Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

Supporting Commonality-Based Analysis of Software Product Lines

Ruben Heradio Gil*[1], David Fernandez-Amoros†[1], Jose A. Cerrada Somolinos‡[1], and Jose A. Cerrada Somolinos§[1]

[1]ETS de Ingenieria Informatica,, Universidad Nacional de Educacion a Distancia, Madrid, Spain

Abstract

Software Product Line (SPL) engineering is a cost effective approach to developing families of similar products. Key to the success of this approach is to correctly scope the domain of the SPL, identifying the common and variable features of the products and the interdependencies between features. In this paper, we show how the commonality of a feature (i.e., the reuse ratio of the feature among the products) can be used to detect scope flaws in the early stages of development. SPL domains are usually modeled by means of feature diagrams following the FODA notation. We extend classical FODA trees with unrestricted cardinalities, and present an algorithm to compute the number of products modeled by a feature diagram and the commonality of the features. Finally, we compare the performance of our algorithm with two other approaches built on top of boolean logic SAT-solver technology such as cachet and relsat.

1 Introduction

Software Product Line (SPL) engineering is an efficient and cost-effective approach to developing portfolios of similar products [36]. The fundamental idea of the approach is to undertake the development of a set of products as a single, coherent development task. Products are built from a Core Asset Base (CAB), a collection of artifacts that have been designed specifically for use across the portfolio [12].

The domain of a SPL must be carefully scoped, identifying the common and variable features of its products and the interdependencies between features. In ill-scoped domains, relevant features may not be implemented, and implemented features may never be used, causing unnecessary complexity and both development and maintenance costs [14]. To avoid these serious problems, SPL domains are usually modeled by means of feature diagrams. *Commonality* is a key metric, that indicates the reuse ratio of a feature across the SPL. According to the *Software Engineering Institute* [11], the commonality C_F of a feature F can be computed by equation 1, where: $\|P_F\|$ is the number of products within the SPL that satisfy the feature and n is the total number of products of the SPL.

$$C_F = \frac{\|P_F\|}{n} \tag{1}$$

*rheradio@issi.uned.es
†david@lsi.uned.es
‡jcerrada@issi.uned.es
§jcerrada@issi.uned.es

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

The relevance of commonality to estimate the costs and benefits of developing and evolving a SPL has been recognized by a number of surveys [7], theses [5, 32] and economic models [11]. In other industries (e.g., the automotive industry), commonality is also considered a key metric in resolving the tradeoff between product similarity and distinctiveness in a family of products [42, 9].

Analyzing feature diagrams is an error-prone and tedious task, and it is unfeasible to carry it out with large-scale feature diagrams. As a result, the automated analysis of feature diagrams is an active area of research in the SPL community [7]. Existing proposals for computing commonality in SPLs translate feature diagrams into propositional logic formulas, that are processed by off-the-shelf tools, such as Boolean Satisfiability (SAT) solvers, general Constraint Satisfaction Problem (CSP) solvers and Binary Decision Diagrams (BDD). Nevertheless, this approach does not scale to real SPLs since it quickly falls into a combinatorial explosion [5, 32]. To overcome this problem, we propose an algorithm that computes commonality and works for large diagrams (its time complexity is just quadratic on the number of features included in the diagram).

Most notations for feature diagrams [14, 18, 27] and commercial SPL tools [43, 44] model dependencies between features by means of a tree structure. In addition, extra cross-tree interdependencies are written in propositional logic. Our algorithm is focused on the essential dependencies and does not consider cross-tree feature dependencies. There are a number of notations available for feature diagrams. In order to make our proposal as general as possible, the algorithm is specified for an abstract notation for feature diagrams, named Neutral Feature Tree (NFT), that works as a pivot language for most of the available notations.

The remainder of this paper is structured as follows. Section 2 introduces the notion of feature diagram and formally defines the abstract syntax and semantics of NFT. Section 3 sums up how to detect scope flaws in the domain analysis stage thanks to commonality. Section 4 presents the sketch of our algorithm, which is described in detail in appendix A. Section 5 describes the computational complexity of the algorithm. Section 6 experimentally evaluates our proposal, translating a set of feature diagrams to boolean logic and processing them with *cachet* [40] and *relsat* [3], two model counters based on SAT-solver technology, verifying empirically that our approach scales better. Section 7 summarizes related work to our proposal. Finally, section 8 presents the conclusions of our work.

2 Formalizing Feature Diagrams

The aim of this paper is to provide an efficient algorithm to calculate the commonality of the features modeled by a feature diagram. In order to make our proposal as general as possible, we should avoid limiting the algorithm to a particular notation for feature diagrams. This is a challenging issue, because there is a profusion of available notations. Since the first language was proposed by the FODA methodology in 1990 [27], a number of extensions and alternative languages have been devised to model variability in families of related systems:

1. As part of the following methods: FORM [28], FeatureRSEB [23], Generative Programming [14], PLUSS [18].
2. In the work of the following authors: Riebisch et al. [38], van Gurp et al. [46], van Deursen et al. [16], Gomaa [21], Pohl [36].
3. As part of the following tools: Gears [43] and pure::variants [44].

Fortunately, Schobbens et al. [41, 33] have demonstrated that most of the notations can be translated easily and efficiently into a pivot language called Varied Feature Diagram$^+$ (VFD$^+$).

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

VFD$^+$ diagrams are single-rooted directed acyclic graphs. However, our algorithm has been tuned to work with feature diagrams structured as trees. For that reason, we propose the usage of a VFD$^+$ subset named NFT, where diagrams are restricted to trees. This does not imply a loss of generality for the algorithm, since, as it will be shown in section 2.4, NFT and VFD$^+$ are fully equivalent (i.e., any VFD$^+$ model has an equivalent representation in NFT).

In this section, we provide a precise definition of NFT. As argued in [24, 22], the syntax and semantics of languages should be formally defined to avoid ambiguities and support the construction of automated reasoning tools, such as the algorithm we propose in this paper. Accordingly, section 2.1 outlines the main parts of a formal language, and sections 2.2 and 2.3 define the abstract syntax and semantics of NFT, respectively. We emphasize NFT is not meant as a user language, but only as a formal "back-end" language used to define our algorithm in a general way.

2.1 Anatomy of a Formal Language

According to Greenfield et al. [22], the anatomy of a formal language includes an *abstract syntax*, a *semantics* and one or more *concrete syntaxes*.

1. The abstract syntax of a language characterizes, in a abstract form, the kinds of elements that make up the language, and the rules for how those elements may be combined. All valid element combinations supported by an abstract syntax conform the *syntactic domain* $\mathcal{L}$ of a language.

2. The semantics of a language define its meaning. According to Harel et al. [24], a semantic definition consists of two parts: a *semantic domain* S and a *semantic mapping* $\mathcal{M}$ from the syntactic domain to the semantic domain. That is, $\mathcal{M} : \mathcal{L} \rightarrow S$.

3. A concrete syntax defines how the language elements appear in a concrete, human-usable form.

Following sections define NFT abstract syntax and semantics. Most notations for feature diagrams may be considered as concrete syntaxes or "views" of NFT.

2.2 Abstract Syntax of NFT

A NFT diagram $d \in \mathcal{L}_{\text{NFT}}$ is a tuple $(N, \Sigma, r, DE, \lambda, \phi)$, where:

1. N is the set of nodes of d, among r is the root. Nodes are meant to represent *features*. The idea of feature is of widespread usage in domain engineering and it has been defined as a "distinguishable characteristic of a concept (e.g., system, component and so on) that is relevant to some stakeholder of the concept" [14].

2. $\Sigma \subset N$ is the set of *terminal nodes* (i.e., the leaves of d).

3. $DE \subseteq N \times N$ is the set of *decomposition edges*; $(n_1, n_2) \in DE$ is alternatively denoted $n_1 \rightarrow n_2$. If $n_1 \rightarrow n_2$ then n_1 is the *parent* of n_2, and n_2 is a *child* of n_1.

4. $\lambda : (N - \Sigma) \rightarrow$ card labels each non-leaf node n with a card boolean operator. If n has children $n_1, ..., n_s$, $\text{card}_s[i..j](n_1, ..., n_s)$ evaluates to `true` if at least i and at most j of the s children of n evaluate to `true`. Regarding the card operator, the following points should be taken into account[1]:

[1] The same considerations are valid for VFD$^+$.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

(a) whereas many notations distinguish between *mandatory*, *optional*, *or* and *xor* dependencies, card operator generalizes these categories. For instance, Figure 1 depicts equivalences between the feature notation proposed by Czarnecki et al. [14] and NFT.

(b) whereas, in many notations, children nodes may have different types of dependencies on their parent, in NFT all children must have the same type of dependency. This apparent limitation can be easily overcome by introducing auxiliary nodes. For instance, Figure 2 depicts the equivalence between a feature model and a NFT diagram. Node A has three children and two types of dependencies: $A \rightarrow B$ is *mandatory* and $(A \rightarrow C, A \rightarrow D)$ is a *xor*-group. In the NFT diagram, the different types of dependencies are modeled by introducing the auxiliary node aux.

5. ϕ[2] are additional textual constraints written in propositional logic over any type of node ($\phi \in \mathbb{B}(N)$).

Additionally, d must satisfy the following constraints:

1. Only r has no parent: $\forall n \in N \cdot (\exists n' \in N \cdot n' \rightarrow n) \Leftrightarrow n \neq r$.
2. d is a tree. Therefore,
   (a) a node may have at most one parent:
   $\forall n \in N \cdot (\exists n', n'' \in N \cdot ((n' \rightarrow n) \wedge (n'' \rightarrow n) \Rightarrow n' = n''))$
   (b) DE is acyclic: $\nexists n_1, n_2 \ldots, n_k \in N \cdot n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_k \rightarrow n_1$.
3. card operators are of adequate arities:
   $\forall n \in N \cdot (\exists n' \in N \cdot n \rightarrow n') \Rightarrow (\lambda(n) = \text{card}_s) \wedge (s = \|\{(n, n') | (n, n') \in DE\}\|)$

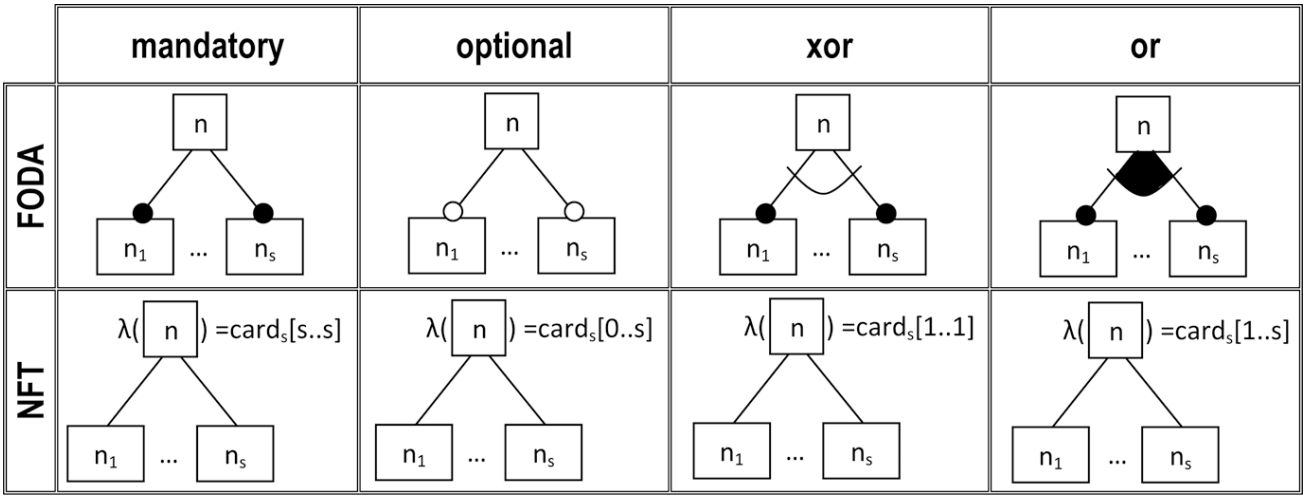

Figure 1: *card* operator generalizes *mandatory*, *optional*, *or* and *xor* dependencies

[2] Also named cross-tree constraints [5].

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

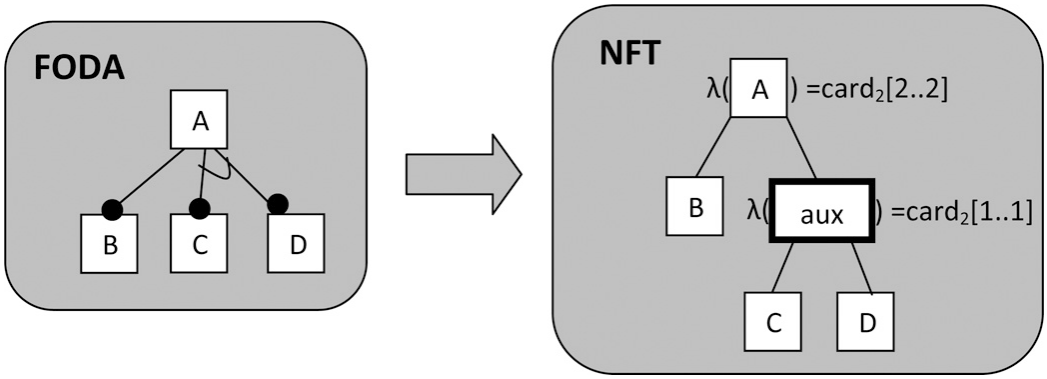

Figure 2: Different types of dependencies between a node and its children can be expressed in NFT by introducing auxiliary nodes

2.3 Semantics of NFT

Feature diagrams are meant to represent sets of products, and each product is seen as a combination of terminal features. Hence, the semantic domain of NFT is $\mathcal{P}(\mathcal{P}(\Sigma))$, i.e., a set of sets of terminal nodes. The semantic mapping of NFT ($\mathcal{M}_{\text{NFT}} : \mathcal{L}_{\text{NFT}} \rightarrow \mathcal{P}(\mathcal{P}(\Sigma))$) assigns a SPL to every feature diagram d, according to the next definitions:

1. A *configuration* is a set of features, that is, any element of $\mathcal{P}(N)$. A configuration c is valid for a $d \in \mathcal{L}_{\text{NFT}}$, iff:
   (a) The root is in c ($r \in c$).
   (b) The boolean value associated to the root is `true`. Given a configuration, any node of a diagram has associated a boolean value according to the following rules:
      i. A terminal node $t \in \Sigma$ evaluates to `true` if it is included in the configuration ($t \in c$), else evaluates to `false`.
      ii. A non-terminal node $n \in (N - \Sigma)$ is labeled with a card operator. If n has children $n_1, ..., n_s$, $\text{card}_s[i..j](n_1, ..., n_s)$ evaluates to `true` if at least i and at most j of the s children of n evaluate to `true`.
   (c) The configuration must satisfy all textual constraints ϕ.
   (d) If a non-root node is in the configuration, its parent must be too.
2. A *product* p, named by a valid configuration c, is the set of terminal features of c: $p = c \cap \Sigma$.
3. The SPL represented by $d \in \mathcal{L}_{\text{NFT}}$ consists of the products named by its valid configurations (SPL $\in \mathcal{P}(\mathcal{P}(\Sigma))$).

2.4 Equivalence between NFT and VFD$^+$

NFT differentiates from VFD$^+$ in the following points:

1. **Terminal nodes vs. primitive nodes**. As noted by some authors [1], there is currently no agreement on the following question: are all features equally relevant to define the set of possible products

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

that a feature diagram stands for? In VFD$^+$, Schobbens et al. have adopted a neutral formalization: the modeler is responsible for specifying which nodes represent features that will influence the final product (the primitive nodes P) and which nodes are just used for decomposition ($N - P$). P. Schobbens points that primitive nodes are not necessarily equivalent to leaves, though it is the most common case. However, a primitive node $p \in P$, labeled with $\text{card}_s[i..j](n_1, ..., n_s)$, can always become a leaf ($p \in \Sigma$) according to the following transformation $\mathcal{T}_{P \to \Sigma}$:

(a) p is substituted by an auxiliary node aux_1.

(b) the children of aux_1 are p and a new auxiliary node aux_2.

(c) aux_1 is labeled with $\text{card}_2[2..2](p, \text{aux}_2)$.

(d) p becomes a leaf. aux_2's children are the former children of p.

(e) aux_2 is labeled with the former $\text{card}_s[i..j](n_1, ..., n_s)$ of p.

Figure 3 depicts the conversion of a primitive non-leaf node B into a leaf node.

2. **Directed acyclic graphs vs. trees**. Whereas diagrams are trees in NFT, in VFD$^+$ they are directed acyclic graphs. Therefore, a node n with s parents $(n_1, ..., n_s)$ can be translated into a node n with one parent n_1 according to the following transformation $\mathcal{T}_{\text{directed acyclic graph} \to \text{tree}}$:

(a) $s - 1$ auxiliary nodes $\text{aux}_2, ..., \text{aux}_s$ are added to the diagram.

(b) edges $n_2 \to n, ..., n_s \to n$ are replaced by new edges $n_2 \to \text{aux}_2, ..., n_s \to \text{aux}_s$.

(c) Batory [1] demonstrated how to translate any edge $a \to b$ into a propositional logic formula $\phi_{a,b}$. Using Batory's equivalences, implicit edges $\text{aux}_2 \to n, ..., \text{aux}_s \to n$ are converted into textual constraints $\phi_{\text{aux}_2,n} ... \phi_{\text{aux}_s,n}$ and are added to ϕ ($\phi' \equiv \phi \wedge \phi_{\text{aux}_2,n} \wedge ... \wedge \phi_{\text{aux}_s,n}$).

Figure 4 depicts the conversion of a node D with two parents B and C into a node with a single parent.

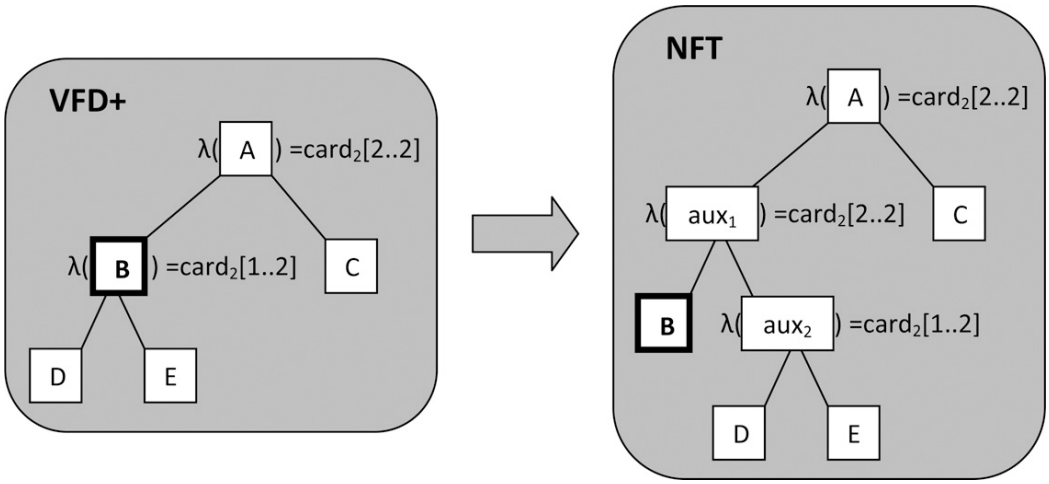

Figure 3: Any primitive non-leaf node can be converted into a leaf node by using $\mathcal{T}_{P \to \Sigma}$

In order to identify when a transformation on a diagram keeps (1) the diagram semantics and (2) the diagram structure, Schobbens [41] proposes the following definition of *graphical embedding*: "a translation

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

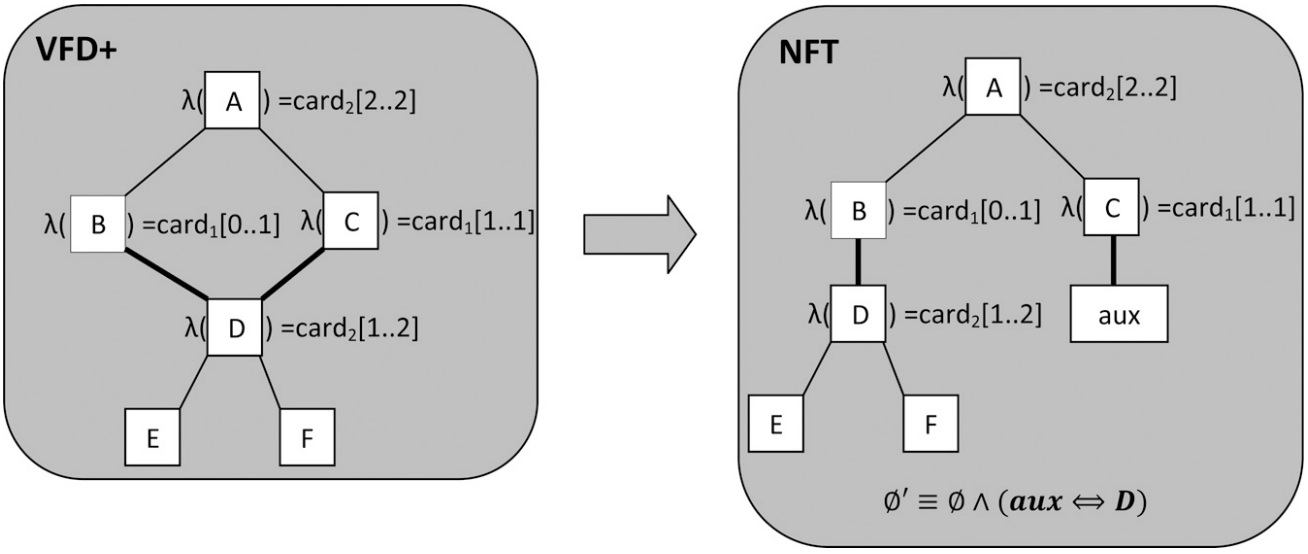

Figure 4: Any DAG can be converted into a tree by using $\mathcal{T}_{\text{DAG}\to\text{tree}}$

$\mathcal{T} : \mathcal{L} \to \mathcal{L}'$ that preserves the semantics of $\mathcal{L}$ and is node-controlled, i.e., $\mathcal{T}$ is expressed as a set of rules of the form $d \to d'$, where d is a diagram containing a defined node or edge n, and all possible connections with this node or edge. Its translation d' is a subgraph in $\mathcal{L}'$, plus how the existing relations should be connected to nodes of this new subgraph". According to this definition, $\mathcal{T}_{P\to\Sigma}$ and $\mathcal{T}_{\text{directed acyclic graph}\to\text{tree}}$ are graphical embeddings that guarantee the equivalency between NFT and VFD$^+$.

2.5 An example of NFT diagram

Figure 5 is a NFT representation of the FAME-DBMS model proposed by Kastner et al. [29]. FAME-DBMS is a database SPL prototype designed specifically for small embedded systems. To customize FAME-DBMS, we can choose between different operating systems, between a persistent and an in-memory database, and between different memory allocation mechanisms and paging strategies. Furthermore, index support using a B$^+$-tree is optional, and so are debugging and logging. Finally, it is possible to select from three optional operations get, put, and delete.

The example has 22 features, 13 of which are terminal. Terminal features are represented by simple boxes and non-terminal ones by double boxes. The decomposition edges are hierarchically depicted, e.g., the line between Storage and Unindexed means Storage → Unindexed. λ labels are abbreviated by the [low..high] notation, e.g., [0..3] in box API means $\text{card}_3[0..3](\text{Delete}, \text{Put}, \text{Get})$. Finally, the example does not include any cross-tree constraint ϕ.

There are 200 valid configurations for Figure 5. Table 1 includes one of them and its corresponding product:

3 Commonality-Based Analysis of a Product Line

This section presents how commonality can help to detect ill-scoped domains. Subsection 3.1 is focused on the identification of problematic features which are rarely reused or not reused enough to produce

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

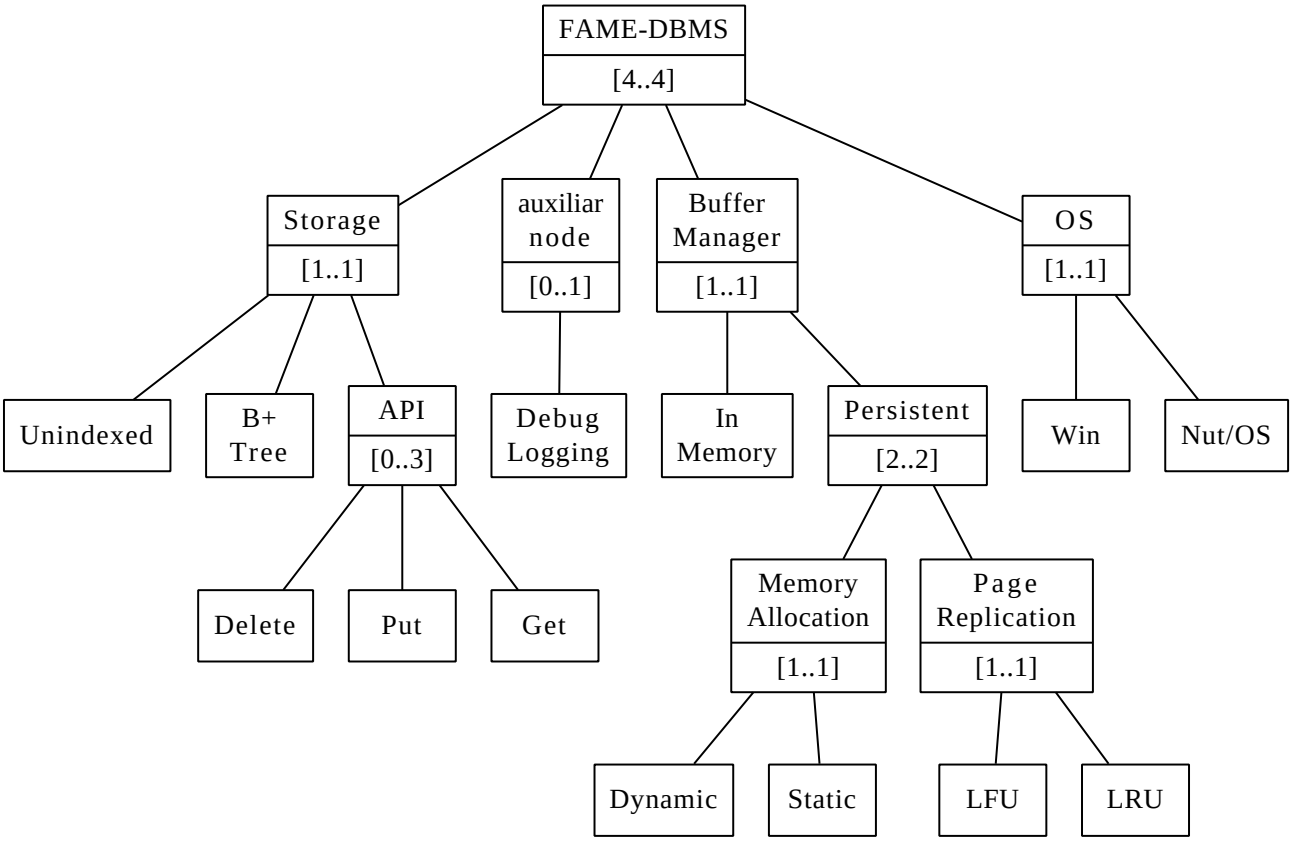

Figure 5: NFT representation of the FAME-DBMS SPL

configuration	FAME-DBMS, Storage, Unindexed, Buffer Manager, In Memory, OS, Win
product	Unindexed, In Memory, Win

Table 1: Example of valid configuration / product for the feature diagram in Figure 5

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

significant payoff. Then, subsection 3.2 shifts to a more abstract level of detail, showing how commonality may be used to get an overall idea of the amount of reuse achieved with the SPL.

3.1 Detection of Problematic Features

Figure 6 summarizes the commonalities of the terminal features of the FAME-DBMS feature diagram. For instance, whereas feature Unindexed is included in 10% of the products, Debug Logging appears in 50%. Because of Figure 6, the domain engineer can quickly get a feel for the essential features of the SPL.

Economic models for SPLs proposed in [10, 30, 35, 37, 47] use the following measures to quantify the effort of reusing an asset and the effort of making it easy to reuse:

- The Relative Cost of Reuse (RCR) represents the proportion of the effort that it takes to reuse software compared to the cost normally incurred to develop it for one-time use. For instance, a feature has RCR = 0.2 if it can be reused for only 20% of the cost of implementing it.
- The Relative Cost of Writing for Reuse (RCWR) represents the proportion of the effort that it takes to develop reusable software compared to the cost of writing it for one-time use. For instance, if it costs an additional 50% effort to create a feature for reuse (i.e., it is necessary a more generic design, additional documentation...) then RCWR = 1.5.

Poulin [37] defines a metric called *payoff threshold*, which shows how many times a feature has to be reused before the investment made to develop the feature is recovered. The payoff threshold of a feature F is calculated by equation 2.

$$\text{Payoff Threshold}_F = \frac{\text{RCWR}_F}{1 - \text{RCR}_F} \tag{2}$$

If C_F is the commonality of feature F and n is the total number of products of the SPL, F causes a scope flaw when equation 3 holds.

$$(C_F \cdot n) < \text{Payoff Threshold}_F \tag{3}$$

In addition, commonality supports the detection of *core features*. A feature F is a core feature if it is part of all the products, i.e., $C_F = 1$. Core features are the most relevant features of a SPL since they are supposed to appear in all products. Hence, detecting them in a feature diagram is useful to determine which features should be developed first [45] or to decide which features should be part of the core architecture of the SPL [34].

3.2 Global Analysis of the SPL Scope

Frequently, a domain engineer has to deal with feature diagrams that contain so many features that it is necessary to condense the data for easy comprehension of the general characteristics of a SPL. We propose to use a commonality histogram to graphically represent the degree of reuse of the features. For instance, Figure 7 shows that most of the terminal features of the FAME-DBMS feature diagram are reused by the (20%-40%][3] of the products, i.e., there are 3 features with commonality between [0-0.2], 7 between (0.2-0.4] and 3 between (0.4-0.6].

Many managers favor an incremental approach to product line adoption, one that first tackles areas of highest and most readily available commonality, earning payback early in the adoption cycle. Under this

[3] In Figure 7 we have adopted the right-end inclusion convention, which stipulates that a class interval contains its right-end but not its left-end boundary point.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

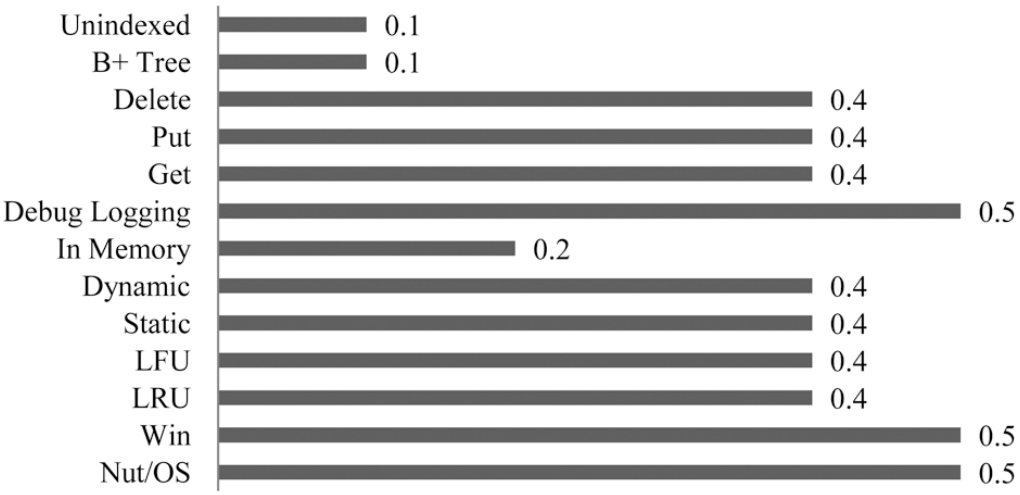

Figure 6: Commonality histogram for the FAME-DBMS terminal features

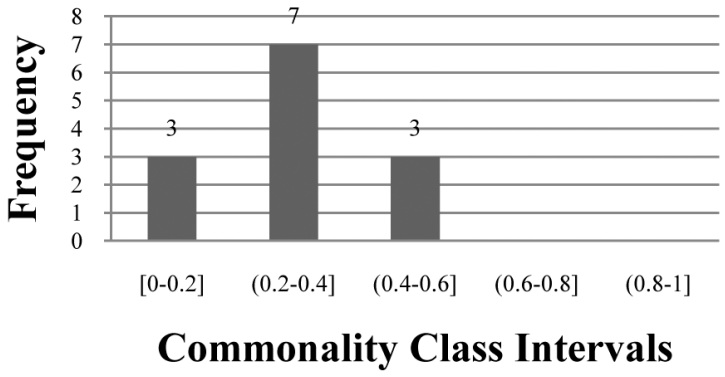

Figure 7: Commonality histogram for the FAME-DBMS terminal features

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

approach, the organization plans from the beginning to develop a SPL. It develops part of the Core Asset Base (CAB), including the architecture, and then builds one or more products. In the next increment, it develops a portion of the rest of the CAB and builds additional products. Over time, it evolves more of the CAB in parallel with new product development. In order to quantify the reuse improvement achieved in each development increment, Cohen [13] proposes a measure called Degree of Reuse (DOR), which is the portion of a complete product that is made reusing the CAB; e.g., a DOR of 0.25 means that the core assets are used in the development of 25% of the software of a typical product.

Cohen does not provide an equation to calculate DOR accurately. For that reason, we propose equation 4 to compute DOR, which relies on the commonality concept. The following points sum up the meaning of the equation parameters:

1. C_F is the commonality of F and n the total number of products of the SPL.
2. Size_F is the size of the software that implements the feature F (a number of techniques to estimate software size are presented in [19]).
3. The dividend is the size of all the software encompassed by the SPL, i.e., the size of all the products. Such size is calculated indirectly multiplying the size of the software that implements every feature (i.e., Size_F) by the number of times that software is reused (i.e., $C_F \times n$).
4. The numerator is the size of the all the software that is made by reusing core assets.

$$\text{DOR} = \frac{\sum_{F \in \text{CAB}} (\text{Size}_F \times C_F \times n)}{\sum_F (\text{Size}_F \times C_F \times n)} = \frac{\sum_{F \in \text{CAB}} (\text{Size}_F \times C_F)}{\sum_F (\text{Size}_F \times C_F)} \tag{4}$$

For instance, Table 2 summarizes a possible development state for the FAME-DBMS example. The table includes the commonality for each feature, the size measured in thousands of Source Lines of Code (KSLOC) of the software that implements each feature, and if a feature is part of the CAB (i.e., whether it has been adapted for reuse throughout the SPL). For example, feature *Unindexed* has commonality 0.1, estimated size of 1 KSLOC and belongs to the CAB. Thanks to equation 4, we can conclude that a reasonable level of reuse has been achieved in the SPL, since the core assets are used in the development of 79.6% of a typical product:

$$\text{DOR} = \frac{1 \cdot 0.1 + 1 \cdot 0.4 + ... + 2 \cdot 0.5}{1 \cdot 0.1 + 2 \cdot 0.1 + ... + 2 \cdot 0.5} = \frac{4.7}{5.9} = 0.796$$

Although there are well documented examples of cost reduction, shorter development times, and quality improvement achieved by introducing the SPL paradigm in industry [17], the approach is not always the best economic choice for developing a family of related products. For instance, the products may be prohibitively dissimilar from each other. In such cases, it is not worthwhile to develop and maintain a CAB (i.e., if an asset is not going to be reused, it makes no sense to invest in making it easily reusable). SIMPLE defines a measure, named *homogeneity*, that characterizes how similar the products are. The metric varies from 0 to 1, where 0 indicates that the products are all totally unique and 1 indicates that there is only one product.

In the presence of unknown feature commonalities, SIMPLE proposes to calculate homogeneity by means of equation 5, where: $\|F_U\|$ is the number of features unique to one product, and t is the total number of features. Unfortunately, this measure may produce erroneous results in some scenarios. For instance, consider a SPL with 100 features, where every feature has commonality of 0.02 (i.e., each feature is reused by only 2 products); although the SPL is quite heterogeneous, equation 5 says that the SPL is completely homogeneus (i.e., homogeneity $= 1 - \frac{0}{100} = 1$).

$$\text{Homogeneity} = 1 - \frac{\|F_U\|}{t} \tag{5}$$

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

F	C_F	Size$_F$ (KSLOC)	is F part of the CAB?
Unindexed	0.1	1	✓
B$^+$ Tree	0.1	2	✗
Delete	0.4	1	✓
Put	0.4	1	✗
Get	0.4	1	✗
Debug Logging	0.5	2	✓
In Memory	0.2	2	✓
Dinamic	0.4	2	✓
Static	0.4	1	✗
LFU	0.4	1	✗
LRU	0.4	1	✗
Win	0.5	2	✓
NutOS	0.5	2	✓

Table 2: Input parameters to compute DOR for the FAME-DBMS example

Alternatively, SIMPLE proposes the more reliable equation 6 to calculate homogeneity, where: t is the number of features of the SPL and C_F is the commonality of feature F. Using equation 6 for the scenario proposed in the previous paragraph, we check the SPL is certainly heterogeneous (i.e., homogeneity = $\frac{\sum_{i=1}^{100} 0.02}{100} = 0.02$). Note that commonality is the basis for this reliable calculation.

$$\text{Homogeneity} = \frac{\sum_{i=1}^{t} C_F}{t} \tag{6}$$

As a preliminary step to compute commonality, in section 4.1 we propose how to calculate the total number of products n of a feature diagram; n is useful on its own to calculate the *variability factor* of a feature diagram. Variability factor is a value between 0 and 1 that is computed by Equation 7. The smaller the ratio the more restrictive is the feature diagram and vice-versa. The relevance of the variability factor has been related to decision-making strategies for adopting the product line approach [6].

$$\text{Variability} = \frac{n}{2^{\text{total number of features}}} \tag{7}$$

4 Computing Commonality

In the previous section, we have summarized some approaches to identify problematic features (i.e., when the payoff threshold of a feature is too high), to measure the level of reuse achieved by the SPL (i.e., the DOR) and to evaluate the similarity among the products of a SPL (i.e., the homogeneity). All those approaches require knowing the commonality of the features beforehand.

This section proposes an algorithm to compute feature commonality from a feature diagram. According to equation 1, to compute the commonality of a given feature, it is necessary to calculate the number of products that include it and compare it with the total number of products in the SPL. Section 4.1 describes how to compute the total number of products in a SPL. In section 4.2 we shall extend this procedure to compute the products that include a certain feature. These procedures are different, because for a feature node other than the root, the number of products the feature appears in, depends not only on the subtree below the feature, but also on the rest of the tree.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

4.1 Total Number of Products

The number of products of a node n is denoted as $P(n)$. Thus, the total number of products represented by a NFT diagram is $P(r)$, where r is the root. For a leaf node l, $P(l) = 1$. Table 3 shows the formulas to compute $P(n)$ for a non-leaf node n with s children, $n_1, n_2, \ldots n_s$, of type *mandatory* (i.e., n is labeled with $\text{card}_s[s..s]$), *optional* ($\text{card}_s[0..s]$), *or* ($\text{card}_s[0..s]$) and *xor* ($\text{card}_s[1..1]$) respectively. All these formulas are linear, hence, time complexity for computing $P(n)$ in these cases is $O(s)$. Therefore, in these particular cases, the complexity for computing $P(r)$ is linear on the number of nodes, i.e., $O(N)$.

type of relationship	formula
mandatory ($\text{card}_s[s..s]$)	$P(n) = \prod_{i=1}^{s} P(n_i)$
optional ($\text{card}_s[0..s]$)	$P(n) = \prod_{i=1}^{s} (P(n_i) + 1)$
or ($\text{card}_s[1..s]$)	$P(n) = (\prod_{i=1}^{s} (P(n_i) + 1)) - 1$
xor ($\text{card}_s[1..1]$)	$P(n) = \sum_{i=1}^{s} P(n_i)$

Table 3: Total number of products for *mandatory*, *optional*, *or* and *xor* relationships

In general, when a node n has s children and is labeled with $\text{card}_s[low..high]$, $P(n)$ is calculated by equation 8, where S_k is the number of products choosing any combination of k children from s. For the sake of clarity, let us denote $P(n_1), P(n_2), \ldots P(n_s)$ as $p_1, p_2, \ldots, p_s$. In a straightforward approach, S_k can be calculated by summing the number of products of all possible k-combinations (see equation 9). Unfortunately, this calculation has exponential complexity.

$$P(n) = \sum_{k=\text{low}}^{\text{high}} S_k \tag{8}$$

$$S_k = \sum_{1 \le i_1 < i_2 < i_3 \ldots < i_k \le s} p_{i_1} p_{i_2} \ldots p_{i_k} \tag{9}$$

A better complexity can be reached by using recurrent relations. The base case is $S_0 = 1$. According to equation 9, $S_1 = \sum_{i=1}^{s} p_i$. Calculating S_2, the number of products for combinations of 2 siblings that include n_1 is $p_1 p_2 + p_1 p_3 \ldots + p_1 p_s = p_1(p_2 + p_3 + \ldots + p_s) = p_1(S_1 - p_1)$. Likewise, the number of products of 2-combinations that include n_2 is $p_2(S_1 - p_2)$. Adding up every 2-combinations, we get $\sum_{i=1}^{s} p_i(S_1 - p_i)$. However, in the sum each term $p_i p_j$ is being accounted for twice; once in the round for i and another in the round for j. Thus, removing the redundant calculations:

$$\begin{aligned} S_2 &= \frac{1}{2} \sum_{i=1}^{s} p_i(S_1 - p_i) \\ &= \frac{1}{2}(S_1 \sum_{i=1}^{s} p_i - \sum_{i=1}^{s} p_i^2) \\ &= \frac{1}{2}(S_1^2 - \sum_{i=1}^{s} p_i^2) \end{aligned}$$

Calculating S_3, the number of products for combinations of 3 siblings that include n_1 is p_1 multiplied by the number of products for 2-combinations that do not contain n_1, i.e., $p_1(S_2 - p_1(S_1 - p_1))$. Adding up every 3-combinations:

$$\sum_{i=1}^{s} p_i(S_2 - p_i(S_1 - p_i)) = S_2 S_1 - S_1 \sum_{i=1}^{s} p_i^2 + \sum_{i=1}^{s} p_i^3$$

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

This time, every triple $p_i p_j p_k$ is being accounted for three times. Hence, removing the redundant computations:

$$S_3 = \frac{1}{3}\left(S_2 S_1 - S_1 \sum_{i=1}^{s} p_i^2 + \sum_{i=1}^{s} p_i^3\right)$$

Our reasoning leads to the general equation 10 which allows efficient computation of the S vector. Though we will compute it from the pseudocode later on, combining equations 8 and 10, we could conclude that the total number of products of a SPL represented by a NFT diagram can be calculated in quadratic time, i.e., $O(N^2)$; what constitutes a considerable improvement from exponential to polynomial computational complexity.

$$S_0 = 1$$
$$S_k = \frac{1}{k}\sum_{i=0}^{k-1}((-1)^i S_{k-i-1}\sum_{j=1}^{s} p_j^{i+1}) \text{ for } 1 \le k \le s \tag{10}$$

In appendix A, Algorithm 3, and its auxiliary Algorithms 4 and 5, implement the calculation of P.

Let us consider the simple diagram in Figure 8. It is easy to compute that nodes B and D generate 7 products each and C generates 3. Since A has *or* cardinality, we could use the corresponding equation $P(n) = (\prod_{i=1}^{s}(P(n_i)+1)) - 1$. Thus, P(A) = (7+1)(3+1)(7+1)-1 = 255. As an example, we will compute P(A) using equation 10. We will begin computing the powers of the number of products from the children of A and their sum (Table 4).

power	B	C	D	sum
1	7	3	7	17
2	49	9	49	107
3	343	27	343	713

Table 4: Powers of the number of products from the children of A and their sum

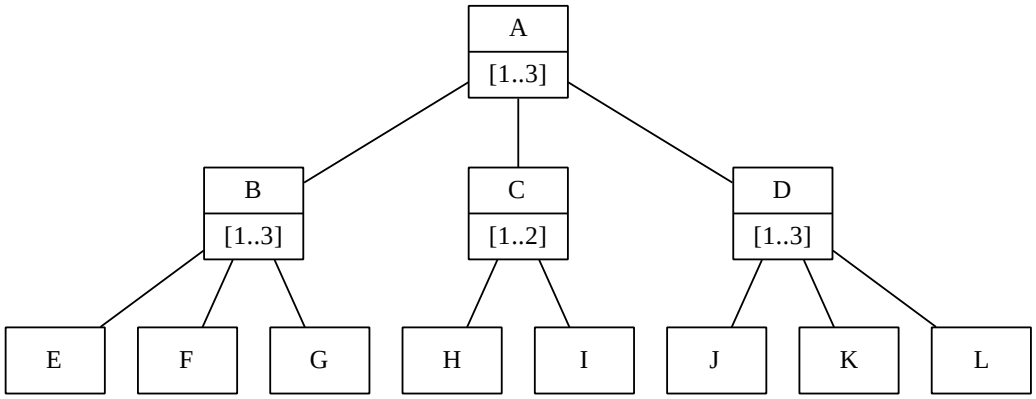

Figure 8: A sample FD

Now, $S_0 = 1$ by definition, $S_1 = 17$, as it is the sum of children's products, $S_2 = 1/2(17 \cdot 17 - 1 \cdot 107) = 91$, following the general formula 8 and $S_3 = 1/3(91 \cdot 17 - 17 \cdot 107 + 1 \cdot 713) = 147$. Adding up S_1, S_2 and S_3, we get again 255.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

The nested loops in Algorithm 5 determine the complexity of Algorithm 3. Since High $\leq N$, it is safe to say the complexity is $O(N^2)$. Thus, #Products^{OD} is in $O(N^2)$ given that #Particular is linear and #General is quadratic.

4.2 Updating the Number of Products Considering the Entire Context of Nodes

We will now tackle another question. Let n be a node, with s children whose number of products are respectively p_1, p_2, ..., p_s, and let us suppose we have computed already $P(n)$ using equation 10. This calculation would provide us with vector S. What would happen if we should add a new child with p_{s+1} products? We may compute a new vector S' using the general equation, but it is possible to derive S'_i from S_i directly, for any suitable i.

Obviously, S'_i will contain all the possibilities in S_i, since all of them are valid combinations of i children of n. These are the combinations in S'_i which do not include the new node. The combinations including the new child amount to $p_{s+1} \cdot S_{i-1}$. So, $S'_i = S_i + p_{s+1} \cdot S_{i-1}$.

In order to calculate the real value of $P(n)$ (i.e., considering not only the descendants of n, but also its antecessors and siblings), what we really want to do is exactly the opposite, i.e., having computed S_i, eliminate a child m and compute the vector S'_i (equation 11).

$$\begin{aligned} S'_0 &= 1 \\ S'_i &= S_i - p_m \cdot S_{i-1} \end{aligned} \tag{11}$$

Going back to our previous example, say we want to eliminate node C. Now $S_0 = 1$ by definition, $S_1 = 17 - 3 \cdot 1 = 14$, $S_2 = 91 - 3 \cdot 14 = 49$ and $S_3 = 147 - 3 \cdot 49 = 0$ (as expected, since there are only two siblings left). Let us focus on the subtree with root E. The valid products for this subtree are {{E}} and, consequently, P(E)=1 using equation 10. Now, let us consider the subtree with root B, which has children E, F and G. The valid products for this subtree are {{E}, {F}, {G}, {E, F}, {E, G}, {F, G}, {E, F, G}}. Using equation 10 we get P(E)=P(F)=P(G)=1 and P(B)=7. Whereas P(B) is valid, the P values for its children must be updated considering their full context (e.g., though E is included in 4 products, P(E)=1). Algorithm 6 (appendix A) uses equation 11 to support the updating. For instance, the number of products that include E is P'(E)=P(E)·TakeOneOut(7,1,3,[1,1,1],1) = 1·4 = 4.

Now, we are ready to present the Algorithm 1 (appendix A) to calculate commonality. First, we will use an array PList to store the number of products for each node, taken as the root of the corresponding subtree. Then, we will multiply this amount by the variability provided by the siblings of the node, propagating the variability to the whole subtree. When we have the definitive number of products each node appears in, it is immediate to compute the commonality CList. The number of products for each node is computed in a bottom-up approach.

Our example is three-level-deep (i.e., level 1 includes node A, level 2 includes B, C and D, and level 3 includes E, F, G, H, I, J, K and L). Figure 9 depicts the calculation of P for all nodes step by step:

1. Calculating P for level 3: the call to #Products^{OD} for leaf nodes returns 1, therefore PList[E] = PList[F] = ... = PList[L] = 1.

2. Calculating P for level 2 and recalculating P for level 3:

   (a) PList[B] = #Products^{OD}([1,1,1],1,3) = $2 \cdot 2 - 1 = 7$. Now we recompute PList[E], PList[F] and PList[G]. In these three cases, P'_B = TakeOneOut(7,1,3,[1,1,1],1) = 4, so PList[E] = PList[F] = PList[G] = 4.

   (b) PList[C] = #Products^{OD}([1,1],1,2) = $2 \cdot 2 - 1 = 3$, P'_C = TakeOneOut(3,1,2,[1,1],1) = 2, so now PList[H] = PList[I] = 2.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

(c) The subtree headed by D has the same structure as that of B.

3. Calculating P for level 1 and recalculating P for levels 2 and 3:

   (a) PList[A] = #Products^{OD}([7,3,7],1,3) = $8 \cdot 4 \cdot 8 - 1 = 255$.

   (b) For the subtree under B, P'_A = TakeOneOut(255,1,3,[7,3,7],7) = 32 and then, PList[B] = $7 \cdot 32 = 224$, PList[E] = PList[F] = PList[G] = $4 \cdot 32 = 128$.

   (c) For the subtree under C, P'_A = TakeOneOut(255,1,3,[7,3,7],3) = 64, so PList[C] = $3 \cdot 64 = 192$. Now PList[H] = PList[I] = $2 \cdot 64 = 128$.

   (d) The case for the subtree under D is symmetrical to that under B.

Finally, commonality is computed: CList[A] = 1, CList[B] = 224 / 255 = .87, CList[C] = 192/255 = .75, CList[D] = .87, CList[E] = CList[F] = CList[G] = 128/255 = .50, CList[H] = CList[I] = 128/255 = .50, CList[J] = CList[K] = CList[L] = .50.

5 Computational Complexity

The algorithm just presented is quadratic in the number of features. For a clearer analysis, it may help if we consider the operations step-by-step. As we have already noted, #Products^{OD} is $O(N^2)$. If we call that function for all the nodes, as we do in the first loop in #Products, the result is in $O(N^2)$, where N is the total number of nodes. This can easily be proven by means of structural induction: the leaf-nodes are the base case of the induction and they take constant time to be processed, so the condition holds trivially. Let now n be the root of the diagram with children $n_1, n_2, \ldots n_s$ with $N_1, N_2, \ldots, N_s$ nodes in their respective subtrees. Our induction hypothesis is that #Products$^{OD}(n_i) \in O(N_i^2)$. So, the time for the first loop in #Products is delimited by equation 12.

$$\sum_{i=1}^{s} N_i^2 + k \leq \left(\sum_{i=1}^{s} s \right)^2 + sk = N^2 + sK \leq N^2 + Nk \in O(N^2) \tag{12}$$

Where k is a constant that represents the time it takes to append one item to PChildrenList.

Therefore, the first loop is quadratic. Next there is a call to Products^{OD}, which we know to be quadratic. Finally, we have to consider the second loop. The call to TakeOneOut for some node n_i takes time in $O(N_i)$ and the inner loop of its descendants also is $O(N_i)$, so the body in the second loop of #Products belongs in $O(N_i^2)$. We apply again the argument expressed in equation 12 to conclude that this second loop in #Products is again $O(N^2)$. Therefore, the sequence of the operations is $O(N^2)$.

Commonality computing just calls #Products and then traverses all the nodes to perform a division, so the complexity for the algorithm Commonality is $O(N^2 + N) = O(N^2)$.

6 Experimental Evaluation

Following the directions given by Juristo et al. [26], this section evaluates our algorithm experimentally.

6.1 Objective Definition

The Objectives (O) of our evaluation are:

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

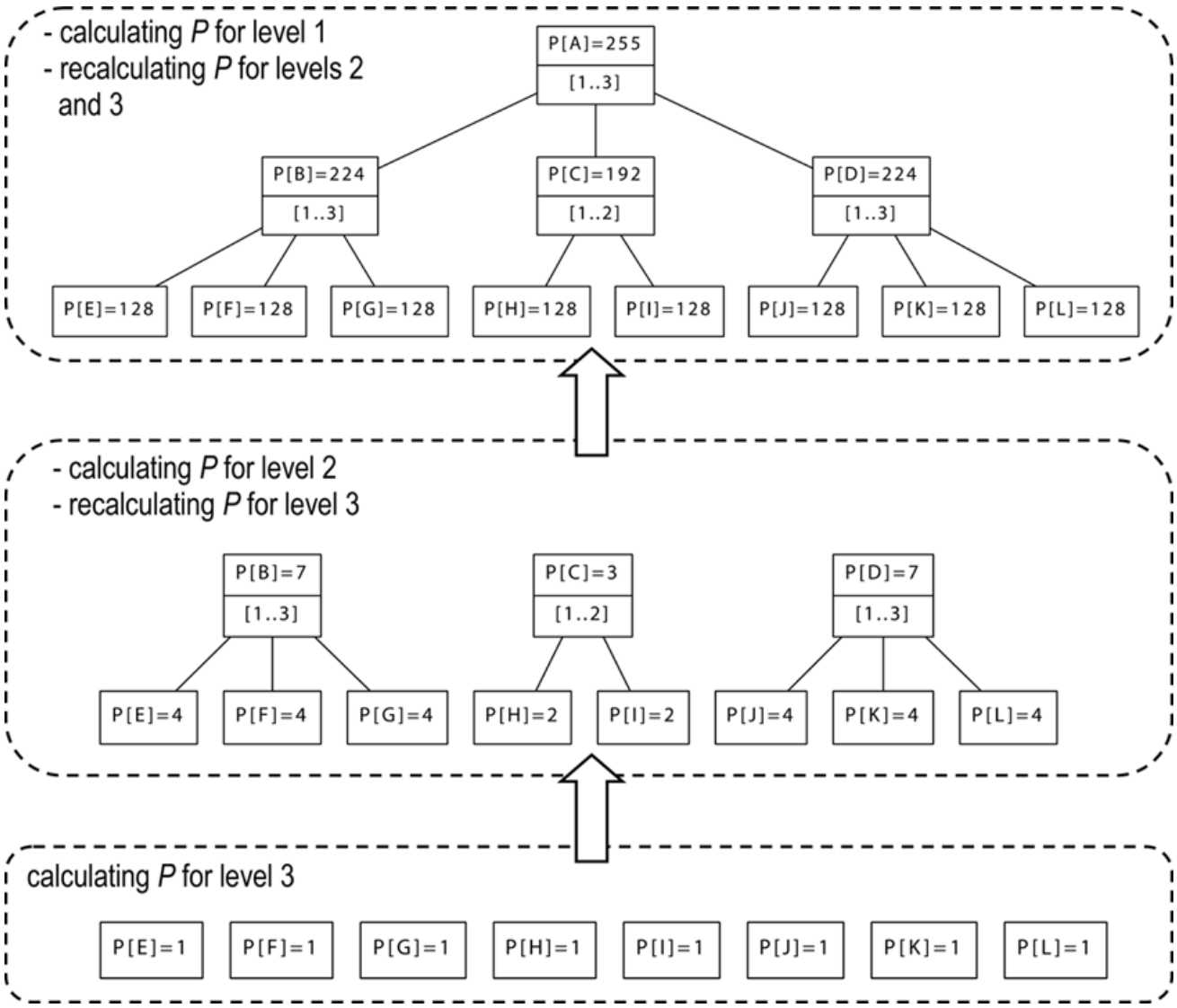

Figure 9: Calculating P step-by-step

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

1. O_1: To validate the results of our algorithm (i.e., are the results of our algorithm correct?). Since computing commonality by hand is unfeasible except for tiny feature diagrams, we will validate our algorithm results by comparing them to the results returned by alternative proposals to compute commonality.

2. O_2: To evaluate the scalability of our algorithm (i.e., does our algorithm work for feature diagrams of any size?) and comparing it to the scalability of alternative proposals.

To the extent of our knowledge [7], the only proposals to compute commonality alternative to our approach are [5, 4]. Unfortunately, those proposals rely on general purpose logic tools and do not scale except for the smallest diagrams (see section 7). Consequently, we have just used such proposals to verify that the commonality results of our algorithm are correct.

Since the alternative proposals we consider for direct experimental comparison do not really compute the commonality, it would be inappropriate to compare them with our algorithm, so the experiment only considers computing the number of products in the exemplar feature diagrams. Therefore the Hypotheses (H) for testing are:

1. H_1: Our algorithm computes the number of products correctly.

2. H_2: Our algorithm scales for large feature diagrams.

6.2 Experimental Design

A set of feature diagrams is automatically generated and the total number of products is counted using three different approaches: the propositional-logic exact model counters *cachet* [40] and *relsat* [3], and our algorithm. Although exact model counters have exponential time complexity for the worst cases, there is experimental evidence that they perform well for certain formulaes with two thousand variables [39].

As noted in section 2, the group cardinality constructor *card* generalizes any kind of relation between features (e.g., mandatory, optional...). So, the experiment is focused on how the constructor *card* is managed. The feature diagrams consist of a root node n, with s terminal children and cardinality $h..h+1$ (i.e., $\lambda(n) = \text{card}_s[h..h+1](n_1, ..., n_s)$), where h is the integer division of s by 2. For instance, Figure 10 depicts the corresponding feature diagram for 4 terminal nodes.

Since the input to cachet and relsat are logic formulas in CNF (Conjunctive Normal Form[4]), we sketch here the feature diagram translation to CNF following the directions in [8]. The tree-structure is dealt with $s+1$ clauses: there is a clause to express that node n is true. Also, each child n_i implies the parent node n. For instance, the tree-structure in Figure 10 is encoded by:

$$A \wedge (B \rightarrow A) \wedge (C \rightarrow A) \wedge (D \rightarrow A) \wedge (E \rightarrow A) \equiv$$
$$A \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg D \vee A) \wedge (\neg E \vee A)$$

For the cardinality restriction, we treat the *low* and *high* restrictions separately. Saying that at least *low* children have to be present in a product is equivalent to say that at most $s - low$ children can be excluded (i.e., in the logical formula no more than $s - low$ can be false). Which means that as soon as $s - low + 1$ literals are selected, at least one of them must be true (this constraint is a clause). So, the *low* restriction is equivalent to the conjunction of all possible clauses obtained by choosing $s - low + 1$ children of n. For instance, in Figure 10 the low limit is encoded by:

[4] In boolean logic, a formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal is an atomic formula or its negation.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

$$(B \vee C \vee D) \wedge (B \vee C \vee E) \wedge (B \vee D \vee E) \wedge (C \vee D \vee E)$$

The *high* restriction is somewhat easier. Since in a set of $high + 1$ children at least one of them has to be false, we just compute all the sets of children of size $high + 1$ and add a clause with all the set members negated.

$$\neg(B \wedge C \wedge D \wedge E) \equiv \neg B \vee \neg C \vee \neg D \vee \neg E$$

To sum up, Figure 10 is encoded by the following formula with 10 clauses:

$$\begin{aligned} &A \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg D \vee A) \wedge (\neg E \vee A) \wedge \\ &(B \vee C \vee D) \wedge (B \vee C \vee E) \wedge (B \vee D \vee E) \wedge (C \vee D \vee E) \wedge \\ &(\neg B \vee \neg C \vee \neg D \vee \neg E) \end{aligned}$$

In order to support the replication of the experiment, a prototype implementation of our algorithm, the experiment described in this section and a number of case studies are available on:

`https://sourceforge.net/projects/commonality-spl`

6.3 Experimental Results

The results of the experiment are summarized by Table 5, and Figures 11 and 12[5]. Whenever the approaches are able to compute the number of products, the computed results always coincide.

6.4 Result Analysis

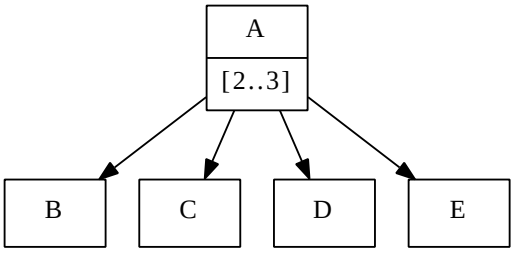

Figure 10: Testing FD with 4 terminal nodes

According to the experimental results, the hypotheses H_1 and H_2 are satisfied. Regarding H_2, the number of products and the number of clauses grows exponentially with the number of nodes (see in Figure 11). As showed by Table 5 and Figure 12, cachet and relsat have a hard time keeping up with the pace of growth of the input: cachet gives up at size 15 and relsat at size 20, while our proposal takes less than 10 milliseconds to complete in all cases, which goes to show that using purely logic tools does not provide a scalable solution for the problem of feature model counting in the presence of extended

[5]The time it takes for cachet, relsat and our proposal to compute the number of products is subject of an error of ±10 milliseconds in Table 5 and Figure 12.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

#terminal nodes	#clauses	#products	time (milliseconds)		
			cachet	relsat	our algorithm
1	2	2	30	0	0
2	4	3	30	0	0
3	6	6	30	0	0
4	10	10	30	0	0
5	16	20	30	0	0
6	28	35	40	10	0
7	50	70	30	10	0
8	93	126	30	0	0
9	178	252	40	10	0
10	341	462	40	40	0
11	672	924	40	140	0
12	1300	1716	40	550	0
13	2588	3432	40	3130	0
14	5020	6435	*error*	10890	0
15	10026	12870	*error*	56780	0
16	19465	24310	*error*	249890	0
17	38914	48620	*error*	1.72E+06	0
18	75601	92378	*error*	6.11E+06	0
19	151184	184756	*error*	2.85E+07	0
20	293951	352716	*error*	*error*	0

Table 5: Summary of the experiment

cardinality. This is because cachet and relsat, as well as the vast majority of SAT-solvers and exact model counters, rely on a technique called DPLL[6] [15], which is exponential on the number of clauses. To make matters worse for them, in this case, the number of clauses also grows exponentially with the number of nodes. Admittedly, real feature models are not likely to display such a complex structure, but then again, extended cardinality could not be efficiently processed hitherto.

7 Related Work

In recent years, many researchers have worked on the automated analysis of feature diagrams including, but not limited to, consistency checking of a feature diagram [31], configuration support [1] and safe refactoring transformations [20]. Nevertheless, proposals for computing commonality are rare [7].

Benavides et al. [4, 5] translate feature diagrams into propositional logic formulas (i.e., feature diagram $\rightsquigarrow \psi$). Off-the-shelf tools, such as SAT and CSP solvers are then used to enumerate all the different sets of variable assignments that satisfy the logic formula ψ. Each one of these sets represents a particular product. There is a correspondence between features and variables, so that if a feature F is encoded in ψ by a boolean variable V, then F is included in the product represented by a set S iff the value of V in S is true. Hence, commonality of F is calculated by counting the number n_F of sets where V is true, and dividing n_F by the total number of sets.

As noted by Sang et al. [39], any backtracking SAT algorithm can be trivially extended to one that counts the number of satisfying assignments by simply forcing it to backtrack whenever a solution is

[6] DPLL stands for Davis-Putnam-Logemann-Loveland, the inventors of the DPLL technique.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

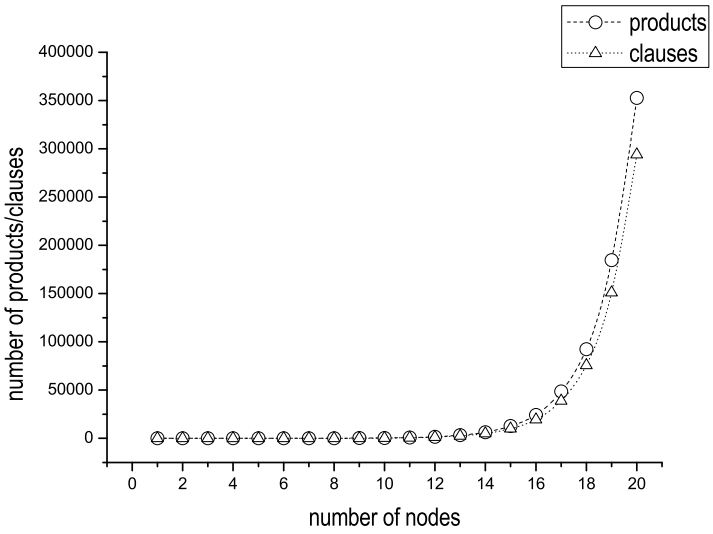

Figure 11: Growth of the number of products and clauses by the number of nodes

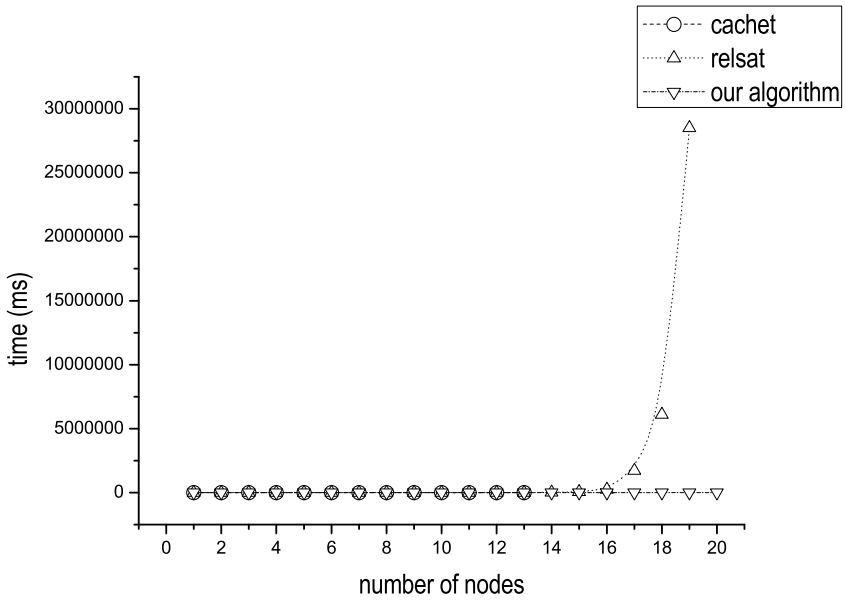

Figure 12: Growth of the time to calculate the number of products by the number of nodes

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

found. However, such a simple approach, is unfeasible for all but the smallest problem instances. As Benavides reports [5], CSP technology scales even worse than SAT-solvers to compute the number of satisfying assignments. The approach can be improved by using tools specifically designed for counting the number of valid assignments that a formula has, such as *relsat* [2] and *cachet* [39, 40], two state of the art model counters for propositional logic. The scalability of this approach has been tested in the previous section.

Regarding the calculation of feature commonality with model counters, in [39], Sang et al. propose how to extend cachet to compute the marginal probabilities of each variable, i.e., the probability that a variable is present in a random product. Although that probability would correspond to feature commonality, the current cachet implementation does not support it.

Mendonça [32] proposes another approach that uses BDDs to generate partial assignments to variables representing features as an intermediate step in computing the number of products. The actual counting is performed efficiently, taking advantage of the tree structure in a similar way to the function we have presented in Algorithm 3 (see appendix A). Unlike Benavides', Mendonça's approach does not compute commonality. Moreover, it lacks the expressive power of the cardinality construct, as it deals only with the standard FODA cardinalities *mandatory, optional, or* and *xor*. Regarding the scalability of the approach, constructing a BDD from a formula may require large amounts of memory depending on the variable ordering for representing the BDD. The size of the resulting BDD can be reduced with a good variable ordering, though computing the best variable ordering is an NP-hard problem [25].

To sum up, currently available proposals to compute commonality and the total number of products handle feature diagrams as logic formulas, which are processed by off–the–shelf tools designed for propositional logic. The main advantage of such approach is being able to tackle cross-tree constraints. However, it has the drawback of running in worst-case exponential time (and this is true even without considering cross-tree constraints). In contrast, we are proposing a non–logic–based approach that has quadratic complexity and, consequently, can work effectively for diagrams of any size. On the other hand, our algorithm does not take into account cross-tree constraints. However, it could be combined with DPLL-style search [15] to manage such kind of constraints (although scalability would probably be affected).

8 Conclusions

Commonality measures the reuse ratio of features among the products in a SPL. In this paper, we have discussed its importance in SPL scoping and provided an algorithm to compute commonality in just quadratic time on the number of features (i.e., our proposal scales for large feature diagrams).

To make our proposal as general as possible, we have specified the algorithm for an abstract notation for feature diagrams, named NFT, that works as a pivot language for most of the available notations. We have formally defined the abstract syntax and semantics of NFT.

It is interesting to note that we have added an extension to usual FODA cardinalities (i.e., the card[low, high] construction) without incurring in any complexity penalty, since commonality calculation without it would still be quadratic. The expressive power of the formalism has been thus improved at not asymptotic cost.

9 Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful feedback. This work has been partially supported by the Spanish Government under the CICYT project DPI2008-05444, by the Comunidad de Madrid under the research network S2009/DPI-1559 and by the Universidad Nacional de Educación a Distancia under grant 2010V/PUNED/004.

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

References

[1] Don Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference*, pages 7–20, 2005.

[2] R.J. Bayardo and J.D. Pehoushek. Counting models using connected components. In *17th National Conference on Artificial Intelligence*, pages 157–162, Austin, Texas, USA, 2000.

[3] Roberto J. Bayardo and Joseph Daniel Pehoushek. Counting models using connected components. In *17th National Conference on Artificial Intelligence*, pages 157–162, Austin, Texas, USA, 2000. AAAI Press / The MIT Press.

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. volume 3520, pages 491–503, 2005.

[5] David Benavides. *On the automated analysis of software product lines using feature models. A framework for developing automated tool support*. PhD thesis, University of Seville, Spain, June 2007.

[6] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated reasoning on feature models. In *17th International Conference on Advanced Information Systems Engineering*, pages 491–503, Porto, Portugal, 2005.

[7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[8] Armin Biere, Marijn J.H. Heule, Hans van Maaren, Toby, and Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[9] Thorsten Blecker and Nizar Abdelkafi. The development of a component commonality metric for mass customization. *IEEE Transactions on Engineering Management*, 54:70–85, February 2007.

[10] Barry Boehm, A. Winsor Brown, Ray Madachy, and Ye Yang. A software product line life cycle cost estimation model. In *International Symposium on Empirical Software Engineering*, Redondo Beach, CA, USA, 2004.

[11] Paul Clements, John McGregor, and Sholom Cohen. The structured intuitive model for product line economics. Technical report, CMU/SEI-2005-TR-003, Software Engineering Institute, 2005.

[12] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[13] Sholom Cohen. Predicting when product line investment pays. Technical report, CMU/SEI-2003-TN-017, Software Engineering Institute, 2003.

[14] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods Tools and Applications*. Addison-Wesley, 2000.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[16] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.

[17] Christian Dinnus and Klaus Pohl. *Software Product Line Engineering*, chapter Experiences with Software Product Line Engineering, pages 413–434. Springer Berlin Heidelberg, 2005.

[18] Magnus Eriksson, Jürgen Böstler, and Kjell Borg. The pluss approach – domain modeling with features, use cases and use case realizations. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin Heidelberg, 2005.

[19] Daniel D. Galorath and Michael W. Evans. *Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves*. Auerbach Publications, 2006.

[20] R. Gheyi, T. Massoni, and P. Borba. Algebraic laws for feature models. *Journal of Universal Computer Science*, 14(21):3573–3591, 2008.

[21] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[22] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns Models Framworks and Tools*. Wiley, 2004.

[23] Martin Griss, John Favaro, and Massimo Alessandro. Integrating feature modeling with the rseb. In *5th International Conference on Software Reuse*, pages 76–85, Washington, DC, USA, 1998.

[24] David Harel and Bernhard Rumpe. Modeling languages: Syntax semantics and all that stuff - part i: The basic stuff. Technical report, Faculty of Mathematics and Computer. The Weizmann Institute of Science MCS00-16, 2000.

[25] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[26] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer, 2001.

[27] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[28] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[29] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmuller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: Analysis and case studies. In *13th International Software Product Line Conference*. SEI, August 2009.

[30] Sana Ben Abdallah Ben Lamine, Lamia Labed Jilani, and Henda Hajjami Ben Ghezala. Cost estimation for product line engineering using cots components. In *9th International Software Product Line Conference*, Rennes, France, 2005.

[31] Mike Mannion. Using first-order logic for product line model validation. In *2nd International Software Product Line Conference*, pages 176–187, London, UK, 2002. Springer-Verlag.

[32] Marcilio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[33] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Requirements Engineering Conference*, pages 243–253, 2007.

[34] J. Pe na, M. Hinchey, A. Ruiz-Cortés, and P. Trinidad. Building the core architecture of a multiagent system product line: With an example from a future nasa mission. In *7th International Workshop on Agent Oriented Software Engineering*, pages 208–224, Hakodate, Japan, 2006.

[35] Jarley Palmeira Nobrega, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Income: Integrated cost model for product line engineering. In *34th Euromicro Conference Software Engineering and Advanced Applications*, Parma,Italy, 2008.

[36] Klaus Pohl, Gunter Bockle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[37] Jeffrey S. Poulin. The economics of software product lines. *International Journal of Applied Software Technology*, 3(1):20–34, March 1997.

[38] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *6th World Conference on Integrated Design & Process Technology*, Pasadena, California, 2002.

[39] Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *7th International Conference on Theory and Applications of Satisfiability Testing*, pages 20–28, 2004.

[40] Tian Sang, Paul Beame, and Henry A. Kautz. Heuristics for fast exact model counting. In *8th International Conference on Theory and Applications of Satisfiability Testing*, pages 226–240, 2005.

[41] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[42] Timothy W. Simpson, Zahed Siddique, and Roger Jianxin Jiao. *Product Platform and Product Family Design: Methods and Applications*. Springer, 2005.

[43] BigLever Software. Gears. http://www.biglever.com/index.html.

[44] Pure Systems. pure::variants. http://www.pure-systems.com/.

[45] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. Improving decision making in software product lines product plan management. In *Software Product Management: Issues and Perspectives*. The Icfai University Press, 2008.

[46] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture*, pages 45–54, Amsterdam, The Netherlands, 2001.

[47] Jacco H. Wesselius. *Software Product Lines: Research Issues in Engineering and Management*, chapter Strategic Scenario-Based Valuation of Product Line Roadmaps, pages 53–89. Springer Berlin Heidelberg, 2006.

A Algorithm to Calculate Commonality

This appendix describes in detail our algorithm to calculate the commonality of the features modeled by a feature diagram. In order to facilitate the understanding of the algorithm, it has been decomposed into a main program (algorithm 1) and five auxiliary subprograms (algorithms 2, 3, 4, 5 and 6).

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

Algorithm 1: Commonality(Tree, CList)

Data: Tree is a NFT diagram
Result: CList includes the commonality values for all Tree nodes
begin
  #Products(RootOfTree, PList)
  forall the *n node in Tree* **do**
    CList[n] ← PList[n] ÷ PList[RootOfTree]

Algorithm 2: #Products(n, PList)

Result: PList stores the total number of products for each tree node
begin
  /* computing PList in a bottom-up approach */
  PChildrenList$_n$ ← [ ]
  forall the *m child of n* **do**
    #Products(m, PList)
    PChildrenList$_n$ ← PChildrenList$_n$ ∪ PList[m]
  /* calculating PList considering exclusively descendants */
  PList[n], SList$_n$ ←
    #Products^{OD}(PChildrenList$_n$, Low$_n$, High$_n$)
  /* updating PList considering the entire context */
  forall the *m child of n* **do**
    P'_n ← TakeOneOut(PList[n], Low$_n$, High$_n$,
      SList$_n$, PList[m])
    PList[m] ← PList[m] · P'_n
    forall the *d descendant of m* **do**
      PList[d] ← PList[d] · P'_n

Algorithm 3: #Products^{OD}(PChildrenList, Low, High): P, SList

Data: PChildrenList includes the P values of the children of the current node; cardinality limits of the node are [Low, High]
Result: P is the total number of products for the node; SList includes the node S values (OD: Only node Descendants are considered)
begin
  if *(node is leaf)* ∨ *(children are mandatory* ∨ *optional* ∨ *or* ∨ *xor)* **then**
    P ← #Particular(PChildrenList, Low, High)
    SList ← nil
  else
    P, SList ← #General(PChildrenList, Low,
      High)
  return P, SList

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

Algorithm 4: #Particular(PChildrenList, Low, High): P

```
begin
    P ← 1
    N ← number of children
    if Low=N ∧ High=N then                                   // mandatory
        forall the i such that 1 ≤ i ≤ N do
            P ← P·PChildrenList[i]
    else if Low=0 ∧ High=N then                              // optional
        forall the i such that 1 ≤ i ≤ N do
            P ← P·(1 + PChildrenList[i])
    else if Low=1 ∧ High=N then                              // or
        forall the i such that 1 ≤ i ≤ N do
            P ← P·(1 + PChildrenList[i])
        P ← P − 1
    else if Low=1 ∧ High=1 then                              // xor
        forall the i such that 1 ≤ i ≤ N do
            P ← P + PChildrenList[i]
    // else leaf (do nothing)
    return P
```

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

Algorithm 5: #General(PChildrenList, Low, High): P, SList

```
begin
    N ← number of children
    if Low = 0 then
        P ← 1
    else
        P ← 0
    SList[0] ← 1
    PowerSumList[0] ← N
    forall the k such that 1 ≤ k ≤ N do
        PowerList[k] ← 1
    forall the k such that 1 ≤ k ≤ High do
        ThisPowerSum ← 0
        forall the j such that 1 ≤ j ≤ N do
            PowerList[j] ←
              PowerList[j]·PChildrenList[j]
            ThisPowerSum ←
              ThisPowerSum+PowerList[j]
        PowerSumList[k] ← ThisPowerSum
        SList[k] ← 0
        Parity ← 1
        forall the i such that 0 ≤ i < k do
            SList[k] ← SList[k]+Parity·
              SList[k − i − 1]·PowerSumList[i + 1]
            Parity ← −1·Parity
        SList[k] ← SList[k] ÷ k
        if k > Low−1 then
            P ← P+SList[k]
    return P, SList
```

Draft version of the paper published in IET Software, doi: 10.1049/iet-sen.2010.0022

Algorithm 6: TakeOneOut(P, Low, High, SList, P_m): P'

```
begin
    N ← number of children
    if N=0 then                                                    // leaf node
        P' ← 1
    else if Low=N ∧ High=N then                                    // mandatory
        P' ← P ÷ P_m
    else if Low=0 ∧ High=N then                                    // optional
        P' ← P ÷ (P_m + 1)
    else if Low=1 ∧ High=N then                                    // or
        P' ← (P + 1) ÷ (P_m + 1)
    else if Low=1 ∧ High=1 then                                    // xor
        P' ← P − P_m
    else                                                           // the general case
        SList'[0] ← 1
        if Low = 0 then
            P' ← 1
        else
            P' ← 0
        forall the k such that 1 ≤ k < High do
            SList'[k] ← SList[k] − P_m · SList'[k − 1]
            if k > Low − 1 then
                P' ← P' + SList'[k]
    return P'
```