Towards a time-efficient algorithm to calculate the total number of products of a Software Product Line

Completed Research

Ruben Heradio Gil[1] and David Fernandez Amoros[2]

[1] Dpto. de Ingenieria de Software y Sistemas Informaticos
Universidad Nacional de Educacion a Distancia
Madrid, Spain
rheradio@issi.uned.es
[2] Dpto. de Lenguajes y Sistemas Informaticos
Universidad Nacional de Educacion a Distancia
david@lsi.uned.es

Abstract. Feature Diagrams (FDs) are widely used to scope the domain of Software Product Lines (SPLs). In addition, valuable information can be inferred from FDs; for instance, the total number of possible products of a SPL. A common approach to calculate the number of products is translating FDs into propositional logic formulas, which are processed by off-the-shelf tools, such as SAT solvers. However, this approach only works for small FDs. We think more scalable solutions can be reached avoiding the diagram-to-logic transformation and taking advantage of the tree organization of FDs. Due to the profusion of feature modeling notations, this paper formally defines a pivot language, named Neutral Feature Tree (NFT), where FDs are restricted to be trees. Most popular FD notations can be easily and efficiently translated to NFT. The paper also proposes a general and time-efficient algorithm to calculate the number of products without considering crosstree constraints.

1 Introduction

Software Product Line (SPL) practice has become an important and widely used approach for the efficient development of complete portfolios of software products [17]. The domain of a SPL must be carefully scoped, identifying the common and variable requirements of its products and the interdependencies between requirements. In a bad scoped domain, relevant requirements may not be implemented, and some implemented requirements may never be used, causing unnecessary complexity and both development and maintenance costs [6]. To avoid these serious problems, SPL domains are usually modeled with variability languages.

Since FODA's feature modeling language was proposed in 1990 [14], 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 [15], FeatureRSEB [11], Generative Programming [6], Software Product Line Engineering [17], PLUSS [7].
2. In the work of the following authors: M. Riebisch et al. [19], J. van Gurp et al. [24], A. van Deursen et al. [23], H. Gomaa [9].
3. As part of the following tools: Gears [3] and pure::variants [18].

Unfortunately, this profusion of languages hinders the efficient communication among specialists and the portability of variability models between tools. In order to face this problem, P. Schobbens et al. [20, 13, 16] propose the Varied Feature Diagram$^+$ (VFD$^+$) as a pivot notation for variability languages. VFD$^+$ is expressively complete and many variability languages can be easily and efficiently translated into it.

D. Benavides [2] has surveyed a number of analysis operations that infer valuable information from feature diagrams. One these operations is calculating the total number of products of a SPL. This value is used by economic models, such as the Structured Intuitive Model for Product Line Economics (SIMPLE)[5] and the Constructive Product Line Investment Model (COPLIMO)[4], to estimate the SPL costs and benefits. For instance, SIMPLE estimates the cost of building a SPL using equation 1, where: C_{org} expresses how much it costs for an organization to adopt the SPL approach, C_{cab} is the cost of developing the SPL *core asset base*[3], n is the number of products of the SPL, $C_{\text{unique}}(\text{product}_\text{i})$ is the cost of developing the unique parts of a product, and $C_{\text{reuse}}(\text{product}_\text{i})$ is the development cost of reusing core assets to build a product.

$$C_{\text{SPL}} = C_{\text{org}} + C_{\text{cab}} + \sum_{i=1}^{n}(C_{\text{unique}}(\text{product}_\text{i}) + C_{\text{reuse}}(\text{product}_\text{i})) \tag{1}$$

The importance of knowing the number of products of a feature diagram is recognized by many commercial tools for SPL development, such as Gears and pure::variants, which provide the calculation without considering textual constraints between features [4]. On the other hand, some researchers have proposed a full calculation, that includes textual constraints, by translating feature diagrams d into some kind of logic. For instance:

– D. Batory [1] proposes a translation of d into propositional logic. Resulted formulas are processed by off-the-shelf Logic-Truth Maintenance Systems and Boolean Satisfiability (SAT) solvers.
– D. Benavides [2] provides an abstract conversion of d into Constraint Satisfaction Problems (CSP). FaMa Tool Suite [22] adapts this abstract conversion to general CSP solvers, SAT solvers and Binary Decision Diagrams (BDD) solvers.

[3] According to the SPL approach, products are built from a *core asset base*, a collection of artifacts that have been designed specifically for reuse across the SPL.
[4] Textual constraints will be explained in section 2.2.

– Gheyi et al. [8] provides a translation of d into Alloy's logic. Internally, Alloy checks models by converting them into propositional logic and using off-the-shelf SAT solvers.

Unfortunately, the diagram-to-logic approach is only scalable for small feature diagrams. However, in SPLs with complex domains, companies have reported feature models with over 1000 features (e.g., M. Steger et al. [21] present a case study with 5200 features).

This paper proposes a time-efficient algorithm to calculate the SPL number of products without considering textual constraints. Thus, we provide an upper bound for the number of products. In contrast with evaluated commercial tools, the algorithm is fully general and deals with a VFD$^+$ subset, named Neutral Feature Tree (NFT), where diagrams are restricted to be trees[5]. This paper formally defines NFT and demonstrates that NFT and VFD$^+$ are completely equivalent. We think NFT is a good starting point for future implementations of analysis operations which take advantage of feature diagrams structured as trees.

The remainder of this paper is structured as follows. Section 2 formally defines the abstract syntax and semantics of NFT. Section 3 presents the sketch of our algorithm[6]. Finally, section 4 summarizes the paper and outlines directions for future work.

2 An abstract notation for modeling SPL variability

Section 2.1 outlines the main parts of a formal language; sections 2.2 and 2.3 define the abstract syntax and semantics of NFT, respectively; and finally, section 2.4 demonstrates the equivalence between NFT and VFD$^+$.

We emphasize NFT is not meant as a user language, but only as a formal "back-end" language used to define semantics and allow for automated processing.

2.1 Anatomy of a formal language

According to J. Greenfield et al. [10], 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 an 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.

[5] VFD$^+$ diagrams are single-rooted Directed Acyclic Graphs.

[6] There is available an implementation of the algorithm on *http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html*

2. The semantics of a language define its meaning. According to D. Harel et al. [12], a semantic definition consists of two parts: a *semantic domain* $\mathcal{S}$ and a *semantic mapping* $\mathcal{M}$ from the syntactic domain to the semantic domain. That is, $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$.
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 variability languages 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" [6].
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 $n_1 \rightarrow n_2, n_1$ is the *parent* of n_2, and n_2 is a *child* of n_1.
4. $\lambda : (N - \Sigma) \rightarrow \text{card}$ 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[7]:
   (a) whereas many variability 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 K. Czarnecki et al. [6] and NFT.
   (b) whereas, in many variability 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[8] 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: $\forall n \in N \cdot (\exists n' \in N \cdot n' \rightarrow n) \Leftrightarrow n \neq r$.

[7] The same considerations are valid for VFD$^+$.
[8] also named cross-tree constraints [2].

2. d is a tree. Therefore,
   (a) a node may have at most one parent:
   $\forall n \in N \cdot (\exists n', n'' \in N \cdot ((n' \to n) \wedge (n'' \to n) \Rightarrow n' = n''))$
   (b) DE is acyclic: $\nexists n_1, n_2 \ldots, n_k \in N \cdot n_1 \to n_2 \to \ldots \to n_k \to n_1$.
3. card operators are of adequate arities:
   $\forall n \in N \cdot (\exists n' \in N \cdot n \to n') \Rightarrow (\lambda(n) = \text{card}_s) \wedge (s = \|\{(n,n')|(n,n') \in DE\}\|)$

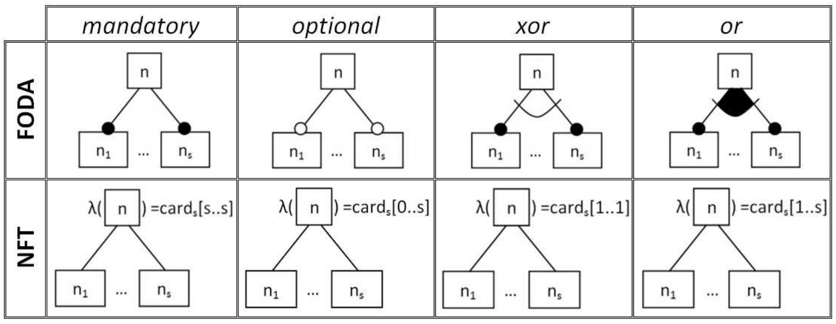

Fig. 1. *card* operator generalizes *mandatory*, *optional*, *or* and *xor* dependencies

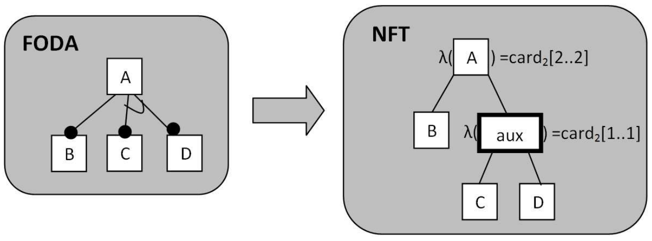

Fig. 2. Different types of dependencies between a node and its children can be expressed in NFT by introducing auxiliary nodes

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 ($\mathcal{M}_{\text{NFT}} : \mathcal{L}_{\text{NFT}} \to \mathcal{P}(\mathcal{P}(\Sigma))$) assigns to every feature diagram d, a product line SPL, 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 $s(s \neq r)$ is in the configuration, one of its parents n, called its justification, must be too: $\forall s \in N \cdot s \in c \wedge s \neq r \cdot \exists n \in N \cdot n \in c \wedge (n, s) \in DE$.
2. A *product* p, named by a valid configuration c, is the set of terminal features of c: $p = c \cap \Sigma$.
3. The *product line* 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 that a feature diagram stands for? In VFD$^+$, P. 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 $\mathcal{T}_{P \rightarrow \Sigma}$:
   (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. **DAGs vs. trees**. Whereas diagrams are trees in NFT, in VFD$^+$ are DAGs. 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 $\mathcal{T}_{\text{DAG} \rightarrow \text{tree}}$:
   (a) $s - 1$ auxiliary nodes $\text{aux}_2, ..., \text{aux}_\text{s}$ are added to the diagram.

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

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

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

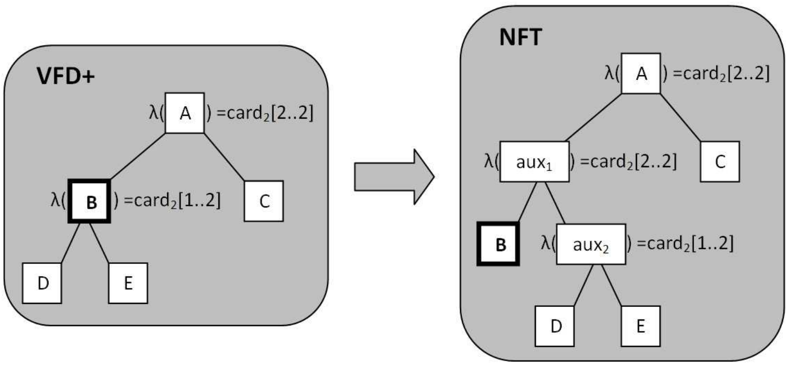

Fig. 3. Any primitive non-leaf node can be converted into a leaf node by using $\mathcal{T}_{P \rightarrow \Sigma}$

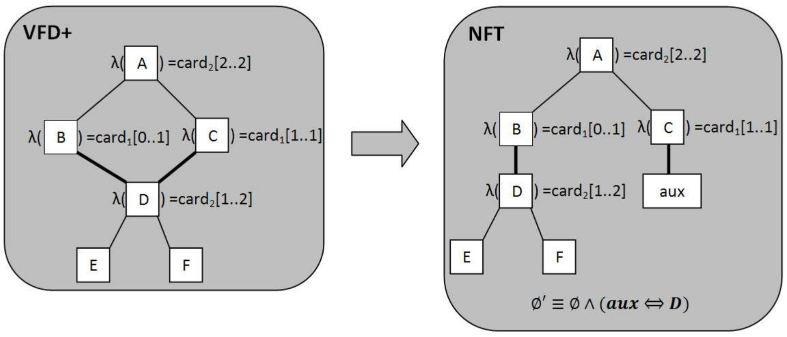

Fig. 4. Any DAG can be converted into a tree by using $\mathcal{T}_{\text{DAG}\rightarrow\text{tree}}$

In order to identify when a transformation on a diagram keeps (1) the diagram semantics and (2) the diagram structure, P. Schobbens [20] defines the notion of *graphical embedding.* A graphical embedding is a translation $\mathcal{T} : \mathcal{L} \rightarrow \mathcal{L}'$ 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 $d \rightarrow d'$, 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, $\mathcal{T}_{P \rightarrow \Sigma}$ and $\mathcal{T}_{\text{DAG} \rightarrow \text{tree}}$ are graphical embeddings that guarantee the equivalency between NFT and VFD$^+$.

3 Calculating the total number of products represented by an NFT diagram without considering textual constraints

This section presents how to calculate the total number of products of a SPL modeled by a NFT diagram without considering textual constraints.

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 includes equations to calculate $P(n)$ for a non-leaf node n that has s children n_i of type *mandatory* (i.e., n is labeled with card$_s[s..s]$), *optional* (card$_s[0..s]$), *xor* (card$_s[1..1]$) and *or* (card$_s[0..s]$). Hence, time-complexity for calculating $P(n)$ in these cases is $O(s)$. Therefore, time-complexity for computing $P(r)$ is quadratic on the diagram number of nodes, i.e., $O(N^2)$.

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

Table 1. Number of products for *mandatory*, *optional*, *or* and *xor* relationships

In general, when a node n has s children and is labeled with card$_s[low..high]$, $P(n)$ is calculated by equation 2, 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 3). Unfortunately, this calculation has the following time-complexity C for P(r): $O(N2^N) \subseteq C \subseteq O(N^2 2^N)$.

$$P(n) = \sum_{k=\text{low}}^{\text{high}} S_k \qquad (2)$$

$$S_k = \sum_{1 \leq i_1 < i_2 < i_3 \ldots < i_k \leq s} p_{i_1} p_{i_2} \ldots p_{i_k} \tag{3}$$

A better time-complexity can be reached by using recurrent equations. The base case is $S_0 = 1$. According to equation 3, $S_1 = \sum_{i=1}^{s} p_i$. Calculating S_2, the number of products for combinations of 2 siblings that include n_1 is $p_1 p_2 + p_1 p_3 ... + p_1 p_s = p_1(p_2 + p_3 + ... + p_s) = p_1(S_1 - p_1)$. Similarly, 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 calculus:

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

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, we get:

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

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

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

Our reasoning leads to the general equation 4, that has a much better time-complexity $O(ks)$. Combining equations 2 and 4, we conclude that the total number of products of a SPL represented by a NFT diagram can be calculated, without considering textual constraints, in cubic time, i.e., $O(N^3)$; what constitutes a considerable improvement from exponential to polynomial computational complexity.

$$S_k = \begin{cases} 1 & \text{if} \quad k = 0 \\ \frac{1}{k} \sum_{i=0}^{k-1} ((-1)^i S_{k-i-1} \sum_{j=1}^{s} p_j^{i+1}) & \text{if} \quad 1 \leq k \leq s \end{cases} \tag{4}$$

3.1 Supporting the calculus for an extension of *card*

As pointed in figure 1, an important contribution of VFD$^+$ is the *card* generalization of the different kinds of relations between features (i.e., mandatory, optional...). At the moment, VFD$^+$ (and NFT) *card* syntax is:

$$\text{card}_s[\text{range}](\text{children})$$

Nevertheless, our proposed calculus for the total number of products supports the following more general syntax:

$$\text{card}_s\langle\text{list}\rangle(\text{children})$$

Where *list* is a non-void list which may include one or more:

1. index $i \in \mathbb{N} \cdot 0 \leq i \leq s$
2. range $[\text{low..high}] \cdot 0 \leq \text{low} < \text{high} \leq s$

Broadly speaking, *list* provides an easy way to express the selection of more than one range of node sons. For instance, $\text{card}_s\langle[0..1], 3, [5..6]\rangle(n_1, ..., n_s)$ expresses the selection of:

– zero to one of $n_1, ..., n_s$
  or
– three of $n_1, ..., n_s$
  or
– five to six of $n_1, ..., n_s$

Formally defining the *list* extension semantics: in a valid configuration c, all non-terminal nodes $n \in (N - \Sigma)$ are labeled with the operator $\text{card}_s\langle\text{list}\rangle(n_1, ..., n_s)$ and the operator always evaluates to `true` for some $e \in$ list, according to the following rules:

– if e is an index, exactly e of the s children of n evaluate to `true`.
– if e is a range [low..high], at least *low* and at most *high* of the s children of n evaluate to `true`.

To calculate the total number of products using card⟨list⟩, equation 2 should be substituted by equation 5.

$$P(n) = \sum_{e \in \text{list}} \begin{cases} S_e & \text{if} \quad e \in \mathbb{N} \\ \sum_{k=\text{low}}^{\text{high}} S_k & \text{if} \quad e \in [\text{low..high}] \end{cases} \tag{5}$$

4 Conclusions and Future Work

Variability notations are widely used to scope the domain of a SPL. Unfortunately, there is a profusion of notations that hinders the efficient communication among specialists and the portability of variability models between tools. P. Schobbens et al. have faced this problem by proposing the pivot notation VFD$^+$ and demonstrating that many kinds of variability diagrams can be easily and efficiently translated into VFD$^+$ diagrams. Valuable information can be inferred from VFD$^+$ diagrams; for instance, the total number of products of a SPL, that is used by economic models to predict SPL costs and benefits. A common approach to infer the SPL number of products is translating diagrams into propositional logic formulas, which are processed by off-the-self tools, such as SAT solvers. However, this approach only works for small diagrams.

We think more scalable solutions can be reached avoiding the diagram-to-logic transformation and taking advantage of the tree organization of diagrams. Since VFD$^+$ diagrams are single-rooted directed acyclic graphs, we have formally defined a VFD$^+$ subset, named NFT, where diagrams are restricted to be trees. We have also demonstrated that NFT and VFD$^+$ are completely equivalent. We have proposed a time-efficient algorithm to calculate the number of products of a SPL from a NFT diagram, without considering textual constraints among nodes. The algorithm has quadratic complexity if the diagram only includes *mandatory*, *optional*, *or* and *xor* relations, and, in general, has cubic complexity for any value of VFD$^+$'s *card* operator. In fact, we have demonstrated that the algorithm also supports an extension of *card* to express disjunction of ranges. For future work, we plan to extend the algorithm to compute textual constraints.

References

1. Batory, D. *Feature models, grammars, and propositional formulas.* Proc. 9th Int. Conf. Software Product Lines, 2005.
2. Benavides, D. *On the automated analysis of software product lines using feature models.* a framework for developing automated tool support. PhD Dissertation. University of Seville, Spain, June 2007.
3. BigLever Software, Inc. *Gears.* http://www.biglever.com/index.html
4. Boehm, B.; Brown, A.; Madachy, R.; Ye Yang. *A software product line life cycle cost estimation model.* International Symposium on Empirical Software Engineering, 2004, page(s): 156- 164.
5. Clements, P. C.; McGregor, J. D.; Cohen, S. G. *The Structured Intuitive Model for Product Line Economics (SIMPLE)* (CMU/SEI-2005-TR-003).
6. Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
7. Eriksson, M.; Borstler, J.; Borg, K. *The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations.* Software Product Lines, 9th International Conference, SPLC 2005, pp. 3344.
8. Gheyi, R.; Massoni, T.; Borba, P. A *Theory for Feature Models in Alloy.* First Alloy Workshop, colocated with the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering, 2006, pages: 71-80.
9. Gomaa, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley 2004.
10. Greenfield, J.; Short, K. *Software Factories: Assembling Applications with Patterns, Models, Framworks, and Tools.* Wiley, 2004.
11. Griss, M.; Favaro, J.; dAlessandro, M. *Integrating feature modeling with the RSEB.* Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 1998, pp. 7685.
12. Harel, D. Rumpe, B. *Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff.* Technical Report MCS00-16, Faculty of Mathematics and Computer. The Weizmann Institute of Science, 2000.

13. Heymans, P.; Schobbens, P.; Trigaux, J.; Bontemps, Y.; Matulevicius, R.; Classen, A. *Evaluating formal properties of feature diagram languages*. Software, IET. Volume: 2, Issue: 3. June 2008, pp: 281-302.
14. Kang, K.; Cohen, S.; Hess, K.; Novak, W.;Peterson, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
15. Kang, K. C.; Kim, S.; Lee, J.; Kim, K. *FORM: a feature-oriented reuse method.* Annals of Software Engineering 5 (1998) 143168.
16. Metzger, A.; Pohl, K.; Heymans, P.; Schobbens, P.; Saval, G. *Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis.* Requirements Engineering Conference, 2007. RE'07. 15th IEEE International (2007), pp. 243-253.
17. Pohl, K.; Bockle, G,; Linden, F. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.
18. Pure Systems. *pure::variants.* http://www.pure-systems.com/
19. Riebisch, M.; Bollert, K.; Streitferdt, D; Philippow, I. *Extending feature diagrams with UML multiplicities.* Sixth Conference on Integrated Design and Process Technology (IDPT' 2002), Pasadena, CA, June 2002.
20. Schobbens, P.; Heymans, P.; Trigaux, J.; Bontemps, Y. *Generic semantics of feature diagrams.* Computer Networks 51(2): 456-479 (2007).
21. Steger, M.; Tischer, C.; Boss, B.; Muller, A.; Pertler, O.; Stolz, W.; Ferber, S. *Introducing PLA at Bosch Gasoline Systems.* Software Product Line Conferences 2004, Boston, MA, Aug/Sep 2004, Springer-Verlag LNCS 3154, pp 34-50.
22. Trinidad, P.; Ruiz-Cortes, A.; Benavides, D.; Segura, S.; Jimenez, A. *FAMA Framework.* 12th International Software Product Line Conference (SPLC 2008).
23. van Deursen, A.; Klint, P. *Domain-specific language design requires feature descriptions.* Journal of Computing and Information Technology (2001).
24. van Gurp, J.; Bosch, J.; Svahnberg, M. *On the notion of variability in software product lines.* Working IEEE/IFIP Conference on Software Architecture (WICSA 01), 2001.