Technical Report:
Evaluating Formal Properties of Feature Diagram Languages

Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Raimundas Matulevičius, Yves Bontemps and Andreas Classen

Project: Product Line ENgineering of food TraceabilitY software
Financing: "Région wallonne" and "Fonds Social Européen"

First Europe Objectif 3

EPH3310300R0462 / 215315

November 2006
Abstract

Feature Diagrams (FDs) are a family of popular modelling languages, mainly used for engineering requirements in software product lines. FDs were first introduced by Kang et al., as part of the FODA (Feature Oriented Domain Analysis) method back in 1990. Since then, various extensions of FODA FD were devised to compensate for purported ambiguity and lack of precision and expressiveness. Recently, we surveyed these notations and provided them with a generic formal syntax and semantics, called FFD (Free Feature Diagrams). We also started investigating the comparative semantics of FFD with respect to other recent formalisations of FD languages. Those results were targeted at improving the quality of FD languages and making the comparison between them more objective.

In this paper, we briefly recall our previous results. Most importantly, we give a more comprehensive account of our research method: (1) we provide better illustration, motivation and discussion of the technical choices at the root of FFD and (2) we put our contributions in a wider perspective. The main questions to which we try to answer are: What are the basic concepts and principles on which our investigation relies and why? How are our contributions to be situated in the wider endeavour of improving the quality of FD languages? How must the investigation be pursued and how can it be complemented?

1. Introduction

A software product line (SPL) is “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [1]. Software Product Line Engineering (SPLE) is a rapidly emerging software engineering paradigm that institutionalises reuse throughout software development. By adopting SPLE, one expects to benefit from scale economies and thereby improve the cost but also the productivity, time to market and quality of developing software.

As illustrated in Fig. 1, one of the main ideas behind SPLE is to dedicate a specific process, named Domain Engineering, to the development of reusable artifacts, a.k.a core assets. These core assets are then reused extensively during the development of final products, called Application Engineering.

- Domain Engineering, a.k.a development for reuse, consists in producing reference requirements, a reference architecture and reusable components. Domain engineering consists roughly of 3 activities: domain analysis, domain design and domain implementation.
- Application Engineering, a.k.a development with reuse, consists in developing the final products (applications), reusing the core assets and adapting the products to the specific requirements. This process is composed of three phases: application requirements engineering, application design and application coding.

Central to the SPL 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” [3]. In order to tackle the complexity of variability management, a number of supporting modelling languages have been proposed. To represent variability at the requirements level, an increasingly popular family of notations is the one of Feature Diagrams (FD). FDs are mostly used to model the variability of application “features” at a relatively high level of granularity. Their main purposes are (1) to capture feature commonalities and variabilities, (2) to represent dependencies between features, and (3) to determine combinations of features that are allowed and disallowed in the SPL.

During 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 [4]. An example of a FODA FD is given in Fig. 2. It 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 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. The edges are used to progressively decompose features into more detailed features (also called subfeatures, or sometimes sons). FODA offers 2 kinds of decomposition:

1. and-decomposition, e.g., between Monitor Fuel Consumption and its sons, Measures and Methods. It indicates that the latter two features should both be present in all feature combinations where Monitor Fuel Consumption is present.

1Inspired from a case study defined in [5].
2. *xor-decomposition*, where edges are linked by an arc, as between *Measures* and its sons, L/Km and Miles/gallon. It indicates that only one of the latter two features should be present in combinations where *Measures* is.

Finally, in FODA, there is also a textual sublanguage called *Composition Rules* (or CR) that lets one express constraints between features that crosscut the tree structure. In Fig. 2, there is only one such constraint, located at the bottom of the diagram. It uses the special keyword *requires* to indicate that when the former feature is present, the latter should be too.

This paper will later survey our previous contributions [6, 7, 8, 9] to give a precise meaning of these and other constructs. However, for now, we retain that the “meaning” of such diagrams seems to be concerned with the possible combinations (or configurations) of features within any of the products in the SPL—and this was indeed acknowledged in FODA [4] and most of its successors (see below). For instance, one of the allowed feature combinations of the FD in Fig. 2 is: Monitor Engine System, Monitor Engine-Performance, Monitor Temperatures, Oil, Engine, Transmission, Monitor RPM, Monitor exhaust levels and temperature, Monitor Fuel Consumption, Measures, L/Km, Methods and Based on type of driving. This combination corresponds to one of the SPL’s valid configurations which (1) does not monitor the coolant temperatures, (2) bases the fuel consumption monitoring on the type of driving, and (3) uses L/Km as measuring unit. The FD thus represents a SPL with three variation points (the features Coolant, Measures and Methods) and twelve different valid configurations. The complexity of this particular FD is relatively low. Consequently, the allowed configurations are not so many, and they can be computed relatively easily—provided that we have a precise semantics. However, we should note that there is an exponential factor involved in determining the number of resulting configurations, which mainly depends on the number of variation points and possible choices associated to each of them.

Since Kang et al.’s initial proposal [4], several extensions to FODA have been devised as part of the following methods: FORM [10], FeatureRSEB [11], Generative Programming [12], PLUSS [13], and in the work of the following authors: Riebisch et al. [14, 15], van Gurp et al. [16]. A brief overview of these proposals is given in Fig. 4 and 5 where, for each proposed FD language, a quick-facts sheet is given in the left column while a FD based on the same example as Fig.2 is given in the right column.

When looking at the FDs in Fig. 4 and Fig. 5, one immediately sees aesthetic differences among languages. For example, we have observed at least 5 different notations for the xor-decomposition construct (see
Fig. 2. FODA (OFT) feature diagram: the Monitor Engine System

These kinds of issues mainly concern concrete syntax, i.e. what the users see. Although concrete syntax is an important issue in its own right [17], in this work, we focus on what is really behind the pictures, i.e., semantics. We noticed that proponents of FD languages often claimed for added value of their language in terms of precision, unambiguity or expressiveness. Nevertheless, our previous work [6, 7, 8, 9, 18] demonstrated that the terminology and evaluation criteria that they used to justify these claims were often vague, and sometimes even misleading.

This is what the current paper has to offer: a method to evaluate and compare FD languages focused on the study of their semantics. This method relies on formally defined criteria and terminology, based on the highest standards in formal language definition [26, 27].

This paper is structured as follows. In Section 2, we situate our work within SEQUAL [28, 29], a comprehensive framework for assessing the quality of modelling languages. Section 3, based on [26, 27], recalls the basic concepts on which our method relies: concrete syntax, abstract syntax, semantic function and semantic domain. On these grounds, the section continues with the definition of the criteria that our method aims to investigate: expressiveness, embeddability (also called naturalness), succinctness and (computational) complexity. The method that we propose in order to evaluate how FD languages meet these criteria is described in Section 4. It emphasises how languages with no clearly defined semantics, or with different semantics domains, can be made suitable for comparison. Section 5 summarises the results obtained so far [8, 9, 18] by applying the method. The paper finishes by discussing the current limitations of the method (Section 6), the research challenges that are still ahead (Section 7), and the conclusions (Section 8).
<table>
<thead>
<tr>
<th>Survey Short Name</th>
<th>Method</th>
<th>Author</th>
<th>Year</th>
<th>Graph Type</th>
<th>Decomposition Types</th>
<th>Constraint Types</th>
<th>Formal Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFD</td>
<td>FeatuRSEB</td>
<td>Griss et al. [16]</td>
<td>1998</td>
<td>DAG</td>
<td>and, xor, opt, or</td>
<td>Textual &amp; Graphical</td>
<td>- Originally: none - A posteriori: [8]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Survey of feature diagram languages (1/2)
<table>
<thead>
<tr>
<th>Main Characteristics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Survey Short Name:</strong> VBFD</td>
<td>![Diagram of VBFD]</td>
</tr>
<tr>
<td><strong>Method:</strong> /</td>
<td></td>
</tr>
<tr>
<td><strong>Author:</strong> van Gurp et al. [16]</td>
<td><strong>Monitor Engine Performance</strong></td>
</tr>
<tr>
<td><strong>Year:</strong> 2001</td>
<td><strong>Monitor Fuel Consumption</strong></td>
</tr>
<tr>
<td><strong>Graph Type:</strong> DAG</td>
<td><strong>Monitor exhaust levels and temperature</strong></td>
</tr>
<tr>
<td><strong>Decomposition Types:</strong> <em>and, xor, or, opt</em></td>
<td><strong>Measures</strong></td>
</tr>
<tr>
<td><strong>Constraint Types:</strong> Textual &amp; Graphical</td>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td><strong>Formal Semantics:</strong></td>
<td><strong>Based on distance</strong></td>
</tr>
<tr>
<td>- Originally: none</td>
<td><strong>Based on type of driving</strong></td>
</tr>
<tr>
<td>- A posteriori: [8]</td>
<td><strong>Based on drive</strong></td>
</tr>
</tbody>
</table>

| **Survey Short Name:** EFD | ![Diagram of EFD] |
| **Method:** FORE | **Monitor Engine system** |
| **Author:** Riebisch et al. [14,15] | **Monitor Engine Performance** |
| **Year:** 2002 | **Monitor Fuel Consumption** |
| **Graph Type:** DAG | **Monitor exhaust levels and temperature** |
| **Decomposition Types:** *card, opt* | **Measures** |
| **Constraint Types:** Textual & Graphical | **Methods** |
| **Formal Semantics:** | **Based on distance** |
| - Originally: none | **Based on type of driving** |
| - A posteriori: [8] | **Based on drive** |

| **Survey Short Name:** VFD | ![Diagram of VFD] |
| **Method:** / | **Monitor Engine system** |
| **Author:** Bontemps et al. [6] | **Monitor Engine Performance** |
| **Year:** 2004 | **Monitor Fuel Consumption** |
| **Graph Type:** DAG | **Monitor exhaust levels and temperature** |
| **Decomposition Types:** *card* | **Measures** |
| **Constraint Types:** / | **Methods** |
| **Formal Semantics:** | **Based on distance** |
| - Originally: [6] | **Based on type of driving** |
| - A posteriori: [8] | **Based on drive** |

| **Survey Short Name:** PFT | ![Diagram of PFT] |
| **Method:** PLUSS | **Monitor Engine system** |
| **Author:** Eriksson et al. [13] | **Monitor Engine Performance** |
| **Year:** 2005 | **Monitor Fuel Consumption** |
| **Graph Type:** Tree | **Monitor exhaust levels and temperature** |
| **Decomposition Types:** *and, xor, opt, or* | **Measures** |
| **Constraint Types:** Graphical | **Methods** |
| **Formal Semantics:** | **Based on distance** |
| - Originally: none | **Based on type of driving** |
| - A posteriori: [8] | **Based on drive** |

*Figure 5. Survey of feature diagram languages (2/2)*
2. Quality of Models and Languages

2.1. Model Quality

Assessing and improving the quality of modelling is a complex and multidimensional task. A comprehensive view of the various concerns involved is given in the SEQUAL (semiotic quality) framework, developed over the last decade by Krogstie et al. [29] as an extension of Lindland et al.’s original framework [30]. SEQUAL is based on a distinction between “semiotic levels”: syntactic, semantic and pragmatic. It adheres to a constructivistic world-view that recognises model creation as part of a dialog between participants whose knowledge changes as the process takes place.

An overview of SEQUAL is given in Fig. 6. It highlights various broad qualities that models should have. Physical quality pursues two basic goals: externalisation, meaning that the explicit knowledge \( K_E \) of a participant has to be externalised in the model \( M \) by the use of a modelling language \( L \); and internalisability, meaning that the externalised model \( M \) can be made persistent and available, enabling other stakeholders to make sense of it. Empirical quality deals with error frequencies when reading or writing \( M \), as well as coding and ergonomy of human-computer interaction in modelling tools. Syntactic quality is the correspondence between \( M \) and the language \( L \) in which \( M \) is written. Semantic quality examines the correspondence between \( M \) and the domain, or universe of discourse, \( D \). Pragmatic quality assesses the correspondence between \( M \) and its social as well as its technical audiences’ interpretations, respectively, \( I \) and \( T \). Perceived semantic quality is the correspondence between the participants’ interpretation \( I \) of \( M \) and the participants’ current explicit knowledge \( K_E \). Social quality seeks agreement among the participants’ interpretations \( I \). Finally, organisational quality looks at how the modelling goals \( G \) are fulfilled by \( M \).

In [28, 29], one can find a more accurate description of the SEQUAL framework as well as suggestions of concrete means to pursue and measure the achievement of the quality goals that we have just briefly presented. By the extent of the envisaged model qualities and its neutrality wrt a particular kind of models, SEQUAL is arguably the most complete framework we know of. However, SEQUAL does not go into much details on how to carry out specific quality evaluation or improvement tasks, as other proposals have addressed, e.g., [31], [32], or [33]. Nevertheless, SEQUAL is amenable to specific criteria and guidelines by tailoring. Its main advantages are that (1) it helps situate one’s investigations within a comprehensive quality space, (2) it acts as a checklist of qualities to be pursued and (3) it recommends general guidelines on how to proceed.

Our investigation is targeted at a specific kind of models, namely, FDs. Furthermore, we mainly target semantic and pragmatic qualities of these models, which we have found to be somehow neglected in the current state of the art (see [8, 9]). So doing, we will see that we inevitably interfere with other qualities, mainly syntactic quality.

The problem with evaluating model quality is that representative objects of study – that is, models – do not always exist, or at least are not easily available. And this is indeed the case for FDs which (1) are an emerging modelling paradigm, and (2) have the purpose of representing highly strategic company information. Therefore, representative models are almost nowhere to find. At this stage, we thus thought we should concentrate on improving the quality of FD languages before any standardisation is attempted and they hopefully become widespread in industry.

2.2. Language Quality

In [28], SEQUAL has been adapted to evaluate language appropriateness (see Fig. 7). Six quality areas were proposed. Domain appropriateness means that language \( L \) must be powerful enough to express anything in the domain \( D \) [34], and that, on the other hand it should not be possible to express things that are not in \( D \). Participant language knowledge appropriateness measures how the statements of \( L \) used by the participants match the explicit knowledge \( K \) of the
participants. Knowledge externalisability appropriateness means that there are no statements in \( K \) that cannot be expressed in \( L \). Comprehensibility appropriateness means that language users understand all possible statements of \( L \). Technical actor interpretation appropriateness defines the degree to which the language lends itself to automatic reasoning and supports analysability and executability. Finally, organisational appropriateness relates \( L \) to standards and other needs within the organisational context of modelling.

Not being able to assess model qualities directly, our investigations were targeted at 3 main language qualities: domain appropriateness, comprehensibility appropriateness and technical actor interpretation appropriateness. The matching of the investigated criteria wrt these qualities is further discussed in Section 6. In the next section, we will first introduce the basic notions behind these criteria (Section 3.1), and then the criteria themselves (Section 3.2).

![Figure 7. SEQUAL : Language Quality (adapted from [28, 29])](image)

### 3. Formal Comparison Criteria

#### 3.1. Basic Principles

**3.1.1. Formal definition of visual languages.** In their illuminating papers “Meaningful modelling: What’s the Semantics of ‘Semantics’?” [26] and “Syntax, Semantics and All That Stuff. Part I: The Basic Stuff” [27], Harel and Rumpe recognise that:

“Much confusion surrounds the proper definition of complex modelling languages [. . .]. At the root of the problem is insufficient re-

Although they are far less complex than, e.g., the Unified Modelling Language (UML) [35], we demonstrated in previous papers [6, 7, 8, 9] that FDs are also “victims” of similar “mistreatments”. The objective of this section is to recall clearly the basic notions of (formal) syntax and semantics from [26, 27]. In the subsequent sections, we will show how, based on these notions, we have devised an approach to (re)define, assess and compare languages.

Very basically, the term ‘syntax’ refers to the notation that a language offers to its users. ‘Semantics’, on the other hand, refers to the meanings that its expressions (programs, diagrams, sentences, . . . ) are aimed to convey.

More precisely, Harel and Rumpe make it clear that the unambiguous definition of a modelling language, be it textual or graphical, must consist of three equally necessary elements: a syntactic domain (\( L \)), a semantic domain (\( S \)) and a semantic function (\( M \)) (see Fig. 8).

Furthermore, the authors argue that \( L \), \( S \) and \( M \) must all be defined formally, i.e. mathematically, in order to keep the risk of ambiguity at its minimum. A language with such formal \( L \), \( S \) and \( M \) is called a formal language. For programming languages, it is easy to understand why they need to be formal: given the same input, a given program must deliver exactly the same output whatever the interpreter that executes it. A formal programming language leaves no ambiguity in this regard to the implementers of interpreters.

![Figure 8. The three constituents of a formal language: syntactic domain (\( L \)), semantic domain (\( S \)) and semantic function (\( M \))](image)

For modelling languages, we do not necessarily need to execute diagrams, especially if the language’s

\(^2\)In [26, 27], one of Harel and Rumpe’s main motivations is to suggest how to improve the UML.
purpose is not to represent behaviours that could be executed, animated or simulated in some way. However, most of the time, there is a great interest in carrying out some computations on diagrams in order to derive properties about their meaning. For example, it might be extremely useful to have a tool that tells whether a given FD allows for at least one feature configuration, or if it is overconstrained and thereby allows none. For a realistic FD, this verification is far from trivial and it can be very time-consuming and error-prone if left to humans. This type of verification is known as satisfiability checking and is one of the many FD-related tasks that can be automated [36, 8, 9] (see also Section 3.4). If we want to describe languages in such a way that no ambiguity is left to tool developers, then we need a formal modelling language. Only then can we prove the correctness of its supporting algorithms or study their efficiency (computational complexity). And only then can we study formal language properties such as expressiveness, succinctness and embeddability (see Section 3.2).

Finally, and maybe most importantly, we should not forget that modelling languages are mainly used to ease communication between human participants [17]. If a language has no well identified, clear and concise definition of a semantics to which one can refer in case of doubt, an expression in this language might well convey an unintended meaning.

The above may seem all too obvious to some readers. However, during our survey, we could observe that many FD languages were never formally defined, despite the actual simplicity of the task (as we will see in Section 3.1.2). Maybe, some answers to why this is so are given in [26, 27] where the authors point out of set of frequent misconceptions about formal semantics, e.g., “Semantics is the metamodel”, “Semantics is dealing with behaviour”, “Semantics is being executable”, “Semantics means looking mathematical”, etc. We redirect the reader to [26, 27] for a brilliant demystification of this folklore. For now, let us return to the definitions of $L$, $S$ and $M$, which we make more precise.

**Syntax** In diagrammatic (a.k.a. visual, or graphical) languages, such as FDs, basic expressions include lines, arrows, closed curves, boxes and composition mechanisms involve connectivity, partitioning and “inside-ness” [26]. These form the physical representation of the data (on screen, or on paper) which is known as concrete syntax.

Most of the (informal) definitions of the semantics of FDs we found in the literature were based on concrete syntax, and usually discussed on FD examples. Most of the time, a substantial part of the semantics was implicit, leaving it to the diagrams to “speak for themselves”. But actually, each readers’ intuition is potentially different. And if we need to provide computer support, we need to make everything explicit. As Harel and Rumpe put it:

“It is possible to guess the meaning of most terms, since a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm. But a computer cannot act on such assumptions. To be useful in the computing arena, any language – whether it is textual or visual or used for programming, requirements, specification, or design – must come complete with rigid rules that clearly state allowable syntactic expressions and give rigid description of their meaning” [26].

According to the state of the art in compilation and formal methods, it is better not to use concrete syntax as a basis to define semantics. One reason is that, for visual languages, it appears particularly difficult to define rigid syntactic rules that clearly segregate between the allowed and the forbidden diagrams (this is also true, to a lesser extent, of textual languages). Another reason is that, when based on the concrete syntax, the expression of the semantic interpretation rules is polluted by considerations related to visualisation, which makes the rules very cumbersome. This is why the common practice in the aforementioned areas is to define the semantics of a language based on a so-called abstract syntax.

The abstract syntax is a representation of data that is independent of its physical representation and of the machine-internal structures and encodings. The set of all data that comply with a given abstract syntax is called the syntactic domain. Very often, the abstract syntax and the syntactic domain are both called the language and noted $L$. The former corresponds to the intensional definition of language whereas the second corresponds to its extensional (usually infinite) definition. Independence from machine-internal structures and encodings is necessary (1) to make the description of the syntactic rules as simple as possible, and (2) to make the rules portable from one implementation environment to another.

For visual languages, the two most widespread ways to define an abstract syntax are: (1) mathematical notation (set theory) and (2) meta-modelling. In the latter case, the abstract syntax is described through a so-called meta-model describing what is a well-formed (allowed) diagram. A meta-model is usually a UML Class Diagram (CD), possibly complemented with Object Constraint Language (OCL) rules [35]. This format
has the main advantage to be easily readable (UML CDs are a standard visual language with a partially well-accepted, although informal, semantics\(^3\)), and to facilitate some tool implementation tasks, especially persistent storage of diagrams in a repository. Nevertheless, we prefer the mathematical format for its greater universality, unambiguity, conciseness and suitability to undergo rigorous proofs [8]. As an example, in Section 3.1.2, we provide an abstract syntax for OFD. In Section 4.2.1, we will recall how we managed to provide an abstract syntax for several FD languages at once through a generic mathematical structure that we called FFD [8, 9].

A final remark concerning abstract syntax is that it is very often mistaken for the semantics. As mentioned earlier, this is a very frequent misconception about visual languages, especially the UML [26]. However, although several meta-models of FD exist in the literature [39, 40, 41, 42, 20, 43, 44, 25, 45], we did not observe this misconception being made about FDs.

**Semantics** Once we have established a rigid set of syntactic rules, the role of a *semantics* is to assign an unambiguous meaning to each syntactically correct diagram. Harel and Rumpe recognise that “agreement on a language’s meaning is partly a sociological process, without which the communicated data is worthless” [26]. As we have seen in Section 2, this point of view is acknowledged and further elaborated in [28, 29] which adopts a constructivistic view of the modelling activity. The sociological aspects of semantics are however out of the scope of the current investigation, although some elements of discussion will be given in Section 7. For the moment, we stick to a view where “a language’s semantics must provide the meaning of each expression, and that meaning must be an element in some well-defined and well-understood domain” [26]. Following [26, 27], a semantics must have two main constituents: a *semantic domain* (\(S\)) and a *semantic function* (\(M\)). For describing them in the most universal, unambiguous and concise way, we opt again for mathematics.

According to [26], the semantic domain “[…] specifies the very concepts that exist in the universe of discourse. As such, it serves as an abstraction of reality, capturing decisions about the kinds of things the language should express”. Typically, a semantic domain is a mathematical domain built to have the same structure as the real-world objects the language is used to account for, up to some level of fidelity. The semantic domain that we have proposed for OFD (i.e. FORM FDS) and for the other surveyed FD languages is named \(PL\) (Product Lines) [8, 9]. It is recalled in Section 3.1.2.

Having an explicit and well-defined semantic domain is crucial to get a clear idea of the kind of things that the modelling language should be used to represent. Without an explicit definition of it, it is difficult to judge of the appropriateness of the language with respect to (1) the application domain in which the language is to be used, (2) the usage that the participants want to make of it, and (3) the tools that will support it. Furthermore, looking at the semantic domain is necessary to compare two semantic definitions, as we will show in Sections 3.2 and 3.3.

The second constituent of a semantics is \(M\), the *semantic function*. It is the missing link to relate \(L\) and \(S\), that is, to eventually assign a meaning to each syntactically allowable diagram (see Fig. 8). The signature of this function is therefore simply \(M : L \rightarrow S\). In the case of OFD, we have the signature\(^4\) \(M_{OFD} : OFD \rightarrow PL\). Equally important as the signature is the definition of the function. The definition of \(M_{OFD}\) is given in Section 3.1.2 as an example. Semantic functions, like all mathematical functions, can be described in a number of ways. In our example, we found it convenient to express it as four fairly simple rules. These rules describe declaratively how of the syntactic constructs used in the diagram constrain its associated object in \(PL\). The generic semantic function of FFD (\(M_{FFD}\)) is not much more complex, as it counts just one additional simple rule (see Section 4.2.3).

An important point is that the definition of \(M\) should be rigid too, that is, it should make it crystal clear which object (meaning) in \(S\) is assigned to each allowable diagram of \(L\). Since \(M\) is a function, there is one and only one such object for a given allowable diagram. Ambiguity in this context is therefore not possible. As trivial as this may seem, it appears that the concept of ambiguity was not always properly used in the surveyed literature. For example, OFT (i.e., FODA FDs) have been criticised for being ambiguous [15]. However, having reconstructed a proper formal semantics from the original plain English definition of OFT [4], we could check that this was not the case [6].

Finally, the semantic function should be *total*, that is, it should not be possible to have a diagram in \(L\) which is not given a meaning in \(S\) by \(M\). A total semantic function ensures that the definition of the semantics is complete. In section 3.2.1, we will address the converse question: is every element in \(S\) expressible by a

---

\(^3\)Several formal semantics of CD have been proposed in the literature [37, 38] but, at the time being, none appears in the official standard [35].

\(^4\)The signature is similar for the other surveyed FD languages, e.g., \(M_{RFD} : RFD \rightarrow PL\) for RFD, etc.
To abstract the various kinds of decompositions, we introduce a labelling function $\lambda$ that maps each node to a boolean $\textit{and}_i$ or $\textit{xor}_i$ operator, respectively for $\textit{and}$- and $\textit{xor}$-decomposition. $s$ denotes the arity of the operator and must be equal to the number of subnodes (subfeatures) of the labelled node. The signature of $\lambda$ is thus $\lambda : N \rightarrow NT$, and $NT$ (node type) is a set of boolean operators. It contains operators $\textit{and}_1$, $\textit{and}_2$, $\textit{and}_3$, ... as well as operators $\textit{xor}_2$, $\textit{xor}_3$, ...$^6$ So, if $f_1$ is $\textit{and}$-decomposed into $f_2$ and $f_3$, we will have $\lambda(f_1) = \textit{and}_2$. A node like $f_1$ will thus be called an $\textit{and}$-node, and sometimes simply $\textit{and}$-node. We adopt similar terminological conventions for nodes labelled with other (types of) operators. Another convention is that terminal features, i.e., features which have no subfeature, are $\lambda$-labelled with $\textit{and}_0$.

The abstract syntax of optional features (those with a hollow circle on top in the concrete syntax) is a little trickier. For each such feature with label $L_1$ in the concrete diagram, the abstract diagram possesses two nodes, say $f_1$ and $f_1^?$. $f_1^?$ is introduced as an intermediate node between $f_1$ and those nodes which should have been its supernodes, had it not been optional. The only (direct) subnode of $f_1^?$ is $f_1$ and $\lambda(f_1^?) = \textit{opt}_1$. $\textit{opt}_1$ is the boolean operator that always returns TRUE. This way to define the abstract syntax of optional nodes came after noticing that they actually played a role similar to the $\textit{and}$- and $\textit{xor}$-decomposition, except for the fact that this kind of operator only acts upon one subnode.

An important concept we introduce in $L_{OFD}$ and which we also need in the generalised abstract syntax $L_{FFD}$ (see Section 4.2.1) is the concept of primitive node (or primitive feature). As also recognised by other authors [36, 21], there is currently no agreement on the following question: are all features equally relevant to define the set of possible products that the FD stands for? Another way to state the question is: are all the features in an FD relevant to distinguish two products, or is there a subset of the features that is relevant and one that is not? Actually, this question primarily addresses semantics, but has consequences for the syntax.

For example, in Fig. 2, one could question whether the (absence or presence of the) feature Measures is useful to describe a product, or if (the absence or presence of) its subfeatures, L/Km and Gallon/mile, suffice(s). Since there was no agreement in the literature, we adopted a neutral formalisation. Our solution accounts for the fact that the modeller can consider only part of the features as relevant. Although there is no construct in the concrete syntax (neither for OFD, nor any other FD language we know), we need to intro-

---

$^3$In Fig. 9, a graphical representation is given for the abstract syntax. This is by no means a proposal for a new concrete syntax. It just serves to illustrate the formal definition.

$^6$In addition, $NT$ also contains $\textit{and}_0$ and $\textit{opt}_1$, explained further in the section.
duce one in the abstract syntax, namely a subset \( P \) of \( N \) \((P \subseteq N)\). We will see the impact of \( P \) when we address the semantics of OFD. Finally, although we leave it to the modeller to determine \( P \), we reasonably expect that \( P \) (1) contains terminal features and (2) does not contain opt-nodes. But we need not impose these rules.

The last part of OFD’s concrete notation that \( \mathcal{L}_{OFD} \) should account for is the textual constraint language, called “composition rules” \((\text{CR})\). In the concrete diagram, the language is used to specify a (possibly empty) set of rules, usually located at the bottom of the diagram. In the abstract syntax, we call the set of rules \( \Phi \) and define it as a set of words obeying the following production rule: \( \text{CR} ::= f_1(\text{requires} | \text{mutex})f_2 \), where \( f_1, f_2 \in P \).

**Definition 3.1 (Original Feature Diagram)** An OFD \( d \in \mathcal{L}_{OFD} \) is a tuple \((N,r,\lambda,\text{DE},\Phi)\) where:

- \( N \) is its set of nodes;
- \( P \subseteq N \) is its set of primitive nodes;
- \( r \in N \) is the root;
- \( \lambda : N \rightarrow NT \) labels each node with an operator from \( NT \), where \( NT = \text{and} \cup \text{xor} \cup \text{opt}_1 \), i.e. a set of boolean functions (operators) where:
  - and is the set of operators \( (s \in N) \)
    - that return \( \text{TRUE} \) iff all their \( s \) arguments are \( \text{TRUE} \);
  - xor is the set of operators \( (s \in N \setminus \{0,1\}) \)
    - that return \( \text{TRUE} \) iff exactly one of their \( s \) arguments is \( \text{TRUE} \);
  - \( \text{opt}_1 \) is the operator that returns \( \text{TRUE} \);
- \( \Phi \subseteq \text{CR} \) are the textual constraints.

Furthermore, \( d \) must satisfy the following well-formedness rules:

1. Only \( r \) has no parent: \( \forall n \in N.(\exists n' \in N.n \rightarrow n' \land n = n') \Rightarrow n = r. \)
2. \( \text{DE} \) is acyclic: \( \forall n_1, \ldots, n_k \in N.n_1 \rightarrow \ldots \rightarrow n_k \rightarrow n_1. \)
3. Node operators are of adequate arities: \( \forall n \in N.\lambda(n) = \text{op}_k \land k = \#((n,n')|n \rightarrow n') \)
4. Terminal features are \( \text{and}_0 \)-labelled: \( \forall n \in N.(\exists n' \in N.n \rightarrow n' \land \lambda(n) = \text{and}_0) \)

Note that the first two well-formedness rules above should be enforced at the level of the concrete syntax (e.g., by a graphical OFD editor), whereas the last two rules should be guaranteed when moving from the concrete to the abstract syntax, and the modeller should not care about them.
The semantic domain of OFD ($\mathcal{S}_{\text{OFD}} = PL$) In the surveyed literature, there seems to be an agreement that FDs are meant to represent sets of products, and each product is seen as a combination of features. These tenets were present from the beginning in OFT [4] and were adopted without much controversy in its extensions, including OFD. In particular, none of the surveyed languages attempted to further define the “contents” of a feature beyond its name (viz., the labels appearing in the nodes of the FD), except for some recent work [22] (see Section 7.2). We published the first formalisation of a semantic domain specifically devoted to FDs in [6]. In this semantic domain, named $PL$, the atomic building blocks are features (nodes), a bit in the same way that propositions are the atomic building blocks in the semantic domain of propositional logic (see, e.g., [46]). However, we want to leave the flexibility to the modellers to decide which features are relevant for them to discriminate products, so we use $P$ instead of $N$. Definition 3.2 presents mathematically the notions of product and product line, relying on the more general notion of configuration.

Definition 3.2 (Configuration, Product, Product Line) A configuration is a set of features/nodes, i.e., any element of $\mathcal{P}N$.

- A product is a configuration that contains only primitive features, i.e., any element of $\mathcal{P}P$.
- A product line is a set of products, i.e., any element of $PL = \mathcal{P}P$.

Fig. 10 gives an illustration of this. Like a configuration, a product, say $c$, is a combination (i.e. a set) of features (nodes). In this case, $c$ is the set \{$f_1, f_3, f_4, f_5, f_6, f_7$\}. A product line, e.g. $pl$, is a set of products. Here, $pl$ is a set of 3 products: \{$\{f_3, f_6\}, \{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}, \{f_1, f_4, f_5, f_7\}$\}.

Recently, in other formalisation proposals, some authors [19, 20, 21, 22, 23, 25] have chosen semantic domains different from $PL$, for example using lists instead of sets [19]. How to compare $PL$ with other semantic domains will be discussed in Section 3.3 but the cause usually turns out to be an implementation bias. For the time being, and as it was anticipated in Section 3.1, an important observation is that $PL$, as all semantic domains, is indeed insufficient to describe semantics: it does say what is a product line, but it fails to say which products pertain to the product line a given OFD stands for. This is the role of the semantic function.

The semantic function of OFD ($M_{\text{OFD}} : OFD \rightarrow PL$) In Fig. 11, we depict $M_{\text{OFD}}$, the semantic function of OFD. To every diagram $d$, it assigns a $PL$, noted $M_{\text{OFD}}(d)$. $M_{\text{OFD}}(d)$ is more formally described in Definitions 3.3 and 3.4. Definition 3.3 indicates which set of products is returned by $M_{\text{OFD}}(d)$: the set of the configurations (combinations of features) which are valid wrt $d$, restricted to their primitive features.

Definition 3.3 (Semantic function) The semantics of an OFD $d$ is a product line (Definition 3.2) consisting of the products of $d$, i.e. its valid configurations.

- (Definition 3.4) restricted to primitive features/nodes: $M_{\text{OFD}}(d) = \{c' | c \in d \land c' = c \cap P\}$

Definition 3.4 provides four clear and compact rules telling what in OFD is a valid configuration wrt $d$. The fact that a configuration $c$ is valid wrt $d$ is noted $c \models d$.

Definition 3.4 (Valid configuration) A configuration $c \in \mathcal{P}N$ is valid for a $d \in \mathcal{L}_{\text{OFD}}$, noted $c \models d$, iff:

1. The root is in: $r \in c$

2. The meaning of nodes is satisfied: If a node $n \in c$, and $n$ has sons $s_1, \ldots, s_k$ and $\lambda(n) = op_n$, then $op_n(s_1 \in c, \ldots, s_k \in c)$ must evaluate to $\text{TRUE}$.

3. The configuration must satisfy all textual constraints: $\forall \phi \in \Phi, c \models \phi$, where $m \models \phi$ means that we replace each node name $n$ in $\phi$ by the truth value of $n \in c$, evaluate $\phi$ and get $\text{TRUE}$. For instance:

- if $\phi$ is a CR constraint of the form $f_1 \text{ requires } f_2$, we say that $m \models \phi$ when $f_1 \in c$ $\Rightarrow (f_2 \in c)$ evaluates to $\text{TRUE}$;

- if $\phi$ is a CR constraint of the form $f_1 \text{ mutex } f_2$, we say that $m \models \phi$ when and not $f_1, f_2 \in c$ evaluates to $\text{FALSE}$.

4. If $s$ is in the configuration and $s$ is not the root, one of its parents $n$, called its justification, must be too: $\forall s \in N, s \in c$ $\land s \neq r$ : $\exists n \in N : n \in c \land n \rightarrow s$. 
When $M_{OFD}[d]$ returns an empty set of products, i.e. the empty $PL$, it means that $d$ is non-satisfiable (or, inconsistent). In Fig. 11, this is the case for $d_3$. This happens when there is no product combination that can satisfy the constraints in $d$. Checking consistency, as well as many other tasks, can usually not be performed efficiently just by processing syntax, nor by letting the modeller inspect the diagram. Hence, the utility of defining the semantics in a way that enables faithful implementation into a computer programme automating time-consuming and error-prone tasks.

![Figure 11. OFD’s semantic function: $M_{OFD}$](image)

We now take a closer look at the product validity rules of Definition 3.4. The application of the rules is illustrated in Fig. 12. We assume that all features in the OFD are primitive, except for $f_9$ which was generated to account for an optional feature in the concrete syntax.

1. The first rule imposes that the root ($r = f_1$) appears in every valid product. Hence, the product $\{f_2, f_3, f_4, f_7, f_9\}$, for instance, cannot be part of it.

2. The second rule describes the semantics of the boolean operators coming from the decomposition links and from the optional features. This rule relies on the semantics of the boolean operator $opt_1$ as well as the operators in $\text{and}$ and $\text{xor}$. Their semantics has been recalled earlier in this section. In the example, the rule is applied to discard the product $\{f_1, f_2, f_3, f_5, f_6, f_9\}$, say $c$, because $f_3$ appears in it together with more than one node among $f_5$, $f_6$ and $f_7$. Indeed, $f_3$ is labelled with $\text{xor}_3$ and has 3 sons: $f_5$, $f_6$ and $f_7$. In $c$, $\text{xor}_3(f_5 \in c, f_6 \in c, f_7 \in c)$ would then evaluate to FALSE.

3. The third rule is similar in spirit to the former, except that it deals with the operators (requires and mutex) appearing in $\Phi$, the CR constraints accompanying the graphical part of the OFD. When applied to the example, the rule interprets the CR $f_4$ requires $f_9$ by checking the truth value of $(f_4 \in c) \Rightarrow (f_9 \in c)$, which in the case of the product $c = \{f_1, f_2, f_3, f_4, f_7, f_9\}$ evaluates to FALSE.

4. The fourth and last rule is called the justification rule. It guarantees that, except for the root, a node cannot be present in a valid product without one of its parent nodes being present as well. It says "one of its parents" because OFDs are DAGs and a node can therefore have multiple parents. In the example, this rule discards the product $c = \{f_1, f_2, f_3, f_4, f_6, f_8\}$ because $f_8$ belongs to $c$ but $f_6$, its only parent, does not. The justification rule has been often overlooked in the literature. For example, in [24], a formal semantics of FD is proposed without such a rule. This leads to strongly counter-intuitive semantics. Without the justification rule, the OFD in Fig. 12 would accept, e.g., products $\{f_1, f_2, f_3, f_6\}$ or $\{f_1, f_2, f_3, f_4, f_6, f_8\}$ as part of its semantics. Justifications also explain the difference between decomposition through and-nodes and requires constraints: the presence of subfeature is justified by its and-parent, while requires give no justification.

Eventually, when all the rules in Definition 3.4 have been taken into account and when all the non-primitive features in the products have been removed according to Definition 3.3, we see that the semantics of the OFD in Fig. 12 comes unambiguously as the following product line, made of five valid products: $\{\{f_1, f_2, f_3, f_4, f_7, f_9\}, \{f_1, f_2, f_3, f_4, f_7, f_9\}, \{f_1, f_2, f_3, f_7\}, \{f_1, f_2, f_3, f_7\}, \{f_1, f_2, f_3, f_5\}\}.$

![Figure 12. OFD’s semantics: an example](image)

### 3.2. Comparison criteria

When languages are equipped with a formal semantics, as presented in Section 3.1, and when they...
share a common semantic domain (S), we can compare them in a rigorous yet natural way. Three criteria are commonly used:

- **expressiveness**: what can the language express? (Section 3.2.1)
- **embeddability** (also called naturalness, or macroeliminability): when translating a diagram to another language, can we keep its structure? (Section 3.2.2)
- **succinctness**: how big are the expressions of a same semantic object? (Section 3.2.3)

When the semantic domains are not identical, we have first to align them before applying these criteria (see Section 3.3). Also, in any case, we can look at the efficiency (complexity) of the reasoning algorithms (see Section 3.4)

### 3.2.1. Expressiveness

Expressiveness is commonly understood as what can be expressed in a language. For a formal language, we can be more specific: the expressiveness E of a language L is the part of its semantic domain (S) that it can express, i.e., the image of its syntactic domain (L) through its semantic function M. This is what Definition 3.5 says, and what Fig. 13 illustrates. The diagrams in the syntactic domain of the language X (L_X) have an image E(L_X), a subset of X’s semantic domain (S).

**Definition 3.5 (Expressiveness)** The expressiveness of a language L is the set \(E(L) = \{M(d) | d \in L\}\), also noted \(M[L]\). A language \(L_1\) is more expressive than a language \(L_2\) if \(E(L_1) \supset E(L_2)\). A language L with semantic domain S is expressively complete if \(E(L) = S\).

If S is the common semantic domain of several languages, say W, X, Y and Z (see Fig. 13), their respective expressiveness can be compared. In the example we illustrate a situation where, because of their respective definitions, no two languages have the same expressiveness. Also, \(L_Z\) is more expressive than \(L_Y\). This is written \(E(L_Z) \supset E(L_Y)\). The expressiveness of \(L_X\) and \(L_Y\) are disjoint: \(E(L_X) \cap E(L_Y) = \emptyset\). The expressiveness of \(L_X\) and \(L_Z\) overlap: \(E(L_X) \cap E(L_Z) \neq \emptyset\). In general, the relationships between the syntactic domains (disjoint, overlapping, equal) of several languages should be considered non-correlated with the relationships existing between their respective semantic domains. This is because the semantic functions can be very different from one language to another.

In Fig. 13, we also notice that \(E(L_W) = S\). In cases like this, when the image of \(L\) is the whole of \(S\), we say that \(L\) is expressively complete: the part of the semantic domain it can express is the semantic domain itself. Complete expressiveness is a major quality for a language. It ensures that it can express all the intended meanings.

![Figure 13. Comparing expressiveness](image)

The usual way to prove that a language \(L_2\) is at least as expressive as \(L_1\) is to provide a translation (Definition 3.6) from \(L_1\) to \(L_2\):

**Definition 3.6 (Translation)** A translation is a total function \(T : L_1 \rightarrow L_2\) that preserves semantics: \(M_2[T(d_1)] = M_1[d_1]\).

![Figure 14. Translation between expressively complete languages](image)

The results that we have obtained studying the expressiveness of FD languages are summarised in Section 5.1.

Since languages compete for expressiveness, it often happens that they reach the same maximal expressiveness. This is for instance the case for programming...
languages, that are almost all Turing-complete and can thus express the same computable functions. Consequently, we need finer criteria than expressiveness to compare these languages.

The idea is to study the properties of the translations between those languages:

- do they preserve structure?
- do they increase size?

The former property is addressed by the concept of embeddability (Section 3.2.2) whereas succinctness takes care of the second (Section 3.2.3).

3.2.2. Embeddability. When two languages have the same expressiveness, in theory, there is a translation between them. However, this translation might destroy the structure of the original diagram.

For textual languages, the requirement to preserve structure (i.e., embeddability) has been called macro-eliminability by [47] (inspired by [48]). Macro-eliminability relies on the assumption that the concerned textual languages have a context-free grammar, which then allows to define the translation in a compositional way.

Unfortunately, context-free syntax is not a realistic assumption when dealing with visual languages [26, 27]. In particular, there is no such syntax for DAG-shaped FDs. The compositional definition of the translation can thus not be applied as such. In [8], we have proposed a definition of graphical embeddability which generalises the definition of embeddability for context-free languages. We recall it here (see Definitions 3.7 and 3.8) in a simplified form:

**Definition 3.7 (Graphical embeddability)** A graphical language $L_1$ is embeddable into $L_2$ iff there is a graphical embedding (Definition 3.8) from $L_1$ to $L_2$.

**Definition 3.8 (Graphical embedding)** A graphical embedding is a translation (Definition 3.6) $\mathcal{T} : L_1 \to L_2$ that is node-controlled [49]: $\mathcal{T}$ 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 $L_2$, plus how the existing relations should be connected to nodes of this new subgraph.

The notion of a node-controlled translation [49] is illustrated in Fig. 19 and further discussed in Section 5.2.

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 diagrams and generates only a linear increase in size. This way, traceability between the two diagrams is greatly facilitated and tool interoperability is made more transparent. Furthermore, limiting the size of diagrams helps avoiding tractability issues for reasoning algorithms taking the diagrams as an input (see Section 3.4).

Embeddability can also exist between a language and a subset of itself. A language that is non-trivially self-embeddable [8] is called harmfully redundant (see Definition 3.9). This means that it is unnecessarily complex: all diagrams can be expressed in the simpler sub-language without loss of structure and with only a linear increase in size.

**Definition 3.9 (Harmful redundancy)** A language $L$ is harmfully redundant iff there is a construct $C$ in $L$ that has a graphical embedding in $L \setminus C$.

The embeddability results that we have obtained so far concerning FD languages are summarised in Section 5.2. However, linear translations are not always possible. In this case, the blow-up in the size of the diagram must be measured. This is achieved by examining succinctness.

3.2.3. Succinctness. For languages with same expressiveness, embeddability guarantees that their respective diagrams are (roughly) of the same size (since, by definition, there exists a linear translation between them). When equally expressive languages are not embeddable, succinctness (see Definition 3.10) allows to compare the size of their respective diagrams by computing the size of the diagrams before and after translation from one language to the other. Stated otherwise, succinctness measures the blow-up caused by a change of notation.

**Definition 3.10 (Succinctness)** Let $\mathcal{G}$ be a set of functions from $\mathbb{N} \to \mathbb{N}$. A language $L_1$ is $\mathcal{G}$-as succinct as $L_2$, noted $L_2 \leq \mathcal{G}(L_1)$, iff there is a translation $\mathcal{T} : L_1 \to L_2$ that is within $\mathcal{G}$: $\exists g \in \mathcal{G}, \forall n \in \mathbb{N}, \forall l_1 \in L_1, |l_1| \leq n \implies |\mathcal{T}(l_1)| \leq g(n)$. Common values for $\mathcal{G}$ are “identically” = $\{n\}$, “thrice” = $\{3n\}$, “linearly” = $O(n)$, “cubically” = $O(n^3)$, “exponentially” = $O(2^n)$. We will omit “identically”.

If $L_1$ is more succinct than $L_2$, this usually entails that $L_1$’s diagrams are likely to be more readable. Also, if one needs to translate from $L_1$ to $L_2$, succinctness will be an indicator of the difficulty to maintain traceability between the original and the generated diagram. Traceability of linear translations is easy (see Note 7).
Syntactic domains Semantic domains  Semantic functions
L1
L2
S1
S2
T AM1
M2
M2 !d2" = A(M1 !d1 ")=M2 !T(d1) "
M1!d1"d1

We then create a domain two domains to determine the information they share.

Comparing languages with different semantic domains is actually possible, but it requires preliminary work which is now explained.

Consider two languages with their syntactic domains L1 and L2 and two different semantic domains, respectively S1 and S2. Their semantic functions are M1 and M2. We must first compare intuitively the two domains to determine the information they share. We then create a domain S for this shared information and provide functions A1 : S1 → S and A2 : S2 → S, called abstractions. The purpose of these abstraction functions is to remove additional information and keep the “core” of the semantic domain, where we will perform the comparisons. For example, in [18], we used an abstraction to remove the ordering of features and products from SFD. However, the question of the relevance of this discarded information remains and should be studied carefully.

A simple but frequent case is illustrated in Fig. 15, where domain S1 contains more information than S2; we then take S2 as the common domain. An abstraction A removes from elements of S1 their supplementary information and maps them in S2. It then makes sense to look for quasi-translations T : L1 → L2 between their syntactic domains. They are translations for the abstracted semantics A ◦ M1, and can thus be used to compare languages for expressiveness, embeddability or succinctness. Hence, if we apply T to a diagram d1 in the syntactic domain L1 we will obtain a diagram d2 in the syntactic domain L2 with the same abstracted semantics. Semantically, if we apply the semantic function M1 to d1 and then the abstraction function A we will map on the same element of S2 as if we apply T to d1 and then M2: A(M1[[d1]]) = M2[[T(d1)]]

Figure 15. Abstracting a semantic domain

When applied to more than two languages, this method will create many semantic domains related by abstraction functions. The abstraction functions can be composed and will describe a category [50] of the semantic domains. At the syntactic level, the translations can also be composed to yield expressiveness and succinctness results. Similarly, the composition of embeddings yields an embedding.

Our current results obtained for languages that do not share semantic domains are summarised in Section 5.5.

3.4. Complexity

Among its many advantages, formalising a language enables to define rigorously a set of questions that can be asked about the diagrams written in this language. In computability and complexity theory, these questions are called decision problems. For example, in a formal languages that have a set as semantic domain, we can state precisely the satisfiability problem, that is, checking whether a diagram actually has a “non empty” semantics; this indicates whether a diagram is consistent or not. Satisfiability is naturally formalised as: given a diagram d, is M[[d]] = 0 true? But there are many other such questions. Once these problems are formalised, one can ask (1) whether answers (i.e. computable functions) exist at all to solve this problem (decidability) and (2), if so, what is their relative computational difficulty (complexity).

A typical measure of complexity is time complex-
ity, i.e., the number of steps that it takes to solve an instance of the problem as a function of the size of the input, using the most efficient algorithm [51]. Exploring the memory usage of the most efficient algorithm is called space complexity. Time and space are usually ranked into complexity classes, like NP-complete, PSPACE-complete, ... [51].

For example, worst-case execution time of satisfiability checking might grow linearly with the size of the diagrams in some FD languages whereas, in other languages, it may grow exponentially with it. In the latter case, we are likely to face tractability issues.

In our previous work [8], we have studied the complexity of a series of problems for the surveyed FD languages:

- **satisfiability**: given a diagram $d$, is $M[d] = \emptyset$ true?
- **equivalence**: given two diagrams $d_1$ and $d_2$, is $M[d_1] = M[d_2]$ true?
- **model-checking** (called **product-checking** for FDs): given a product $c$ and a diagram $d$, is $c \in M[d]$ true?
- **intersection**: compute a new diagram $d_3$ such that $M[d_3] = M[d_1] \cap M[d_2]$.
- **union**: compute a new diagram $d_3$ such that $M[d_3] = M[d_1] \cup M[d_2]$.
- **reduced product**: compute a new diagram $d_3$ such that $M[d_3] = \{ c_1 \cup c_2 | c_1 \in M[d_1], c_2 \in M[d_2] \}$.

These are classical problems for languages whose semantic domain is a set. Their relevance in the context of SPL requirements engineering is further elaborated in Section 5.4 and in [9]. Complexity results are important because they help evaluate the scalability of the tool support answering those questions for a given language. Formalisation of both the syntax and semantics is a necessary prerequisite to devise precise questions and make sure that they match intuition. Complexity results then give an indication about the worst case, and how to handle it. Heuristics taking into account the most usual cases can be added to the backbone algorithm, to obtain practical efficiency. Finally, we should note that although formality is required, comparing languages wrt complexity does not require that these languages have the same semantic domains.

4. A Comparison Method for FD languages

In order to compare FD languages $X_1, \ldots, X_n$ according to the criteria exposed in the previous section, we need to have formally defined languages, that is, for language $X_i$, we need to know $L_{X_i}$, $S_{X_i}$ and $M_{X_i}$. Furthermore, if we want to be able to compare expressiveness, embeddability and succinctness, we also need to have $S_{X_i} = S_{X_1} = \ldots = S_{X_n}$.

However, this ideal situation almost never occurs in practice. Most of the time, we have to cope with

- languages that have no formal semantics at all (this is the most frequent case which we addressed in [8, 9]),
- languages with a formal semantics but defined in quite a different way from what is advocated in Section 3.1,
- or languages with a formal semantics compliant with the recommendations of Section 3.1 but using a different semantic domain.

The overall process for comparing FD languages according to our formal criteria must thus be carried out in two steps:

1. make the languages suitable for comparison,
2. make the comparisons.

In Section 4.1, we offer a systematic approach to cope with the first step, in all different situations. In Section 4.2, we recall FFD, the main tool that we have used until now to formalise informal FD languages. How the criteria are concretely evaluated on FD languages (that is, step 2) is presented in Section 5, where we also present a summary of the results obtained so far.

4.1. Making languages suitable for comparison

Let us call $X$ the language we want to compare with the others ($Y_1, \ldots, Y_n$) which, we assume, are fully and clearly formalised according to [26, 27] and have identical semantic domains. We distinguish the three cases mentioned earlier.

4.1.1. Case 1: $X$ has no formal semantics. There are two alternatives:

- The first alternative is to define the syntax and semantics for each FD language individually following Harel and Rumpe’s principles (Section 3.1). That is, we define $X$ independently from $Y_1, \ldots, Y_n$. This is what we did in Section 3.1.2 where we formalised OFD as an example. This is also what we did in [6] where we formalised OFT.

\[\text{We have also started to address instances of the latter two cases recently in [18].}\]
• The second alternative is to make scale economies and define several languages at once. In [9], we observed that most of the FD languages largely share the same goals, the same constructs and, as we understood from the informal definitions, the same (FODA-inspired) semantics. For this reason, we proposed to define not one FD language but a family of related FD languages (see Fig. 16).

We defined a parametric abstract syntax, called FFD, in which parameters (see Section 4.2.1) correspond to variations in \( L_X, L_Y_1, \ldots, L_Y_n \). This definition follows, but slightly adapts, the principles of Section 3.1. The semantic domain (PL, see Section 4.2.2) and semantic function (see Section 4.2.3) are common to all FD variants, maximizing semantic reusability. With this method, we can write diagrams in a language of the FD family, but expressed in a form that does not comply with the operational semantics at once, then a generic approach, like in Case 1, could be applied too. However, the main difference between Case 2 and Case 1 is that, in Case 2, formalisation decisions are usually much more straightforward since they have already been made. However, they might be hard to dig out if they are coded in the operational semantics attached to some tool. Also, formalisations are not necessarily error-free, and errors can thus be discovered when re-formalising [18].

4.1.3. Case 3: \( X \) has a formal semantics with clear \( L_X, S_X \) and \( M_X \) but \( S_X \neq S_Y (i \in \{1, \ldots, n\}) \). The third and last case is when we have a clear and self-contained mathematical definition of \( L, S \) and \( M \) for all languages (either from the origin, or having previously gone through Case 1 or 2) but the semantic domains of the languages to be compared differ, so that they cannot be compared directly for expressiveness, embeddability and succinctness. In this case, we thus need to define a relation between the semantic domains. The way to proceed was already explained in Section 3.3.

4.2. FFD

Here, we briefly recall FFD, a generic construct we developed the formalisation (Case 1) or re-formalisation (Case 2) of FD languages.

4.2.1. Syntactic Domain \((L_{FFD})\). The definition of the generic syntactic domain of FFD \((L_{FFD})\) is a simple generalization of the syntactic domain of OFT (see [9]), OFD (see Section 3.1.2) and the other surveyed languages (see Fig. 4 and Fig. 5). FFD stands for “Free Feature Diagrams” to emphasise its reusability for defining FD languages. In order to cover all the

\[ \text{Even more if } W \text{ is also given a formal semantics in a similarly “indirect” way, just as } X. \]
FD languages being formalised, \( L_{FFD} \) was defined as a parametric construction. Its four parameters (GT, NT, GCT, TCL) correspond to the four variations we have identified in the abstract syntaxes of the languages. That is, we ignored concrete syntax variations such as depicted in Fig. 3.

The abstract syntax variations are as follows:

- The decomposition edges in a FD can form a tree or a DAG. For example, in OFT, they form a tree whereas in OFD, they can form a DAG. Although a DAG is more general than a tree, we should also take trees into account specifically. For this, we introduce the parameter \( GT: GT (\text{Graph Type}) \) is either DAG or TREE.

- The type of nodes in a FD may change. In OFT and OFD, the allowed node types (NT) are xor-nodes, and-nodes and opt1-nodes. or-node and card-node are also included in some other languages and xor-node is not present in others. This variation point corresponds to the parameter \( NT: NT \) is a set of boolean functions (operators). It is defined in the same way as in Section 3.1.2, except that it possibly includes more operators. Noteworthy is the \( card_{[..]} \) operator introduced to account for the EFD language [14, 15]. The operator returns TRUE iff at least \( i \) and at most \( j \) of its \( s \) arguments are TRUE. Also, we should note that it is in principle possible to add more node types by defining their (commutative) boolean operators, in case a new language using a new form of decomposition appears.

- The type of graphical constraints may change. In OFT and OFD, there are no graphical constraints, but most other languages offer this possibility. This variation point corresponds to the parameter \( GCT: GCT (\text{Graphical Constraint Type}) \) is a set of binary boolean operators. E.g.: Requires (⇒) or Mutex (\( \Rightarrow \)).

- Finally, the type of textual constraints may change. This variation point corresponds to the parameter \( TCL (\text{Textual Constraint Language}) \). In general, it can be a subset of the language of boolean formulae where the predicates are the nodes of the FD. In our survey, all the investigated FD languages used CR (Composition Rules, see Section 3.1.2) except one (PFT [13]) which does not have a textual language. Nevertheless, we decided to accommodate for more languages since some authors have proposed to use more powerful languages, e.g., boolean logic [21].

The complete formalisation of \( L_{FFD} \) is available in [9]. Once we had it, it was easy to define all of the surveyed FD languages by simply providing the right parameters. As Table 1 shows, defining a FD language boils down to filling in a row of the table. In order to be complete, the transformation from the concrete FD languages to FFD should also be given. An example is given in Fig. 18 which illustrates the transformation from an EFD (concrete syntax) to an FFD (abstract syntax).

<table>
<thead>
<tr>
<th>Short Name</th>
<th>GT</th>
<th>NT</th>
<th>GCT</th>
<th>TCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFT [4]</td>
<td>TREE</td>
<td>and ∪ xor ∪ (opt1)</td>
<td>⊤</td>
<td>CR</td>
</tr>
<tr>
<td>OFD [10]</td>
<td>DAG</td>
<td>and ∪ xor ∪ (opt1)</td>
<td>⊤</td>
<td>CR</td>
</tr>
<tr>
<td>RFD [11]=VFDD [16]</td>
<td>DAG</td>
<td>and ∪ xor ∪ (opt1)</td>
<td>([\text{opt2}], \text{opt3})</td>
<td>CR</td>
</tr>
<tr>
<td>EFD [14, 15]</td>
<td>DAG</td>
<td>card ∪ (opt1)</td>
<td>⊤</td>
<td>CR</td>
</tr>
<tr>
<td>OGT [12]</td>
<td>TREE</td>
<td>and ∪ xor ∪ (opt1)</td>
<td>⊤</td>
<td>CR</td>
</tr>
<tr>
<td>PFT [13]</td>
<td>TREE</td>
<td>and ∪ xor ∪ (opt1)</td>
<td>([\text{opt2}], \text{opt3})</td>
<td>⊥</td>
</tr>
<tr>
<td>VFD [9]</td>
<td>DAG</td>
<td>card</td>
<td>⊤</td>
<td>⊥</td>
</tr>
</tbody>
</table>

Table 1. FD languages defined through FFD

Optional nodes The last adaptation concerns the optionality. In most FD languages, the nodes can be mandatory or optional, except for EFD [14] where the edges (not the nodes) are mandatory or optional. Both solutions are clearly relevant and therefore, for generality, we proposed specific decompositions for optional and mandatory nodes. Let us consider Fig. 17(a), a very basic FD. OFT and OFD [10] hint that it should be abstracted to (b), while EFD [14] to (c). Because we want to account for both, we take the finer decomposition (d), adding an \( opt1 \)-node \( f_1 \) under \( f_0 \). \( f_1 \) must have \( f_1 \) as a son, thus we also add a new edge (that can be omitted in the concrete syntax)\(^{12}\).

4.2.2. Semantic Domain \( (S_{FFD}) \). With FFD, we formalised a set of languages that were inspired from OFT (FODA FDs). In particular, we understood that they all shared the same semantic domain: \( PL = S_{FFD} \).

4.2.3. Semantic function \( (M_{FD}) \). The genericity of FFD added little complexity to the definition of the semantic function. For example, with respect to the definitions of \( S_{OFT} \) [4] and \( S_{OFD} \) (see Definitions 3.3 and 3.4), the only change is the addition of one more validity rule to account for graphical constraints which are present in some other languages, e.g., RFD [11] and EFD [14, 15] (see Definitions 4.1).

\(^{11}\)In [9], \( L_{FFD} \) is simply called FFD.

\(^{12}\)We treated mandatory nodes (filled circles) similarly. They can be seen as and1-nodes.
Definition 4.1 (Valid configuration (in FFD)) A configuration \( c \in \mathcal{P} \) is valid for a \( d \in \mathcal{L}_{\text{FFD}} \), noted \( c \models d \), iff:

1. \( c \) is valid according to Definition 3.4.

2. \( c \) satisfies all graphical constraints: \( \forall (n_1, op_2, n_2, \_ ) \in CE, \ op_2(n_1 \in c, n_2 \in c) \) must be true.

5. Language Evaluation Results

Here, we summarise the results obtained when we have applied our general methodology of comparative semantics to the FD languages. For the languages defined generically with FFD (see Section 4.2), the details and proofs can be found in [8]. The treatment of vDFD [19] is found in [18].

5.1. Expressiveness

For expressiveness, the distinction between languages that only admit trees and the ones that allow sharing of features by more than one father (DAGs or vDFD) turns out to be important. While tree-shaped languages are usually incomplete, OFD are already expressively complete without the constraints, and thus a fortiori RFD, EFD and VFD. vDFD are “almost” trees in that only terminal features (i.e. the leaves) can have multiple fathers (justifications), but this is sufficient to obtain expressive completeness.

In contrast, tree-shaped diagrams turned out to be expressively incomplete; in particular, OFT [4] cannot express disjunction. This justifies \( a \) posteriori the proposal [16] (VBSD) to add the or operator to OFT. But even so, we do not attain expressive completeness: this language is still unable to express \( \text{card}_3[2..2] \), the choice of two features among three. This justifies similarly the proposal [14] (EFD) to use the \text{card} operator. Both [16] and [14] also propose to allow DAGs: this extension alone, as we have seen, ensures expressive completeness. But we will see below better justifications in
terms of embeddability rather than succinctness.

When designing a FD language, it is thus essential to include more than trees to reach expressive completeness. Trees, however, are easier to understand and manipulate because they have a compositional semantics. vDFD [19] manage to have both advantages.

5.2. Embeddability

As explained in Section 3.2.2, a construct can be embedded (or macro-eliminated [47]) in another language if we can express it by a fixed schema.

An optional node \( n \) can be translated into a \( \text{xor}_2 \)-node, say \( n^? \) with two sons: the original node \( n \), and the TRUE node \( v \) which is an \( \text{and}_0 \)-node (i.e., with no son). As we in in see Fig.19, all incoming edges from parents of \( n \) are redirected to the new top node \( (n^?) \), and all outgoing edges to sons start from the node \( n \).

![Figure 19. Node-controlled translation (graphical embedding) of redundant optional node (in OFD concrete syntax)](image)

This supports our view that optionality is better treated as an operator.

<table>
<thead>
<tr>
<th>Instead of ...</th>
<th>write ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{opt}_1(f) )</td>
<td>( \text{card}_1<a href="f">0...1</a> )</td>
</tr>
<tr>
<td>( \text{xor}_m(f_1,\ldots,f_m) )</td>
<td>( \text{card}_m<a href="f_1,%5Cldots,f_m">1...1</a> )</td>
</tr>
<tr>
<td>( \text{and}_s(f_1,\ldots,f_s) )</td>
<td>( \text{card}_s<a href="f_1,%5Cldots,f_s">s...s</a> )</td>
</tr>
</tbody>
</table>

Table 2. Embedding COFD into VFD

We constructed an embedding from OFD without constraints (called COFD in [8]) to VFD, presented in Table 2. To save space, we use the textual form for the graphs. For instance, a node bearing a \( \text{xor}_m \) operator is translated to a node bearing a \( \text{card}_m[1...1] \) operator. In the next section, we will consider how those embeddings increase the size of the graph. Here we see that the VFD diagram resulting from the embedding of a COFD diagram has the same size. This result indicates that \( \text{card} \)-nodes proposed by [14] can embed all the other constructs. We proposed thus to use them systematically inside tools. We slightly differ from [14] that also uses optional edges: these can be modelled by \( \text{card}_1[0..1] \)-nodes and would be harmfully redundant. We proposed VFD to eliminate this slight drawback. Please note that this latter suggestion only concerns abstract syntax. In the concrete syntax, it is probably a good idea to keep optional nodes as this would decrease the size and visual complexity of the diagrams.

5.3. Succinctness

In [8], we also discovered translations that are not expressible as macros, because they depend on the number of neighbours. In that case, it is still interesting to compute the increase in size of the graph, as measured by succinctness. RFD and OFD are of similar succinctness, but, when translating VFD or EFD to OFD, we translate a \( \text{card}_n \)-node to a OFD graph of size \( O(k^2) \) [8]. A VFD of size \( O(k) \) could contain \( k \) \( \text{card}_1 \)-nodes, giving a cubic translation at the end:

**Theorem 5.1** \( \text{COFD} \leq O(\text{VFD}^3) \).

This result indicates again that \( \text{card} \)-nodes are a useful addition, but for different reasons than presented in [14].

5.4. Complexity

For FDs, solving all the standard problems of Section 3.4 turns out to be practically useful:

- **Equivalence** of two FD is needed whenever we want to compare two versions of a product line (for instance, after a refactoring). When they are not equivalent, the algorithm can produce a product showing their difference. For FD languages based on DAGs, and that allow non-primitive features, such as OFD, EFD, VFD, this problem is \( \Pi_1 \)-complete [8] (just above NP-complete [51]).

- **Satisfiability** is a fundamental property. It must be checked for the product line but also for the intermediate FDs produced during a staged configuration [42]. For FD languages based on DAGs, this problem is NP-complete.

- **Model-checking** (here, also called Product-checking) verifies whether a given product (made of primitive features) is in the product line of a FD. It is not as trivial as expected, because the selection performed for non-primitive nodes must be reconstructed. This gives an NP-complete problem. When recording this selection, the problem becomes linear again.

- **Union** is useful when teams validate in parallel the feature combinations that lead to an acceptable product, without feature interference. Their
work can be recorded in separate FDs. The union of these FDs will represent the validated products. For FD languages based on DAGs, this problem is solved in linear time, but the resulting FD should probably be simplified for readability.

- **Intersection** and **reduced product** are similar.

The complexity results show the role of non-primitive features: on one hand, it is useful to record them to accelerate the checking of products, but they should not become part of the semantics since this would restrict the expressiveness and strongly reduce the possible transformations of diagrams.

### 5.5. Relating semantic domains

A number of FD languages [19, 21, 20, 22, 24] are defined with their own semantic domain. We treated the language of [19], that we called vDFD, in detail in [18]. It is a textual language, containing three function symbols with a variable number of arguments: \texttt{all}, \texttt{one-of}, \texttt{more-of}. Its semantics is defined by rewriting rules within the ASF+SDF tool environment. The rule are to be applied in this order:

1. Normalisation rules to eliminate duplicate features and degenerate cases of the various constructs;
2. Variability rules to count the number of products allowed in a FD;
3. Expansion rules to expand a normalised feature expression into a disjunctive normal form;
4. Satisfaction rules to determine which feature expressions in disjunctive normal form satisfy the feature constraints.

We can thus take their disjunctive normal form as their semantics. It is a list of lists of primitive features. On the other hand, our semantic domain, \texttt{PL}, is sets of sets of primitive features (Section 3.1.2). In this case, it was quite easy to relate those two domains: we mapped each list to the set of elements it contains, removing the ordering information. 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.

Overall, although the semantics proposed in the literature are usually more detailed (see Section 7.2), we are most probably able to connect them to ours by abstractions. This seems to indicate that our semantics provides a commonly agreeable semantic basis.

### 6. Limitations

The main limitation of our work is explicit in its scope: the proposed method and its current results concern only formal language properties. In order not to over-interpret our conclusions, one should look at this work with a comprehensive view of model quality in mind. For example, with respect to the SEQUAL framework (recalled in Section 2), in order to be accurate and effective, we have deliberately chosen to address only part of the qualities required from a “good” modelling language:

- **Domain appropriateness** is addressed by looking at language expressiveness,
- **Comprehensibility appropriateness** is addressed by looking at embeddability and succinctness,
- **Technical actor interpretation appropriateness** is addressed by looking at complexity and also embeddability and succinctness.

Furthermore, we are conscious that our criteria cover only part of each of the three cited languages qualities (see below). Not trying to answer all the questions at once is a deliberate attempt to avoid answering none of them. In Section 7, we will further discuss several qualities that we have not addressed at all but that, we think, require similar attention. In particular, our approach evacuates concerns related to concrete syntax.

In contrast, a more holistic (quality-wise) attempt to compare feature modelling languages is reported in [45]. However, it is specific in the sense that it concerns the usage of feature diagrams in a particular company, for a given kind of project. This leads us to point out that the notion of a “good” modelling language is only relative to the context of use of the language. The priorities to be put on the expected qualities and criteria are very likely to be different from one company, or project, to another. This could lead us to relativise in some contexts the importance of formality. But we think that, for FD, formality is very likely to deliver more than it will cost since (1) the languages are relatively simple, (2) formality can be made largely transparent to the users (hidden being a graphical concrete syntax), (3) the automation possibilities are many [36, 9, 8], and (4) the information that feature models are used to convey is of critical importance for companies and therefore should suffer no ambiguity.

SEQUAL also helps identify another limitation of our contributions: for the moment, we have only looked at language quality, adopting a theoretical approach. A complementary work is to investigate models empirically. In Section 2, we emphasised the difficulty of
such an endeavour because of the limited availability of “real” FDs. Nevertheless, we do not consider it impossible and can certainly learn a lot by observing how practitioners create and use FDs. Although we have focussed on studying theoretical properties of FD languages, we need to recognise that no formal semantics, nor criteria, can ever guarantee by itself that the languages help capture the right information (neither too little, nor too much) about the domain being modelled. Only empirical research can help us give a convincing answer to this other aspect of domain appropriateness.

Moreover, even within the clearly confined scope of our research, we face some threats to validity. Our examination of formal language properties was not supported by tools (except for mere text editors). All the formalisation of, and reasoning (comparisons, demonstrations of theorems) on languages were carried out by humans. Therefore, we cannot guarantee that human errors, miss- or over-interpretations are completely absent from our results. In addition, we need to draw the reader’s attention on the fact that our formalisations were made only by considering the published documents, and without contacting the authors for clarifications, nor testing their tools. Some of our formalisation choices might therefore only be due to the way things were phrased in the surveyed papers, or to an erroneous understanding from our part.

Finally, concerning the results obtained until now by applying the method, we made clear that there are very relevant FD language proposals [42, 20, 23, 24, 22] on which we could not yet apply our method due to lack of time. This is a prioritary topic of future work.

7. Future Work

Ultimately, our research aims at accelerating the advent of a standard feature modelling language of an overall excellent quality, including (1) unambiguous and appropriate syntax and semantics, (2) efficient and proved correct reference algorithms. To move forward in this direction, much work is still needed, not only by us:

7.1. Validating the results

Having made explicit the semantics of several feature modelling languages, we need to confront them with their proponents and, more generally, the communities of researchers and practitioners working on the subject. Doing so, we will be able to correct possible misinterpretations (oversimplifications, arbitrary choices, etc.) we might have made, but also point out issues that were overlooked in informal definitions. We expect especially lively debates on the issue of edge-based vs. node-based semantics (see discussion on optional/mandatory nodes in Section 4.2.1) and on the notion of primitive feature/node (see Section 3.1.2).

Also, testing the tools that implement reasoning algorithms supporting the studied languages would also be a way to get a clearer understanding of their semantics.

7.2. Extending the results

Our method has been applied to most informal FD languages. A generic formalisation of all of them, FFD, was delivered and has helped gather precise results on them. Another informal language that could similarly benefit from formalisation is Orthogonal Variability Modelling (OVM) [3]. Also, some constructs found in the surveyed languages still have to be formalised, most notably, layers, generalisation and implementation links in FORM [10] and binding times in [16].

Concerning other formal languages, the comparative semantics of FFD with vFFD (van Deursen and Klint’s FDs) [18] was studied. More recently, several formalisation proposals for FDs appeared in the literature. Comparative semantics should be applied to them as well:

1. Batory [21] provides a translation of FDs 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 in his first semantics. The semantics of propositional formulae is closer to ours but differs in two respects: (1) decomposable features are not eliminated, and (2) the translation of operators by an equivalence leads to (we suspect) a counter-intuitive semantics, that differs from the first grammar-based semantics.

2. In [20], 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 semantic domain 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 [42], the authors provide an even richer syntax. The semantics of the latter is yet to be defined, but is intended
to be similar to [20].

3. Benavides et al. [22] propose to use constraint programming to reason on feature models. They extend the FD language of [20] with extra-functional features, attributes and relations between attributes. Subsequently, they describe tool-support based on mapping those FD to Constraint Satisfaction Problems (CSP).

4. Wang et al. [24] 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 configuration. Their semantics slightly differ from ours, since (1) they omit justifications and (2) they did not eliminate auxiliary symbols.

Here, we just gave very sketchy “first impressions” of existing formal definitions. Each of them now needs to be carefully studied according to the proposed method and criteria.

7.3. Applying the results

Two main applications of our current results can be considered:

- One of the main expected outcomes of our work is the development of efficient tool support for FD languages. Several tools with reasoning capabilities already exist [36] but, for most tasks, they have to face tough tractability issues. Our results (on formalisation and complexity, mainly) can help (1) verify the correctness of these algorithms, and (2) devise optimized algorithms.

- Our work suggests VFD as the language currently obtaining the best ranking according to the studied criteria. To make VFD usable, we still need to provide them with a concrete syntax\textsuperscript{13} (see below) and tool support. This is currently on-going work.

Of course, these applications will also be useful means to validate our results.

7.4. Extending the scope

As mentioned repeatedly in the paper, the scope of our research is limited to a restricted number of qualities and criteria. Studying other qualities and criteria is equally important. In particular, issues related to concrete syntax, ignored in the current paper, are complementary to our current investigations. In our survey of FD languages, we could observe that there were also diverging views on this issue. Despite our focus on semantics, we do not underestimate the impact of a good concrete syntax. In the end, this is the only thing most language users will actually see. Evaluation and improvement of concrete syntax is an area of research that possesses an important body of knowledge which is currently being structured [17, 52], and of which FDs could take advantage. An important topic is the one of reducing the visual complexity of real-size models [52].

Finally, an empirical approach to the quality of FD languages could complement and help validate our theoretical results. For example, complexity results, which are typically worst-case results, should be confronted with observations of the kinds (structure, size,...) of models that are actually used by practitioners. For instance, if it turns out that most real FDs are trees (instead of DAGs), then our complexity results should not be considered too pessimistically.

8. Conclusion

The bad news confirmed by this paper is that the current research on variability modelling is fragmented. The modelling of variability, and particularly the use of feature diagrams, can be a precious help in mastering the complexity of variability management in the context of software product lines engineering. At the requirements level, feature diagrams allow to represent in a concise way the commonalities and the variabilities of a whole family of products in terms of their features. Unfortunately, existing research in the field is characterised by a growing number of proposals and a lack of accurate comparisons between them. In particular, the formal underpinnings of feature diagrams need more careful attention.

The nocuous consequences of this situation are: (1) the difficulty for practitioners to choose appropriate feature modelling techniques, (2) an increased risk of ambiguity in models, (3) underdeveloped or inefficient tool support for reasoning on feature diagrams.

The good news that this paper delivers is that there are remedies to this situation. The ones that we propose are: (1) a global quality framework (e.g. Krogstie et al.’s SEQUAL) to serve as a roadmap for improving the quality of feature modelling techniques; (2) a set of formally defined criteria to assess the semantics-related qualities of feature diagram languages; (3) a systematic method to formalise these languages and make them ready for comparison and efficient tool automation; and (4) a first set of results obtained from the application of this systematic method on a substantial part of the feature modelling languages encountered in the literature.

\textsuperscript{13}One is proposed in Fig. 5 but was not the outcome of a profound reflection.
Although the road ahead is still quite long, we are confident that the community can take profit of our proposal. It could be used for example as part of an arsenal to elaborate a standard feature modelling language. This standard would not suffer from ambiguity, and its formal properties (among others) would be well known, allowing to devise proved correct efficient reference algorithms. A similar approach could also be transposed to cognate areas where existing modelling techniques face similar challenges. In particular, we think of goal modelling techniques.

References


