problems

In this section, we consider three decision problems, which could be implemented by Computer-Assisted Software Engineering (CASE) tools supporting FD. Our envisioned CASE tool shall support two tasks:

1. designing and analyzing new FD, and
2. resolving variation points by interacting with a customer, to obtain a model and its product.

In the latter problem, the customer is asked questions following the root down to the leaves. When the customer chooses, this choice is propagated, pruning the FD. A choice that leads to no product should never be proposed to the user.

Definition 24 (*Satisfiability*). An FD diagram d is satisfiable iff $\llbracket d \rrbracket \neq \emptyset$.

Theorem 4.12. *Deciding whether an FD is satisfiable is NP-easy.*

Proof. Our semantics gives a translation from an FD to an existentially quantified propositional formula (a Σ_1 formula). Such a formula is satisfiable iff it is satisfiable without the existential quantifiers (said otherwise, if its matrix is satisfiable). This problem (SAT) is known to be NP-complete. $\square$

Theorem 4.13. *For any FD language including COFD (e.g., OFD, EFD) deciding whether an FD is satisfiable is NP-complete.*

Proof. Checking whether a set of clauses is satisfiable is known to be NP-complete. We take the literals as primitive features, each clause as an *or*-node. We add *xor*-nodes for each atomic predicate p , with two sons p and $\neg p$. The root is an *and*-node connected to all *or*-nodes and *xor*-nodes. We can eliminate *or*-nodes at the price of a quadratic expansion: a *or*-node with n sons l_i is replaced by a *xor*-node with n sons. Each of its sons s_i is a *and*-node with a son l_i and the other $n - 1$ are optional nodes l_j , $j \neq i$. Finally optional nodes can be replaced by *xor*-nodes, see Fig. 11. $\square$

Some FD notations are easier. The three notations below can be checked in constant time:

1. for diagrams in first normal form, just check that the root has a son;
2. for diagrams with *xor*-nodes only, just check that there is a primitive feature (to exclude the case xor_0);
3. diagrams with *and*-nodes only are always satisfiable.

Trees with *and*-, *xor*-, *vp*-, *or*-nodes can be treated in linear time:

1. put all primitive features on leaves by replacing the ones on internal nodes by a new *and*-node with two sons: a copy of the node and the primitive feature on a *and*-node without sons;
2. replace all *vp*-nodes whose lower bound is above the number of sons by xor_0 ;
3. a xor_0 son of a *xor*-node, a *vp*-node or a *or*-node can be simplified away;
4. if this brings the number of sons of a *vp*-node below its lower bound, the *vp*-node is simplified to xor_0 ;
5. a xor_0 son of an *and*-node allows to replace its parent by xor_0 .

This procedure either brings a xor_0 to the root and the tree is unsatisfiable, or eliminates all xor_0 from the tree, and the tree (and all its subtrees) is (are) satisfiable. In practice, we suggest enforcing this simplification automatically, by forbidding the entry of inconsistent cardinalities. The check is then in constant time. This result also implies that translating a propositional formula to these trees is NP-hard, or more likely impossible because they are not expressively complete.

Note that the satisfiability problem is NP-complete for all useful FD notations, where useful means expressively complete and polynomially succinct.

A related problem is product-checking: the customer has fully specified the products she wants in advance, and we must check whether it is in the product line.

Theorem 4.14. *For any FD language including COFD (e.g., OFD, EFD, VFD) product-checking is NP-complete.*

Proof. For easiness: we take the Σ_1 formula equivalent to the FD, simplify it according to the known

truth value of the primitive features. There remains to discover a valuation for the non-primitive features, which can be done by SAT.

For hardness: a SAT problem in clausal form can be translated to an OFD without primitive features. $\square$

Some FD notations are easier. The notations below can be checked in linear time:

1. for diagrams in first normal form, just check that the product is cited in the list;
2. for diagrams with *xor*-nodes only, if the product contains more than one primitive feature, the answer is no; else, we start from the primitive feature and go up until we reach the root;
3. for diagrams with *and*-nodes, we check that the given product is the set of primitive features reachable from the root;
4. for trees with *and*-, *xor*-, *vp*-, *or*-nodes, we mark the tree with truth values, starting from the primitive features whose values are given in the product. For true nodes, we must set their parent to true and check the satisfaction of its operator, until we reach the root.

The FD design task is made of several use cases. First, an analyst draws an FD representing the feature combinations that seem marketable. Since VFDs are now less popular than EFDs, (although less redundant, as advocated in the previous section) it may be easier to allow input in the syntax of any FD. The tool translates those into VFD, following the rules of Fig. 2, and lets the analyst switch between the two views. Doing so would ensure a better acceptance and a steeper learning curve.

She then tests or verifies the products obtained from the FD. This usually leads to the detection of feature interferences. They can often be fixed by changing the process to build the product, but if time does not allow, it is better not to market these feature combinations. In most cases, these interferences can be expressed by a *mutex* or *requires* constraint.

When an FD becomes too complex, the engineer may want to refactor it. If the refactoring is light, interior nodes stay the same and thus one might want to compare models:

Theorem 4.15. *For any FD language including COFD deciding whether two FD have the same valid models is coNP-complete.*

Proof. The proof is the same: we transform these FDs to clauses and conversely. Checking the equivalence of clauses is known to be coNP-complete. $\square$

Again, this is easier (linear) for simple FD languages. For diagrams in first normal form, diagrams with *xor*-nodes only with *xor*₀ eliminated, diagrams with *and*-nodes, just check that they are the same. Model-equivalence of trees with *and*-, *xor*-, *vp*-, *or*-nodes can also be treated in linear time, provided they are restricted to be consistent as indicated above: just replace *and*-, *xor*-, *or*-nodes by *vp*-nodes using Table 2, then check that the trees are the same: indeed, if they are not the same, either a *vp* has a different cardinality and we can construct different models, or it has different sons and again we can construct a model expressing this difference. In practice, we suggest enforcing the replacement automatically, by storing only *vp*-nodes internally. Model-equivalent trees will thus simply be stored once.

If deep refactoring is done manually, a natural question to ask is whether the PL described by this new FD is the same. Similarly, when PLs are developed by several persons, conflicts may arise. In order to solve them, it can be interesting to first compare PLs, and provide examples of products that are in the difference.

Recall that Π_1 is the first level of the polynomial hierarchy, just above NP [25].

Theorem 4.16. *For any FD language including COFD deciding whether two FDs are equivalent (describe the same product line) is Π_1 -complete.*

Proof. The standard problem for Π_1 is Π_1 -SAT, checking whether a Boolean formula preceded by universal Boolean quantifiers is satisfiable. An FD is transformed to a Σ_1 Boolean formula, by taking its semantics and quantifying existentially non-primitive features. Thus model-equivalence is logical equivalence of two Σ_1 Boolean formula, which is solvable by Π_1 -SAT. Conversely, given a Π_1 -SAT problem, we take the negation of its matrix (Boolean formula), convert it to a BC, and then this BC to an OFD using Theorem 4.8. We quantify existentially by removing primitive features in this OFD. This last OFD is equivalent to the OFD with all products (see, e.g., Fig. 9, lower right corner) iff the original formula is satisfiable. $\square$

Model-equivalence can be checked in linear time for simple languages:

1. for diagrams in first normal form, just check that they are the same;
2. for diagrams with *xor*-nodes only, just check that they have the same primitive features;
3. for diagrams with *and*-nodes, just check that they have the same primitive features.

When two feature interference engineers work independently, they will obtain two different restrictions of the initial product line. To put their work together, we must find an FD representing the intersection.

Theorem 4.17. *An FD expressing the intersection of two given FDs D_1 , D_2 can be computed in linear size and time.*

Proof. We put primitive features on leaves in D_1 , D_2 . Then we rename nodes (priming them in D_2 , say) so that they are disjoint in D_1 and D_2 . First we take the union of D_1 and D_2 (we mean the union of nodes, edges, constraints, respectively of course). This operation is called “putting D_1 and D_2 side by side”. Then we join their roots r_1 and r_2 by a fresh *and*₂-node r . Then we add constraints p requires p' and p' requires p for each primitive feature p (p' is the renaming of p in D_2 . It is not primitive, but a *and*₀ node.) □

Inversely, feature engineer’s duty is to add new interesting features to the product line. If two feature engineers work in parallel, they will already describe the forbidden combinations for the features they know. When putting the two extended product lines together, the most conservative option is to take the union: this cannot create feature interferences (beyond those already present in each PL separately).

Theorem 4.18. *An FD for the disjunction (or union) of two given FDs can be computed in linear size and time.*

Proof. We put D_1 and D_2 side by side but sharing primitive features and join their roots r_1 and r_2 by a fresh *xor*₂-node r . □

The disjunction is what will be initially offered to the customer, since it is safe. But of course the whole point of feature engineering is to offer as many feature combinations as possible, so the “bold” PL

that hopes that no interferences occur will be sent to feature interference engineers so that they detect unexpected interferences. This is obtained by a reduced product:

Definition 25. The reduced product of D_1 and D_2 , noted $D_1 \times D_2$, is $\{c_1 \cup c_2 | c_1 \in \llbracket D_1 \rrbracket, c_2 \in \llbracket D_2 \rrbracket\}$.

Note that the union is here computed at product level, hence the difference with the disjunction. We assume that D_1 and D_2 are identical for their common part ($P_1 \cap P_2$).

Theorem 4.19. *An FD for the reduced product of D_1 and D_2 can be computed in linear time and size.*

Proof. We put D_1 and D_2 side by side sharing primitive features, and join their roots r_1 and r_2 by a fresh *and*₂-node r . □

This proof concludes this section dedicated to studying and comparing the properties of FD languages from a formal perspective.

5. Related works

In this paper, we have studied the formal underpinnings of a family of languages in the line of the original feature trees [19], that have not yet received a formal definition, but where a product line semantics (see Definition 13) is clearly intended. Recently, however, a few more formally defined notations have surfaced:

1. van Deursen et al. [35] deal with a textual FD language to which they provide a semantics by coding rewriting rules in the ASF + SDF specification formalism associated to a tool-environment. Hence, contrary to ours, their semantics is neither tool-independent nor self-contained. Also, a major difference is that their semantics preserves the order of features.
2. 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.

3. In [8], 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. Benavides et al. [2] propose to use constraint programming to reason on feature models. They extend the FD language of [8] with extra-functional features, attributes and relations between attributes. Subsequently, they describe tool-support based on mapping those FDs to Constraint Satisfaction Problems (CSP).
5. Wang et al. [38] 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].

In general, we can argue that related approaches do not rank as well as ours on generality, abstraction and intuitiveness. For some approaches [35,1] the semantics is tool-driven, while on the contrary tools should be built according to a carefully chosen semantics. For the others, 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 [20] or binding times [36] should be further investigated.

Moreover, formality is only one specific quality of languages. For example, Krogstie [23] 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 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 address the complexity of engineering software product lines. However, after surveying past research, we have concluded that most of it has 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.

We have recalled standard, well-defined criteria to evaluate language variants, and measured most variants against these criteria. Many variants are expressively complete, which is a basic criterion. This also implies that they are closed under all Boolean operations, which is important for tool support. To choose among the expressively complete formalisms, we inspected their embeddings and their succinctness. Succinctness points to the use of cardinalities. If we look more finely using embeddings, we can eliminate redundant constructs. This process produces VFDs (Varied Feature Diagrams), an FD language that is expressively complete, harmfully irredundant, that can embed all other known variants, and thus is linearly as succinct.

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, but need more work in several respects including a concrete syntax and trial case studies.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful and constructive

comments. We also appreciated valuable feedback from the colleagues of the PRECISE team, Raimundas Matulevičius and Germain Saval. This work was partially supported by the Walloon Region and FSE.

References

- [1] Don S. Batory, Feature Models, Grammars, and Propositional Formulas, in: Obbink and Pohl [24], pp. 7–20.
- [2] David Benavides, Pablo Trinidad, Antonio Ruiz Cortés, Automated reasoning on feature models, in: Oscar Pastor, João Falcão e Cunha (Eds.), CAiSE, Lecture Notes in Computer Science, vol. 3520, Springer, 2005, pp. 491–503.
- [3] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Semantics of FODA feature diagrams, in: Tomi Männistö, Jan Bosch (Eds.), Proceedings of Workshop on Software Variability Management for Product Derivation (Towards Tool Support), Boston, August 2004.
- [4] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Generic Semantics of Feature Diagrams Variants, in: Proceedings of 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI), IOS Press, 2005.
- [5] Paul C. Clements, Linda Northrop, Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, 2001. August.
- [6] Sholom Cohen, Bedir Tekinerdogan, Krzysztof 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.
- [7] Krzysztof Czarnecki, Simon Helsen, Ulrich Eisenecker, Staged configuration using feature models. Software process improvement and practice, Process and Management 10 (2) (2005) 143–169, Special issue on Software Variability.
- [8] Krzysztof Czarnecki, Simon Helsen, Ulrich W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (1) (2005) 7–29.
- [9] Ulrich W. Eisenecker, Krzysztof Czarnecki, Generative Programming: Methods Tools and Applications, Addison-Wesley, 2000.
- [10] Magnus Eriksson, Jürgen Börstler, Kjell Borg, The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In Obbink and Pohl [24], pages 33–44.
- [11] M. Felleisen, On the expressive power of programming languages, in: N.D. Jones (Ed.), Proceedings of the Third European Symposium on Programming, Lecture Notes in Computer Science, Vol. 432, Springer-Verlag, 1990, pp. 134–151.
- [12] J.P. Gibson, Feature requirements models: understanding interactions, in: Feature Interactions in Telecommunications IV, IOS Press, 1997.
- [13] Hassan Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley Longman Publishing, Redwood City, CA, USA, 2004.
- [14] M. Griss, J. Favaro, M. d'Alessandro, Integrating feature modeling with the RSEB, in: Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 1998, pp. 76–85.
- [15] Martin L. Griss, Implementing product-line features with component reuse, in: ICSR-6: Proceedings of the 6th International Conference on Software Reuse, Springer-Verlag, 2000, pp. 137–152.
- [16] D. Harel, B. Rumpe, Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff, Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, 2000.
- [17] D. Harel, B. Rumpe, Meaningful modeling: what's the semantics of “semantics”? IEEE Computer 37 (10) (2004) 64–72.
- [18] Dirk Janssens, Grzegorz Rozenberg, On the structure of node label controlled graph languages, Information Science 20 (1980) 191–244.
- [19] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
- [20] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, FORM: a feature-oriented reuse method, Annals of Software Engineering 5 (1998) 143–168.
- [21] Kyo Chul Kang, Moonzoo Kim, Jaejoon Lee, Byungkil Kim, Feature-oriented re-engineering of legacy systems into product line assets – a case study. In Obbink and Pohl [24], pp. 45–56.
- [22] Stephen Cole Kleene, Introduction to metamathematics, Bibliotheca Mathematica, vol. I, North-Holland, Amsterdam, 1952.
- [23] John Krogstie, A semiotic approach to quality in requirements specifications, Organizational Semiotics (2001) 231–249.
- [24] J. Henk Obbink, Klaus Pohl (Eds.), Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26–29, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3714, Springer, 2005.
- [25] Christos H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
- [26] Klaus Pohl, Gunter Bockle, Frank van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005, July.
- [27] 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.
- [28] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, Ilka Philippow, Extending feature diagrams with UML multiplicities, in: Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT2002), Pasadena, CA, June 2002.
- [29] Mark Ryan, Pierre-Yves Schobbens, Fire works: a formal transformation-based model-driven approach to features in product lines, in: Tomi Männistö, Jan Bosch (Eds.), Proceedings of the WS on Software Variability Management for Product Derivation – Towards Tool Support, 2004.
- [30] Douglas C. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 25–31.
- [31] Claude Shannon, A symbolic analysis of relay and switching circuits, Master's Thesis, MIT, 1937.

- [32] Claude Shannon, A symbolic analysis of relay and switching circuits, *Transactions American Institute of Electrical Engineers* 57 (1938) 713–723.
- [33] Henry M. Sheffer, A set of five independent postulates for boolean algebras, with application to logical constants, *Transactions of AMS* 14 (1913) 481–488.
- [34] D. Streitferdt, M. Riebisch, 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, IEEE Computer Society Press, 2003, pp. 297–304.
- [35] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, *Journal of Computing and Information Technology* (2001).
- [36] Jilles van Gurp, Jan Bosch, Mikael svahnberg, on the notion of variability in software product lines, in: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, 2001.
- [37] Heribert Vollmer, *Introduction to Circuit Complexity: A Uniform Approach*, Springer-Verlag, New York, 1999.
- [38] Hai Wang, LI Yuan Fang, Jing Sun, Hongyu Zhang, Jeff Z. Pan, A semantic web approach to feature modeling and verification, in: *Proceedings of the International Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2005.

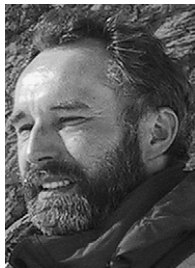


Jean-Christophe Trigaux holds an M.Sc. in Computer Science from the University of Namur, Belgium. He is a Ph.D. student at the PRECISE Lab, CS Department, University of Namur where he works under the supervision of Prof. Patrick Heymans and Prof. Pierre-Yves Schobbens. The topic of his thesis is a formal approach to the modelling of software product lines. His doctoral studies are funded by the Walloon

Region and FSE under the FIRST Europe Objective 3 project PLENTY (Contract No. 215315).



Yves Bontemps holds an M.Sc. and a Ph.D. in Computer Science from the University of Namur, Belgium. In 2005, he completed his Ph.D. Thesis under the supervision of Prof. Pierre-Yves Schobbens. His doctoral studies were funded by a Belgian FNRS “Research Fellow” grant. He is currently a full-time consultant with SmalS-MvM/eGov, Belgium.



Pierre-Yves Schobbens holds M.Sc. and a Ph.D. from University of Louvain, Belgium. He is a full professor at the PRECISE Lab, CS Department, University of Namur. His research interests include formal semantics, logics in computer science, model checking, UML semantics and agent-oriented specification.



Patrick Heymans holds an M.Sc. and a Ph.D. in Computer Science from the University of Namur. He is a full professor at the PRECISE Lab, CS Department, University of Namur. His research interests include Software Modelling Languages, Requirements Engineering and Software Product Lines.