Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models

David Fernandez-Amoros*[1], Ruben Heradio†[1], Jose A. Cerrada[1], and Carlos Cerrada[1]

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

Abstract

A software product line is an engineering approach to efficient development of software product portfolios. Key to the success of the approach is to identify the common and variable features of the products and the interdependencies between them, which are usually modeled using feature models. Implicitly, such models also include valuable information that can be used by economic models to estimate the payoffs of a product line. Unfortunately, as product lines grow, analyzing large feature models manually becomes impracticable. This paper proposes an algorithm to compute the total number of products that a feature model represents and, for each feature, the number of products that implement it. The inference of both parameters is helpful to describe the standarization/parameterization balance of a product line, detect scope flaws, assess the product line incremental development, and improve the accuracy of economic models. The paper reports experimental evidence that our algorithm has better runtime performance than existing alternative approaches.

1 Introduction

A Software Product Line (SPL)[1, 2] is an engineering approach to efficient development of whole portfolios of software products. The basis of the approach is that products are built from a core asset base : a collection of artifacts that have been designed specifically for use across the portfolio. To account for differences among the products, some adaptations of the core assets are usually required. These adaptations should both be planned before development and made easy for the product developers to use. In SPLs with large numbers of products and core assets, as well as requirements to make fine-grained adjustments, managing variability can become problematic very quickly. Mismanagement may result in adding unnecessary variability, implementing variation mechanisms more than once, selecting incompatible or awkward variation mechanisms, and missing required variations. As the product line grows and evolves, the need for variability increases, and managing the variability grows increasingly difficult [3].

To support the variability management of a SPL, the product similarities and differences are usually modeled by a Feature Model (FM) [4]. Implicitly, FMs include valuable information that can be used in economic models to estimate the payoffs of a SPL. This paper proposes an algorithm to compute the total number of products that a FM represents and, for each feature, the number of products that implement it. As it will be shown, the inference of both magnitudes is helpful to describe the standarization/parameterization balance of SPL, detect scope flaws, assess the SPL incremental development, and improve the accuracy of economic models.

*david@lsi.uned.es
†rheradio@issi.uned.es

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Several techniques have been proposed to compute the total number of products of a FM. They follow the general schema of translating the FM into a Boolean logic formula Φ, which is then processed using an off-the-self tool t, such as a SAT-solver or a Binary Decision Diagram (BDD) engine. Moreover, the number of products that implement a feature f can be computed by calling t with $\Phi \wedge f$ as input [5]. Unfortunately, this approach is computationally very expensive since it requires as many calls to t as features the FM has. The approach proposed in this paper has better runtime performance because it just requires one call to our algorithm to compute both the total number of products of a FM and, for all the features, the number of products that implement them.

Research and industry have developed different FM languages. Despite the apparent language heterogeneity, Schobbens et al. [6, 7] have formally demonstrated that most FM notations can be generalized thanks to the *group cardinality* construct[1] and the use of textual constraints (i.e., additional constraints among features specified in propositional logic). To make our approach valid for most FM languages, it deals with both group cardinality and textual constraints.

The target audience of this paper is:

1. SPL tool developers, who are interested in improving their support to decision makers. For instance, Gears[2] is a prominent software tool to develop SPLs. It includes a statistics reporting tool that *collects data from a SPL and then formulates vital statistics to report on the state, growth, and health of the SPL*. One of those statistics is an upper approximation of total number of products modeled by a FM, which does not take into account textual constraints.

2. Researchers in the field of automated analysis of FMs. Benavides et al. [10] have surveyed 53 papers focused on computing 30 analyses from FMs. Our proposal supports the computation of 8 of them: (i) *dead features* (those that, because of their excluding dependencies on other features, cannot be included in any product), (ii) *void models* (FMs that are inconsistent and thus do not model any valid product), (iii) the total number of products modeled by a FM, (iv) *feature commonality* (defined in Section 3), (v) *core features* (those that are part of all the products), (vi) *variant features* (those that do not appear in all the products), (vii) *SPL homegeneity* (defined in Section 3), and (viii) *variability factor* (the ratio between the number of products and 2^n, where n is the number of features).

3. Researchers in SPL economic models, who are interested in the information that can be retrieved automatically from a FM to improve their estimations.

4. Domain analysts who are interested in scoping a SPL adequately and estimating its payoffs.

The remainder of the paper is structured as follows: Section 2 introduces FMs. Section 3 motivates our work. Section 4 reviews in detail several approaches to the product and commonality counting problem. Section 5 presents our algorithm. Next, Section 6 describes the computational complexity of our approach and experimentally compares its runtime performance with several other algorithms. Finally, some concluding remarks are provided in Section 7.

2 Feature models

Since the first FM notation was proposed by the FODA methodology in 1990 [4], a number of extensions and alternative languages have been devised to model variability in families of related systems. Fortunately,

[1] *group cardinality* should not be confused with the concept of *feature cardinality* proposed by Czarnecki et al. [8, 9]; whereas the former describes for a group of features how many of them have to be included in a product, the latter specifies how many instances of a particular feature has to be included in a product. Our algorithm just supports group cardinality

[2] http://www.biglever.com/

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Schobbens et al. [6, 7] have demonstrated that most of the FM notations can be easily and efficiently translated into a pivot language called Varied Feature Diagram$^+$ (VFD$^+$). Schobbens et al. also showed that the automated analysis of a VFD$^+$ model can be optimized whenever the model is structured as a tree. Since VFD$^+$ models are, in general, single-rooted Directed Acyclic Graphs (DAGs), we restrict the input of our algorithm to a VFD$^+$ subset named Neutral Feature Tree (NFT) [11], where models are required to be trees.

From now on, the hypothetical FM in Figure 1 will be used as a running example. It may be part of a larger model about mobile phones. A NFT model is a hierarchically arranged set of features depicted by a tree. Each non-leaf feature has a group cardinality label [*low*..*high*] that indicates that every product includes at least *low* and at most *high* of its children. For instance, feature bluetooth is labeled with [2..3]. So, any product that includes bluetooth has to include one of the following combinations of features: {headset, hands free}, {headset, remote control}, {hands free, remote control} or {headset, hands free, remote control}. In addition, the scope of a FM can be narrowed by adding textual constraints written in propositional logic. For instance, Figure 1 has two textual constraints: 802.11n → HSDPA and 802.11n → HSDPU. Although a product with features {connectivity, bluetooth, headset, remote control, wifi, 802.11n} would satisfy the tree structure, it is not valid because it violates the textual constraints.

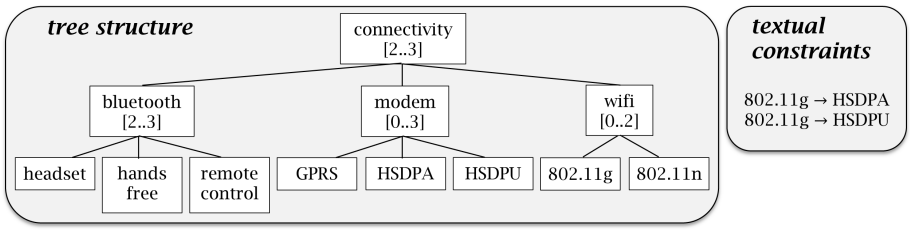


Figure 1: NFT representation of a FM for a mobile phone product line

It should be noted that NFT and VFD$^+$ are fully equivalent, i.e., any VFD$^+$ model can be converted into a NFT one by using the following transformation: whenever a VFD$^+$ model has a feature f with parents f_1, f_2, ..., f_n, its NFT representation just keeps the edge $f_1 \rightarrow f$ and encodes the remaining ones ($f_2 \rightarrow f$, ..., $f_n \rightarrow f$) as textual constraints (see Section 6.2.1 for an explanation on how to encode edges to propositional logic). Hence NFT is a generalization of most FM notations that supports:

1. The group cardinality label, which generalizes the FODA cardinalities (i.e., *mandatory* = [1..1], *optional* = [0..1], *and* = [n..n], *or* = [1..n] and *xor* = [1..1]). For simplicity, the term group cardinality will be reserved to those cases not covered by standard FODA.

2. The textual constraints, which can be any propositional logic formula and thus generalize the *require* and *exclude* dependencies proposed by Pohl [1].

Be warned that, in the appropriate context, the software characteristics in a SPL are often called features or nodes (when referring to the FM tree structure) or variables (for the textual constraints or if the FM has been translated to Boolean logic).

3 Motivation

From here on, the following notational conventions will be used:

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

- (i) $\mathcal{F}$, (ii) $\mathcal{P}$ and (iii) $\mathcal{P}_f$ denote respectively the sets of (i) features, (ii) products and (iii) products that include a particular feature f.
- $\#S$ is the cardinal of a set S. So, $\#\mathcal{P}$ is the total number of products represented by a FM and $\#\mathcal{P}_f$ is the number of products that implement a feature f. In the literature, $\#\mathcal{P}_f$ is often used in relative terms under the name of *commonality* [12, 13]; i.e., the commonality of a feature f is $\frac{\#\mathcal{P}_f}{\#\mathcal{P}}$.

From an input FM written in NFT, our algorithm computes $\#\mathcal{P}$ and $\#\mathcal{P}_f$. This section summarizes the usefulness of $\#\mathcal{P}$ and $\#\mathcal{P}_f$ to properly scope a SPL and estimate its payoffs.

3.1 Descriptive statistics to account for the standarization/parameterization balance of a FM

Domain analysis consists largely in exploring the decisions in a domain to determine which features should be common to all products and which ones variable. As Cleaveland points out in [14], this determination is not a scientific process of discovery but one of design and engineering, and it involves trade–offs among many objectives. Assigning high commonality to features means *standarizing*, which increases the common code, reduces the family size and simplifies the product platform. Assigning low commonality to features means *parameterizing*, which reduces the common code, increases the family size and increases the product platform complexity. Domain analysis attempts to achieve a good balance between these two objectives.

Since a FM models the common and variable features in a SPL, it also models the standardization and parametrization levels. However, getting such levels directly from the FM is not trivial. A naive approach to calculate them might be comparing the number of terminal features[3] that are mandatory and variable (i.e., #features with cardinality label $[n..n]$ versus #features with any other cardinality label). For instance, let us consider the "electronic shopping" FM[4], proposed in [15] to model a SPL of e–commerce systems. It has 192 terminal features: 10 are mandatory and 182 are variable. As 94.79% of the features are variable, we might conclude that the SPL is highly skewed to parametrization. However, this is not the case. The causes of such an erroneous conclusion are: (i) the FM has 21 textual constraints that have been ignored, and (ii) using just a binary distinction between mandatory and variable is too simplistic.

The commonality computation proposed in this paper supports getting a reliable quantification of the FM standarization/parametrization balance since (i) it takes into account textual constraints, and (ii) it measures commonality using a continuous range that goes from 0 (i.e., dead features) to 1 (i.e., features common to all products). Furthermore, it also supports the usage of descriptive statistics to analyze FMs. For instance, Figure 2.a is a histogram[5] that represents how commonality is distributed in the "electronic shopping" example. The histogram, as opposed to the naive approach results, shows that the standarization/parametrization levels are quite balanced: 42 terminal features have commonality between [0.4–0.5), 98 between [0.5–0.6), and 19 between [0.6–0.7), i.e., 82.8% of the features have commonality between [0.4–0.7).

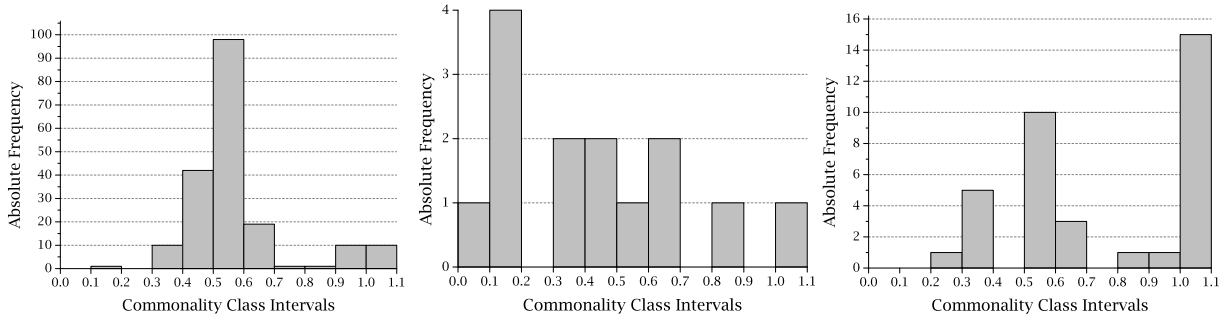
In the SPL literature, we can also find FMs which commonality is not distributed around a middle value. For example, Figures 2.b and 2.c show the commonality histograms of the FMs "Graph Product Line" [16] and "Home Integration System (HIS)" [17], which are skewed to parametrization and standardization, respectively.

[3] since the configuration of non-terminal features is entirely determined by the configuration of the terminal ones, they do not influence the number of products and as such can be ignored for model counting

[4] all the FMs referred in this paper are freely available at SPLOT website: `http://www.splot-research.org/`

[5] the left-end inclusion convention has been adopted in the histograms in Figure 2, which stipulates that a class interval contains its left-end but not its right-end boundary point

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073



(a) Electronic Shopping (b) Graph Product Line (c) HIS

Figure 2: Examples of commonality histograms

The Structured Intuitive Model for Product Line Economics (SIMPLE) [18, 19] defines a measure, named *homogeneity*, that characterizes how similar the products are. Its aim is detecting domains where products are prohibitively dissimilar from each other and thus the SPL approach does not pay off. 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. SIMPLE proposes Equation 1 to compute homogeneity, where $\#\mathcal{UF}$ is the number of unique features (i.e., features implemented by only one product). Unfortunately, this measure may produce erroneous results in some scenarios. For example, consider a SPL with 200 products and 50 features, where every feature is included in just 2 products; although the SPL is clearly quite heterogeneous, Equation 1 says that the SPL is totally homogeneous (i.e., Homogeneity$(\#\mathcal{P}) = 1 - \frac{0}{50} = 1$). Alternatively, SIMPLE proposes the more reliable Equation 2 that requires to know $\#\mathcal{P}_f$ and $\#\mathcal{P}$. Using it for the former scenario, we check that the SPL is certainly heterogeneous (i.e., Homogeneity$(\#\mathcal{P}) = \frac{\sum_{i=1}^{50} \frac{2}{200}}{50} = 0.01$).

$$\text{Homogeneity}(\#\mathcal{P}) = 1 - \frac{\#\mathcal{UF}}{\#\mathcal{F}} \qquad (1) \qquad\qquad \text{Homogeneity}(\#\mathcal{P}) = \frac{\sum_{f \in \mathcal{F}} \frac{\#\mathcal{P}_f}{\#\mathcal{P}}}{\#\mathcal{F}} \qquad (2)$$

Looking carefully at Equation 2, we realize that, in fact, SIMPLE computes homogeneity as the commonality mean (i.e., $\frac{\sum_{f \in \mathcal{F}} \frac{\#\mathcal{P}_f}{\#\mathcal{P}}}{\#\mathcal{F}} = \frac{\sum_{f \in \mathcal{F}} \text{Commonality}_f}{\#\mathcal{F}} = \overline{\text{Commonality}}$). Thanks to our proposal and developing such an idea, other measures of central tendency, such as the commonality median and mode, can be used to estimate the SPL homogeneity. Moreover, dispersion statistics, such as commonality standard deviation, can be used to understand the standarization/parametrization balance.

3.2 Detection of problematic features

According to Czarnecki et al. [20], feature modeling helps to avoid two serious problems: (i) relevant features and variation points are not included in the reusable software, and (ii) many features and variation points are included but never used and thus cause unnecessary complexity and increase development and maintenance costs. Commonality helps identifying low reusable features by looking at $\#\mathcal{P}_f$ in relative terms. For instance, in the "electronic shopping" example there is a feature named "enable profile update on checkout" with $\#\mathcal{P}_f = 3.93 \times 10^{48}$. It seems a core feature that will be reused throughout most of the domain products. Nevertheless, computing its commonality $\frac{3.93 \times 10^{48}}{2.26 \times 10^{49}} = 0.17$ we realize its reusability is actually low.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

In addition, it is possible to detect scope flaws in the early stages of the SPL development by enriching $\#\mathcal{P}_f$ with economic information. Economic models proposed in [21, 22, 23, 24, 25] use two abstractions, called the Relative Cost of Reuse (RCR) and the Relative Cost of Writing for Reuse (RCWR). 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. Of course, we need to pay that extra 20% every time we reuse it. On the other hand, 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 to have a more generic design, additional documentation...) then RCWR = 1.5. Note that RCWR is always greater or equal to 1. Poulin [24] 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 3. Therefore, a feature f causes a scope problem whenever Inequality 4 is satisfied.

$$\text{Payoff Threshold}_f = \frac{\text{RCWR}_f}{1 - \text{RCR}_f} \quad (3) \qquad \#\mathcal{P}_f < \text{Payoff Threshold}_f \quad (4)$$

3.3 Assessment for SPL incremental development

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 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 [26] 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. Although Cohen "just expresses DOR in English", such concept can be formalized with Equation 5, that requires to know $\#\mathcal{P}_f$, where:

1. Size(f) is the size of the software that implements the feature f (a number of techniques to estimate software size are presented in [27]).
2. 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 (Size(f)) by the number of times that software is reused ($\#\mathcal{P}_f$).
3. The numerator is the size of the all the software that is made by reusing core assets.

$$\text{DOR}(\#\mathcal{P}) = \frac{\sum_{f \in \text{CAB}} (\text{Size}(f) \times \#\mathcal{P}_f)}{\sum_{f \in \mathcal{F}} (\text{Size}(f) \times \#\mathcal{P}_f)} \quad (5)$$

3.4 Improving the accuracy of ROI estimations

The computation of $\#\mathcal{P}_f$ may also support increasing the accuracy of existing economic models for SPLs, such as the COnstructive Product Line Investment MOdel (COPLIMO) [21, 28], which estimates the SPL Return On Investment (ROI) by *analogy*. That is, COPLIMO starts estimating the development costs of a particular product, i.e., a domain *stereotype*. Then, supposing that all products in the scope of a SPL are

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

quite similar, it extrapolates the costs of the stereotype to compare the costs of building all the products under a SPL approach versus building them one by one. Unfortunately, whenever the products are not extremely homogeneous, COPLIMO's simplifying assumptions produce problematic distortions in the estimates. As Appendix B shows, such assumptions are unnecessary when $\#\mathcal{P}_f$ is known. In that case, ROI can be more accurately estimated by *composition*, i.e., estimating the costs for each feature and then calculating the costs of each product by adding the costs of the features that it includes.

4 Related work on product counting

There are two main approaches to perform automatic analysis of FMs:

1. A purely logic approach, where the whole FM is translated to Boolean logic and then some standard procedure is applied, such as a SAT solver or a Constraint Satisfaction Problem (CSP) solver.

   A SAT solver is a program that tries to determine if a Boolean formula is satisfiable. A #SAT model counter is a program that tries to determine how many models (i.e. how many satisfying assignments) a formula has[6]. While the SAT problem is known to be NP-complete [29], it is widely believed that the #SAT problem is even harder [30]. Section 6 experimentally compares our approach versus two prominent #SAT counters: *relsat* [31] and *cachet*[7] [33, 34]. It is worth mentioning that although [35] claims that *SAT-based analysis of FMs is easy*, #SAT-based analysis of FMs is much harder and commonality-based analysis of FMs forcing features one by one is highly impractical as will be shown in the experimental evaluation section.

   CSP solvers are also commonly used for FM analysis [36, 37]. The CSP problem involves a set of variables over a domain and a set of constraints over these variables. CSPs work by searching the solution space performing constraint satisfaction and backtracking. There is no advantage to using a CSP solver for a #SAT problem since the CSP techniques restricted to the Boolean case are equivalent to those in SAT and there is an obvious performance overhead for CSP solvers. For that reason they will not be considered for comparison purposes.

2. A hybrid approach, which manipulates the textual constraints and then runs some *ad-hoc* treatment for the tree structure of the FM. These systems rely on the tree structure for good performance, so they are not usually designed to gracefully handle the textual constraints. Some preprocessing may help though. Czarnecki & Wasowski [38] propose to follow the path of translation in the opposite direction, that is, to integrate the textual constraints into the tree. The algorithm is not always able to integrate all the constraints, but a partial result may be of help to a hybrid system in some cases. Another remedy is proposed in [39], where it is proven that all textual constraints can be eliminated at the price of building a new set of features. The approach is theoretical, but the bases for an implementation are laid out. In [13], Mendonça presents a system named Feature Model Reasoning System (FMRS) that depends on a reasoning engine to solve the textual constraints while another module works on the FM. There are several similarities between our proposal and FMRS, since we also apply a reasoning engine and separately process the feature tree. There are also several differences. Our prototype, which we have named *treecount*, uses DPLL [40] as a reasoning engine, while FMRS

[6] a simple SAT solver can be easily modified to act as a model counter, and even as an explicit model generator. Nevertheless, as SAT solver techniques grow increasingly specialized, they become useless for these other problems, which demand their own techniques. For instance, it is now customary for solvers to implement timed restarts; if no answer to the SAT problem is found, then the search is interrupted and continued elsewhere. For satisfiable cases, it suffices to find one model, so the technique seems to speed up the process. However, it does not carry over to efficient counting or enumerating of the models

[7] there is also a variation of cachet called sharpSAT [32], but since the changes affect only cache management and not the search process it will not be addressed in this paper

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

relies on a general constraint solver. FMRS performs constraint propagation inside the tree and also relies on a system of saving and restoring states to help with the backtracking. Our treatment of the FM is more akin to a tree traversal whilst an efficient cache system exploits the bigger size of the model w.r.t. the textual constraints to avoid repeating the same computations over and over as will be explained in the next section. We also perform *Unit Propagation* (UP[8]) [30] over a translation of the FM, so that treecount can finish the search faster. Also, FMRS does not support commonality calculations. The analysis tool in the SPLOT[9] [41] portal uses Reduced Ordered Binary Decision Diagrams (ROBDDs) [42], which are just a special case of Binary Decision Diagrams (BDDs), to compile information about the FM. These decision diagrams hinge on a fixed order in which the variables can be inspected. The BDD represents both the tree and the textual constraints, following an ordering of the variables that has been previously determined by a complex heuristic [43]. Ordering heuristics can take remarkably long, and it is known that finding an ordering that produces an optimal BDD (i.e. a BDD with the minimal number of nodes) is an NP-complete problem [44]. The BDD is then traversed to compute the number of products. As far as efficiency is concerned, Mendonça et al. make a big selling point of the space gains of the BDD vs. DPLL. However, DPLL being a backtracking search, it only keeps in memory the current stack of search nodes. The main reason for efficiency of BDD counting is not only that exploration starts from the *true* node (and thus avoid exploring unsatisfiable branches), but also and importantly so, the efficiency comes from use of dynamic programming, which exploits the sharing of nodes, so that the search paths are not explored separately. As far as commonality is concerned, splot does not support its computation, although it is certainly possible to adapt the *traverse* algorithm in [42] to do so.

In previous work [11], we presented an algorithm to calculate $\#\mathcal{P}$ and $\#\mathcal{P}_f$. It is noteworthy that the algorithm is always exponential in the number of clauses of the textual constraints, not just in the worst case. So, the treatment of the textual constraints renders the algorithm impractical for all but laboratory-sized FMs.

As far as the strengths and weaknesses of the approaches are concerned, component-based model counters such as cachet and relsat are expected to be very efficient when the textual constraints present independent clauses (clauses with no variables in common) since DPLL can efficiently cut the translated formula into many different connected components, and very inefficient when the FM heavily exploits proper group cardinality, since the translation to Boolean logic suffers from a combinatorial explosion. This may also be true for splot, since these independent clauses should produce an additive-like growth in the number of nodes rather than an exponential one. The concept of branch width [45], introduced by Bacchus et al., accurately captures this behavior since they proved that there exists a model counting algorithm which runs in time exponential to the branch width. Comparatively, the rest of the hybrid approaches (FMRS and treecount) are expected to perform poorly in relative terms when textual constraints are mostly independent (i.e. the formula has a low branch-width). In our case, it is because the DPLL search will produce the same subsequences of branching variables over and over. The same problem is expected to happen to FRMS since enumerating the solution sets can effectively cancel out the benefits of a compact BDD.

As a quick recap, there are several working approaches to perform model counting, but commonality computing has been more of a theoretical issue so far. Any model counting scheme can be used to compute commonalities by *forcing* the presence of a particular feature. This can be an effective approach for a

[8] UP can be applied to an unsatisfied clause in which there is exactly one unassigned literal, to derive a satisfying assignment for the variable involved. For instance, if there is a partial truth assignment α where x =true, and y =false, and a clause $\neg x \vee y \vee \neg z$, then UP applied to the clause yield z =false, thus satisfying the clause. UP is a relatively inexpensive way of saving branching decisions during DPLL search and so both techniques are often employed together

[9] We reserve the capitalized version *SPLOT* for the portal and simply use *splot* for the model counter

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

lonely feature, but when all the individual commonalities are needed, this *naive* approach can impose a considerable overhead.

5 Fast commonality counting

This section presents our algorithm, treecount, which takes a FM as input and computes the number of products and the commonality of each feature as output. A pseudocode description of the algorithm is included in Appendix A.

5.1 Solving the textual constraints

The first step of our algorithm is solving the textual constraints of the FM, which must be written in *Conjunctive Normal Form* (CNF[10]), that is, a conjunction of disjunctions of *literals* (a literal is a proposition or its negation). Such disjunctions are called *clauses*, so a CNF formula is a conjunction of clauses. We use a variation of the standard DPLL procedure as depicted in Algorithm 1. The core of our contribution is the Function *computeProducts* that will be described in Figure 5 and presented in a top-down fashion throughout this Section. DPLL has been thoroughly researched by the SAT-solver and model-counting community [47, 48, 49, 50, 51, 33, 34, 31], so we have adopted it as is. Our heuristic in choosing the next variable to be conditioned (i.e. to continue the backtracking) is to take the feature appearing most often in the residual formula, with a preference for smaller clauses to break ties. This is known as *Mom's heuristic* (Maximum Occurrences in clauses of Minimal Size)[52]. This heuristic has proven to be very effective among the different alternatives [53]. It is noteworthy that slight changes in this heuristic may dramatically affect the number of solution sets produced. Of course, efficiency-wise, the least solutions sets, the better. Unlike splot, which first computes the whole BDD, we build the solution sets iteratively.

Both cachet and relsat also employ a typical SAT solver technique called conflict-driven *clause learning*. Whenever a conflict is detected and before backtracking is performed, a new clause is added to the original formula. Since UP is not resolution-complete, it is possible that a series of decisions about branching variables produce a conflict much later in the DPLL search. This series of decisions may be repeated over and over during the search. To avoid this, a new clause that captures the essence of the conflict is added to the original formula so that if the series of decisions that led to the conflict are taken again, the conflict is immediately discovered. We have decided not to include clause learning in our algorithm proposal, due to the fact that in real FMs (as opposed to randomly generated) there are hardly any conflicts. The reason is that the tree structure of a FM is always satisfiable, so any conflict must come from the textual constraints, of which there are usually too few to cause it. This is in stark contrast with SAT-solver test-beds, which are often orders of magnitude larger than FMs, where conflicts are commonplace.

First we will see how to compute $\#\mathcal{P}$ to show some common problems and their possible solutions. Let us consider the example in Figure 1. The two textual constraints translate into CNF as $(\neg 802.11n \vee HSDPA) \wedge (\neg 802.11n \vee HSDPU)$. DPLL is used to build a set of solutions. Since *802.11n* is the most common feature in the textual constraints, it is chosen to start out backtracking. We assign *802.11n* the value *false*. Now, since both constraints are satisfied, we backtrack and flip the value of *802.11n*, which is now *true*. Then, by UP in the first clause, *HSDPA* must be true, and by UP in the second clause, *HSDPU* must also be true. All the clauses are satisfied. So, the solution sets for this example are $\{\neg 802.11n\}$ and $\{802.11n, HSDPA, HSDPU\}$. To summarize the solution sets the following notation is used: a literal with a negation sign in front of it represents a feature whose truth value is *false*, the ones without it are assigned

[10] CNF is the standard form that Boolean formulas must follow to be processed by SAT-solvers. It is always possible, though not necessarily efficient, to translate a formula into CNF [46]

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

true and the rest of the features, not shown, are *unassigned*. The *unassigned* value means that either true or false comply with the constraints.

5.2 Classifying the nodes

The structure of the tree plus a particular solution set may impose that a particular node (i) be present in all the corresponding products (*present*), (ii) be absent from all the products (*absent*), (iii) be in some products and not in others (*potential*) or (iv) be and not be in all the products at the same time (*contradicting*). The type of a node can be defined in terms of two other concepts: *selected* and *deselected*. A node will be selected if it is necessary that the node is present in all the products to comply with the solution set plus textual constraints and it will deselected if it is necessary that the node is absent from all the products. Combining these values all four possibilities are covered. Let us focus mostly on present and potential nodes, since absent and contradicting nodes play a small part. Provided that there are no contradicting children, the *Inner Variability* (IV) of a node (i.e. the different number of valid configurations for its subtree) will depend on the present and potential nodes. The present nodes will provide a present factor and the potential nodes a potential factor. The product of both factors will yield the IV of the node.

Formally, the solution set can be considered as a function $\alpha : A \rightarrow \{\text{true}, \text{false}, \text{unassigned}\}$, where A is the set of nodes. So, consider a node n, with s (possibly zero) children: $n_1, n_2, \ldots, n_s$ and a cardinality relation [*low*..*high*] meaning that at least *low* children and at most *high* children must belong to a product. Suppose the children have already been classified and thus the number of present, potential and contradicting nodes, which are denoted respectively by $\mid \text{pre}(n, \alpha) \mid$, $\mid \text{pot}(n, \alpha) \mid$ and $\mid \text{con}(n, \alpha) \mid$, are available. Then, $\text{sel}(n, \alpha) \equiv (\alpha(n) = \text{true}) \vee \bigvee_{i=1}^{s} \text{sel}(n_i, \alpha)$ and

$$\text{desel}(n, \alpha) \equiv (\alpha(n) = \text{false}) \vee (\mid \text{pre}(n, \alpha) \mid + \mid \text{pot}(n, \alpha) \mid < \textit{low}) \vee (\mid \text{pre}(n, \alpha) \mid > \textit{high}) \vee (\mid \text{con}(n, \alpha) \mid) > 0$$

It is worth noting that the definition is not circular; the type of node can be determined by Algorithm 2, and the needed attributes are computed in a bottom-up fashion. In the NFT syntax (which was introduced in Section 2), the sel property has a strictly logical interpretation; if $\text{sel}(n, \alpha)$, then n is logically implied by the solution set and the tree structure (since the children imply the parent). For desel, the question is more complex, since the cardinalities are also involved. Essentially, $\text{desel}(n, \alpha)$ means that the node cannot be present in this solution set, be it because it is explicitly negated or because it is impossible for it to comply with the cardinality restrictions. For each node n, an attribute m will be used, and set to true iff the node is a mandatory feature w.r.t its parent. The reason is that mandatory nodes are always selected. Let us consider now the computeType function in Algorithm 2. The way to call it for a node n would be ComputeType(n.sel $\vee$ n.m, n.desel).

Back to the example, in the first solution set, *802.11n* is absent and everything else is potential. In the second set, *802.11n*, *HSDPA* and *HSDPU* are directly present, *wifi*, *modem* and *connectivity* are also present given that the sel attribute is synthesized upwards. The rest are still potential.

5.3 Computing variability

The *IV* of a node n (under a solution set), denoted as $IV(n)$, is the number of different valid configurations in the subtree rooted at n, according to the tree structure. Thus, the total number of products represented by a FM for a particular solution set is *IV(r)*, where r is the root. For a leaf node l, $IV(l) = 1$, as long as the node is not negated in the solution set. For a non-leaf node n with s children, $n_1, n_2, \ldots n_s$, the formulas to compute $IV(n)$ for the FODA type of features are:

1. *mandatory/optional*: $IV(n) = \prod_{n_i \text{ is optional}} (IV(n_i) + 1) \cdot \prod_{n_j \text{ is mandatory}} IV(n_j)$
2. *or* (i.e., $\text{card}_s[1..s]$): $IV(n) = (\prod_{i=1}^{s} (IV(n_i) + 1)) - 1$

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

3. *xor* (i.e., $\text{card}_s[1..1]$): $IV(n) = \sum_{i=1}^{s} IV(n_i)$

In general, when a node n has s children and has [*low*..*high*] cardinality, $IV(n)$ is given by Equation 6, where S_k is the variability in choosing any combination of k children from s. For the sake of clarity, let us denote $IV(n_1), IV(n_2), \ldots IV(n_s)$ as $\text{iv}_1, \text{iv}_2, \ldots, \text{iv}_s$. In a straightforward approach, S_k can be calculated by summing the variabilities of all possible k-combinations (see Equation 7). Unfortunately, this calculation has exponential complexity.

$$IV(n) = \sum_{k=low}^{high} S_k \qquad (6) \qquad\qquad S_k = \sum_{1 \le i_1 < i_2 < i_3 \ldots < i_k \le s} \text{iv}_{i_1} \text{iv}_{i_2} \ldots \text{iv}_{i_k} \qquad (7)$$

A better complexity can be achieved making use of recurrent relations. The base case is $S_0 = 1$. According to Equation 7, $S_1 = \sum_{i=1}^{s} \text{iv}_i$. Calculating S_2, the variability for combinations of 2 siblings that include n_1 is $\text{iv}_1\text{iv}_2 + \text{iv}_1\text{iv}_3 ... + \text{iv}_1\text{iv}_s = \text{iv}_1(\text{iv}_2 + \text{iv}_3 + ... + \text{iv}_s) = \text{iv}_1(S_1 - \text{iv}_1)$. Likewise, the variability of 2-combinations that include n_2 is $\text{iv}_2(S_1 - \text{iv}_2)$ and so on. Adding up all these 2-combinations, we get $\sum_{i=1}^{s} \text{iv}_i(S_1 - \text{iv}_i)$. However, in the sum each term $\text{iv}_k iv_l$ has been counted twice; once in the addend in which $i = k$ and then again in the $i = l$ addend. Thus, if we adjust for this duplication: $S_2 = \frac{1}{2}\sum_{i=1}^{s} \text{iv}_i(S_1 - \text{iv}_i) = \frac{1}{2}(S_1 \sum_{i=1}^{s} \text{iv}_i - \sum_{i=1}^{s} \text{iv}_i^2) = \frac{1}{2}(S_1^2 - \sum_{i=1}^{s} \text{iv}_i^2)$

Computing S_3, the variability for combinations of 3 siblings that include n_1 is iv_1 multiplied by the variability for 2-combinations that do not contain n_1, i.e., $\text{iv}_1(S_2 - v_1(S_1 - \text{iv}_1))$. Adding up every 3-combinations: $\sum_{i=1}^{s} \text{iv}_i(S_2 - \text{iv}_i(S_1 - \text{iv}_i)) = S_2S_1 - S_1 \sum_{i=1}^{s} \text{iv}_i^2 + \sum_{i=1}^{s} \text{iv}_i^3$

This time, every triple $\text{iv}_k\text{iv}_l\text{iv}_m$ is being counted three times. Hence, removing the redundant computations: $S_3 = \frac{1}{3}\left(S_2S_1 - S_1 \sum_{i=1}^{s} \text{iv}_i^2 + \sum_{i=1}^{s} \text{iv}_i^3\right)$

Equation 8 shows the general formula for S_k. Though it is not trivial just by looking at the formulas, we will prove in Section 6 that $IV(n)$ can be computed in time quadratic to the number of children of n, which constitutes a considerable improvement from exponential to very low polynomial complexity. Algorithms 3, 4 and 5 implement Equation 8.

$$S_0 = 1; \quad S_k = \frac{1}{k} \sum_{i=0}^{k-1} ((-1)^i S_{k-i-1} \sum_{j=1}^{s} \text{iv}_j^{i+1}) \text{ for } 1 \le k \le s \qquad (8)$$

Let us consider again the simple FM in Figure 1. As an example, let us compute the variability for *connectivity* using Equation 8 for the first solution set $\{\neg 802.11n\}$. This can be done because all the children of *connectivity* in this solution set are potential, so present factor has value 1. To do that, it is needed to know in advance the variabilities of *bluetooth*, *modem* and *wifi*. Equation 8 could be used again, but instead and to see how much effort it takes, we will do it in the inefficient way. Table 1 summarizes the possibilities to choose children for each node: 4 for *bluetooth*, 7 for *modem* and 1 for wifi (since *802.11n* is false). It is easy to see that if the number of children is big, the equivalent to Table 1 can get really long.

We now compute the powers of the number of products from the children of *connectivity* and their sum (see Table 2). Now, $S_0 = 1$ by definition, $S_1 = 12$, as it is the sum of the children variabilities, $S_2 = 1/2(12 \cdot 12 - 66) = 39$, following the general formula 8 and $S_3 = 1/3(39 \cdot 12 - 12 \cdot 66 + 408) = 28$. Adding up S_2 and S_3, we get 67.

In the second solution set, nothing has changed in the *bluetooth* subtree, so the old variability values could be used again. *modem* and *wifi* are now present nodes, each with variability 2 and 1 respectively, so the present factor is 2 and the potential factor consists of taking *bluetooth* [0..1] times, which gives us 5. The product of both factors yields the variability of *connectivity* for this set, 10. Adding up both sets, there are 77 products in this product line.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Feature	Admissible children combinations for non-leaf nodes
connectivity	{bluetooth, modem}, {bluetooth, wifi}, {modem, wifi}, {bluetooth, modem, wifi}
bluetooth	{headset, hands free}, {headset, remote control}, {hands free, remote control}, {headset, hands free, remote control}
modem	{GPRS}, {HSDPA}, {HSDPU}, {GPRS, HSDPA}, {GPRS, HSDPU}, {HSDPA, HSDPU}, {GPRS, HSDPA, HSDPU}
wifi	{802.11g}, {802.11n}, {802.11g, 802.11n}

Table 1: Cardinality restrictions for the mobile phone example

power	bluetooth	modem	wifi	sum
1	4	7	1	12
2	16	49	1	66
3	64	343	1	408

Table 2: Variability powers from *connectivity* children and their sum

5.4 Propagating contextual variability

IV is enough to compute $\#\mathcal{P}$ since $\#\mathcal{P} = \mathrm{IV}(r)$, where r is the root feature. To compute $\#\mathcal{P}_f$ it is necessary to introduce the concept of *Contextual Variability* (CV). Back to the FM in Figure 1, suppose that $\#\mathcal{P}$ and $\#\mathcal{P}_f$ have already been computed. Now imagine a new root node called *mobile phone* with *connectivity* as a mandatory child and another optional child called *USB* are added. Since the new nodes do not occur in the textual constraints, it is obvious that the new FM has $77 \times 2 = 154$ products. That is because for each product in the original FM, there is a new one with *mobile phone* added, and another one with *mobile phone* and *USB* added. The CV of connectivity is precisely 2, because each product in the *connectivity* subtree is expanded into 2 different products in the bigger FM. Even more, the number of products each feature appears in, which we keep to compute the commonality of each feature, has also doubled. Intuitively, the CV of *connectivity* propagates down its whole subtree. One way to compute this CV is to mentally delete *connectivity* and compute what the IV of its parent would be. In general, it is a process that has to be performed at the solution set level.

Given a solution set, the CV for a particular node n, is the number of different configurations for the SPL ignoring the subtree rooted in n. Clearly, every product can be decomposed as the part that depends on the features in the subtree of n and the part that depends on the rest of the features. So, the number of products that n appears in for a particular solution set is the product of its IV and its CV. CV is a global attribute that depends on the whole outside context of the node. Obviously, CV for the root feature is 1, since there is no context. In the general case, we take advantage of the fact that CV propagates downwards rather easily. For a non-root node n with parent p, given a solution set, $\mathrm{CV}(n) = \mathrm{CV}(p) \times \mathrm{IV}(p')$, where p' is a copy of p where the subtree under n has been deleted. We could also say that $\mathrm{IV}(p')$ is the *Sibling Variability* (SV) of n, $\mathrm{SV}(n)$, as we call it in Algorithm 6. Instead of computing $\mathrm{IV}(p')$ from scratch, which would take time quadratic to the number of siblings, some previous results are reused to obtain it in linear time.

Let n be a node, with s children whose inner variabilities are respectively iv_1, iv_2, ..., iv_s, and let us suppose $IV(n)$ has been already computed using Equation 8. This calculation would provide us with vector S. What would happen if we should add a new child n_{s+1} with variability iv_{s+1}? a new vector S' may be computed using the general Equation 8, 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 $\mathrm{iv}_{s+1} \cdot S_{i-1}$. So, $S'_i = S_i + \mathrm{iv}_{s+1} \cdot S_{i-1}$.

To calculate the CV of a child m of n, what we really want to do is exactly the opposite, i.e., having computed S_i, take out m and compute the vector S'_i, i.e., $S'_0 = 1$ and $S'_i = S_i - \mathrm{iv}_m \cdot S_{i-1}$

Going back to our previous example, say we want to take out feature *wifi* in order to compute its CV. Now $S'_0 = 1$ by definition, $S'_1 = 12 - 1 \cdot 1 = 11$, $S'_2 = 39 - 1 \cdot 11 = 28$ and $S'_3 = 28 - 1 \cdot 28 = 0$ (as expected, since there are only two siblings left). *Connectivity* is the root node, so its CV is 1. This means that *wifi* appears in $39 \cdot 1 = 39$ products in this solution set. As *802.11n* is false and *802.11g* is a leaf node, initially

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

IV(802.11g) = 1. If the CV for *wifi* is propagated, then CV(802.11g) is 39, so, *802.11g* appears in 39 products in this solution set.

5.5 Processing a solution set

Let us discuss the algorithm formally. The attribute grammar formalism will be used to avoid the control-flow overhead of traditional pseudocode (i.e., to keep the algorithm clear of code regarding the FM tree traversal). Figure 5 summarizes the productions of the context-free grammar, together with the semantic rules. The terminal symbols are presented in boldface. A syntactic tree of this grammar corresponds exactly to a FM, and each node has a set of attributes, whose values produce the needed information. But before we explain how to get it, let us briefly discuss how to encode FMs in the different syntactic alternatives to suit the grammar.

In NFT there are two types of nodes: group nodes and leaf nodes. Encoding a NFT FM into a string produced by this grammar is straightforward, all that is necessary is to ignore the *mandatory* construction. Regular FODA does not use group cardinality. Suppose there is a node n with s children, of the optional/mandatory type, it can be encoded as a group node with cardinality [0..s] and then use the mandatory attribute appropriately for the children. If n is an *or* node, then it would be a group node with cardinality [1..s] and no mandatory children. Likewise, a *xor* node would require cardinality [1..1] and no mandatory children. The SPLOT metamodel uses group cardinality and the optional/mandatory construction, so it also follows the former encoding.

The first thing to notice when using the grammar is the traversal order for the nodes. Although most attributes are synthesized, which implies that a bottom-up traversal of the syntactic tree is necessary, there are also some inherited attributes, so another top-down traversal will also be necessary. The good thing is that there is no circularity in the definitions. For the sake of clarity, before each definition of the attributes, we have appended an upwards arrow for synthesized attributes, and a downwards arrow for inherited ones.

Let us describe how the evaluation would take place. In the first, ascending phase, the IV is computed for every node. In attribute grammars, the notation for an attribute a of a node n is $n.a$, so IV(n) would be written as *n.iv*. So, we count how many children of each type a node has with the *c* attribute. Then *sel* and *desel* attributes are computed. With these values, together with the mandatory attribute, the node type is computed out of the four possibilities. All the present children variabilities are integrated in the present factor (*pref*) and the variability of potential children are added in the list attribute *pot*. Then, its potential factor is computed via gCard, which also provides vector S. An attribute called *extra* is used to hold the values of the number of products of F, the cardinality bounds and also the vector S, since all these may be used as input parameters for Algorithm 6. Finally, the IV is computed. This is done for each node in a bottom-up fashion.

In the descending phase, the CV is computed for each node using the CV of the parent and *taking the node out*. If the node was present, the process is as simple as dividing the variability of the parent by the variability of the child. If the node was potential, we employ the vector S and the node variability to call Algorithm 6. Either way, we can compute now the number of products the node appears in, and this amount is added to an external array adding up the subtotals along the solution sets. The CV attribute is also used in a slightly subtle way: when a node is deselected, none of its descendants participates in any products, so we set the CV to zero and just let it propagate downhill, so all the descendants end up with zero products.

Finally, a small optimization has been added consisting of caching the results of the computation for each node, so that a node whose attributes are already cached is just visited once. For each node n that is visited, a key is hashed consisting of the name of the node plus the values of the variables in the textual constraints that are also descendants of n. Typically the key is very small since the ratio of variables in

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

the textual constraint versus the total number of variables in the FM is usually low, as we will discuss in Section 6, so this allows us to reduce the runtime of treecount.

6 Computational complexity and experimental evaluation

6.1 Computational complexity

This section shows that Algorithm 1 is quadratic in the number of features and exponential in the number of variables in the textual constraints. So, if N is the number of features in the tree and M is the number of features in the textual constraints, the complexity is in $O(N^2 2^M)$.

For practical purposes it is convenient to introduce the concepts of Extra Constraint Representativeness (ECR) and Clause Density (CD), introduced in [13]. ECR is the number of variables in the textual constraints divided by the total number of features. In the mobile phone example, this would be $\frac{3}{12} = 0.25$. CD is the number of clauses in the textual constraints divided by the number of variables in the textual constraints. For the mobile phone example, it is $\frac{2}{3} = 0.67$. The DPLL process is exponential by nature, so in the worst case it may have to choose and flip almost every variable in the textual constraints to get all the solution sets.

DPLL backtracking search can be seen as a binary tree, with 2^M leaf nodes and 2^{M-1} internal nodes. Following Algorithm 1, the leaf nodes correspond to the case where the textual constraints are satisfied and computeProducts (Table 5) is called. The internal nodes generally need to perform UP and then choose a new variable to keep the backtracking. Of these elements, it is clear that UP and branch selection via Mom's heuristic can be completed in quadratic time. We now prove that processing each solution set, that is, running computeProducts, takes only $O(N^2)$ (in fact, $O(N)$ if only standard FODA models are allowed). For clarity's sake, it may help to consider the operations step-by-step. For a call to #gCard (Algorithm 3) with a node n with s children, the result is in $O(s^2)$, since there are two nested loops upper bounded by s. When #gCard is called for all the nodes, as it is done in computeProducts, it takes $O(N^2)$, where N is the total number of nodes. This can easily be proven by means of structural induction: the leaf-nodes of the FM 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 tree with children $n_1, n_2, \ldots n_s$ with $N_1, N_2, \ldots, N_s$ nodes in their respective subtrees (excluding the root). Now, $N = (\sum_{i=1}^{s} N_i) + s + 1$. Trivially, $\sum_{i=1}^{s} x_i^2 \leq (\sum_{i=1}^{s} x_i)^2$ for any natural number x_i. The induction hypothesis is that $\#\text{gCard}(n_i) \in O(N_i^2)$. So, the time spent computing $\#gCard$ for all the nodes is delimited by Equation 9.

$$\left(\sum_{i=1}^{s} N_i^2\right) + s^2 \leq \left((\sum_{i=1}^{s} N_i) + s\right)^2 \leq N^2 \in O(N^2) \tag{9}$$

The rest of the computations (sel, desel, ...) are linear. Therefore, the bottom-up sequence of computeProducts is quadratic. Finally, Let us consider the top-down sequence. The call to TakeOneOut for some node n_i takes time in $O(N_i^2)$ ($O(N_i)$ for regular FODA). Applying again the argument expressed in Equation 9, this top-down sequence is $O(N^2)$. Therefore, the sequence of the operations in computeProducts is $O(N^2)$.

The total cost of Algorithm 1 includes calling computeProducts ($O(N^2)$) once for each DPLL search leaf node, of which there are 2^M and performing in sequence UP ($O(N^2)$) and branch selection (also $O(N^2)$) for each of the 2^{M-1} internal nodes. That is $O(2^M N^2 + (2^{M-1})(N^2 + N^2)) = O(N^2 2^M)$. Calculation of the commonalities just takes computing the products for each feature and then traversing all the features to perform a division by the total number of products, so the complexity for commonality computing is the same as in Algorithm 1.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Model counting via strict DPLL as performed by cachet and relsat belongs to the complexity class $N^{O(1)}O(2^N)$. The reason is that every feature appears as a variable in the corresponding formula, regardless of the ECR, so every variable counts for the exponential part. The branching heuristics can also be computationally expensive. The complexities for the forcing versions would be the same since basically they consist in calling again the tool N times, once for each feature to force it to be present. Splot's complexity is in the class $O(2^N)$, because the branching strategy is defined by the ordering constraints instead of being reconsidered at each new step as cachet and relsat do. For commonality counting, it is $O(N \cdot 2^N)$ because the BDD is traversed once for each feature, and the BDD may have as many as 2^N nodes. So, as far as worst-case complexity is concerned, our approach moves a great deal of the complexity from the exponential side to the quadratic one. The different complexities are summarized in Table 3.

	Product counting	Commonality counting
cachet	$N^{O(1)}O(2^N)$	$N^{O(1)}O(2^N)$
relsat	$N^{O(1)}O(2^N)$	$N^{O(1)}O(2^N)$
treecount	$O(N^2 \cdot 2^M)$	$O(N^2 \cdot 2^M)$
splot	$O(2^N)$	$O(N \cdot 2^N)$

Table 3: Complexity classes for a FM with N features M of which appear in the textual constraints (M < N)

6.2 Experimental evaluation

In addition to the theoretical evaluation that computational complexity provides, five experiments have been devised to evaluate the validity and scalability of the different approaches on datasets with varying characteristics; with and without group-cardinality, real academic models and computer-generated ones, with and without textual constraints and some others with different clause density values. These experiments will show how treecount[11] behaves w.r.t to the other baselines: the propositional-logic exact model counters cachet [33] and relsat [31], and the open-source version of splot [41]. It is important to note that cachet, relsat and splot needed some tweaking in order to perform commonality counting, since only treecount does so natively: For cachet and relsat, commonality computing requires one run for each feature with an additional constraint to *force* the presence of that feature. The number of products containing the feature is then divided by the number of products containing the root feature to obtain the commonality. Cachet [54] presents a pseudocode to compute the marginals, but the memory requirements multiply those of the counting version, which were already exponential. Where the counting version caches a floating point number, the marginality computing one stores a whole vector of those. The authors claim computing marginals is 10-40% slower than counting, when the problem fits in memory. Lacking evidence to the contrary, it is reasonable to assume that only small models do fit in memory (and those that do not fit in memory quickly degrade into good-old DPLL search). This version of cachet is not available, so, as with relsat, cachet was used as a black-box to force each feature in turn. With splot, the original steps of computing a variable order and building the BDD were followed. After that, the BDD graph was traversed once for each feature to force its presence and thus compute its commonality.

The test machine was an Intel core i7 @3.5Ghz with 16GB of RAM running Mac OS X.

6.2.1 First experiment. Group cardinality #1

The first experiment seeks to test the ability of the different approaches to deal with a sample of group cardinality FMs. The FM scheme is summarized in Figure 3. The FMs consist of a root node n, with s

[11] the prototype implementation of our algorithm, the experiments described in this section and a number of case studies are available on https://sourceforge.net/projects/commonality-spl

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

terminal children and cardinality [h..h+1], where h is the integer division of s by 2. The results of the experiment are summarized by Table 4.

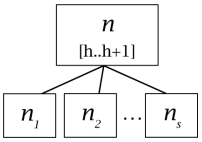


Figure 3: Hard group-cardinality model

terminal nodes	clauses	cachet		relsat		treecount		SPLOT	
		time	dec	time	dec	time	dec	time	BDD size
1	4	9.49	0	0.59	2	0.01	0	48.69	2
2	6	12.27	0	0.63	6	0.01	0	51.99	3
3	8	15.38	2	0.86	16	0.01	0	55.16	6
4	12	18.75	10	1.15	38	0.01	0	58.31	9
5	18	22.06	27	1.63	83	0.01	0	62.93	13
6	30	24.68	54	2.55	182	0.01	0	72.04	17
7	52	28.70	89	5.05	376	0.01	0	82.60	22
8	95	33.64	138	12.12	802	0.01	0	107.38	27
9	180	38.25	197	38.15	1627	0.01	0	143.61	33
10	343	44.65	274	132.13	3432	0.01	0	231.03	39
11	674	55.10	363	507.20	6917	0.01	0	448.66	46
12	1302	78.20	474	2115.55	14506	0.01	0	693	53
13	2590	124.67	599	8740.52	29157	0.01	0	811.45	61
14	5022	217.41	750	37934.01	60902	0.01	0	865.53	69
15	10028	456.22	917	—	—	0.02	0	1041.23	78
16	19467	1003.94	1114	—	—	0.02	0	1685.67	87
17	38916	2376.30	1329	—	—	0.01	0	2452.95	97
18	75603	5645.29	1578	—	—	0.02	0	4114.78	107
19	151186	14596.77	1847	—	—	0.02	0	6618.15	118
20	293953	39313.78	2154	—	—	0.02	0	15847.63	129

Table 4: Group cardinality test. Times in milliseconds.

Since the input to cachet and relsat are logic formulas in CNF, the FM translation to CNF is now sketched, following the directions in [55]. 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, if s = 4, the tree structure is encoded by:

$$n \wedge (n_1 \rightarrow n) \wedge (n_2 \rightarrow n) \wedge (n_3 \rightarrow n) \wedge (n_4 \rightarrow n) \equiv n \wedge (\neg n_1 \vee n) \wedge (\neg n_2 \vee n) \wedge (\neg n_3 \vee n) \wedge (\neg n_4 \vee n)$$

For the cardinality restriction, the *low* and *high* restrictions are treated 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 $s - low$ children can be false). Which means that as soon as $s - low + 1$ children 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. In the example with s=4, the low limit is encoded by: $(n_1 \vee n_2 \vee n_3) \wedge (n_1 \vee n_2 \vee n_4) \wedge (n_1 \vee n_3 \vee n_4) \wedge (n_2 \vee n_3 \vee n_4)$

This mechanism produces $\binom{2h}{h+1}$ clauses. The *high* restriction is somewhat easier. Since in a set of $high + 1$ children at least one of them has to be false, one may just compute all the sets of children of size $high+1$ and add a clause with all the set members negated. For the example, this gives: $\neg(n_1 \wedge n_2 \wedge n_3 \wedge n_4) \equiv \neg n_1 \vee \neg n_2 \vee \neg n_3 \vee \neg n_4$

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Which produces $\binom{2h}{h+2}$ clauses. To sum up, the s = 4 example is encoded by the following formula with 10 clauses: $n \wedge (\neg n_1 \vee n) \wedge (\neg n_2 \vee n) \wedge (\neg n_3 \vee n) \wedge (\neg n_4 \vee n) \wedge (n_1 \vee n_2 \vee n_3) \wedge (n_1 \vee n_2 \vee n_4) \wedge (n_1 \vee n_3 \vee n_4) \wedge (n_2 \vee n_3 \vee n_4) \wedge (\neg n_1 \vee \neg n_2 \vee \neg n_3 \vee \neg n_4)$

Since $\binom{2h}{h+1} \geq \binom{2h}{h} \geq 2^h$ for any h, the number of clauses roughly doubles when going from s nodes to $s + 2$. The intent is to test the ability of logic model counters to cope with this kind of input. As showed by Table 4, treecount completes each test case in 0.02 milliseconds or less with a modest rate of growth. splot comes in second, between 2 and 3 orders of magnitude slower than treecount. Cachet takes twice as long as splot for the bigger models and finally relsat times out at size 15 (the timeout was set at 1 minute).

Using purely logic tools such as cachet and relsat does not provide a scalable solution for the problem of commonality counting in the presence of group cardinality because of their reliance on DPLL. In this experiment, the number of clauses grows exponentially with the number of nodes in the input FM. The number of branching decisions taken for DPLL search is included as an alternative informative measure, as well as the size of the BDD for splot. The differences between relsat and cachet are interesting. Cachet's caching strategy imposes a time overhead w.r.t to relsat but it really pays off in terms of branching decisions, considering that the input (number of clauses) grows exponentially. Relsat, being relatively simpler, performs faster in the first exemplars, but at size 10 is already lagging behing cachet and the number of decisions doubles with each new model. For treecount, the quadratic time growth of the algorithm is negligible for these input sizes.

For splot, the size of the resulting BDD is included in the *dec* column. It turns out that building the BDD takes up a lot of time as expected by the exponentially growing number of clauses, but the resulting graph is in fact quite compact. Moreover, there is a clear pattern of growth: 1, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7.... It stands to reason that there should be a relation between the DAGs, maybe a recurrence relation similar to the ones described in Section 5. If there is a fast way to compute the graph for the proper group cardinality nodes, it should be reasonably easy to *apply* it to the rest of the BDD, especially since such cardinality graph should be agnostic as to the ordering of the variables (i.e. the nodes can be renamed in any way). So, with some effort, BDD could be an adequate tool for managing group cardinality in FMs.

Admittedly, real FMs are not likely to display such a complex structure, but then again, extended cardinality could not be efficiently processed hitherto. Even a slight use of group cardinality may slow down the response of a design framework relying on model counters.

6.2.2 Second experiment. Real Models

In the second experiment, the same four approaches are tested on real-world FMs, as collected in the SPLOT repository. The 30 biggest models were chosen, excluding repetitions. Interestingly, none of these models makes use of group cardinality, so the outcome will help establish if treecount, group cardinality-capable, is penalized in its absence. As before, the number of branching decisions/BDD size is included where applicable. BDD size is an average of the different runs which explains why it is sometimes a fractional number, since the ordering heuristic is not deterministic and so different runs of the same test may produce differently-sized BDDs.

The results are shown in Table 5. As before, running times should not fool us. Cachet's infrastructure makes it slower than relsat in most exemplars. However, the number of branching decisions is consistently below that of relsat, and eventually cachet takes over on the biggest exemplar. Treecount is the fastest system in all cases, again by several orders of magnitude. It always makes less branching decisions than relsat and, in the vast majority of cases, less than cachet. The difference is particularly striking in those exemplars without textual constraints. For those, treecount makes no branching decisions, but cachet and relsat still have to embark in DPLL search to decompose the connected components of the FM. Splot does a good job when the BDD is small but struggles to complete the traversals otherwise. Results seem to confirm that treecount is a good choice when ECR is low. The *Electronic Shopping* exemplar deserves

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Name	#F	ECR	CD	cachet		relsat		treecount		splot	
				time	dec	time	dec	time	dec	time	BDD
Web Portal	49	0.24	0.50	180.60	2171	49.22	10124	0.21	31	118.66	191.03
Thread	51	0	—	167.92	571	27.61	1010	0.03	0	110.18	88
Doc Generation	53	0.25	0.62	184.62	1344	68.62	19416	0.28	31	123.84	340
Android SPL	54	0.17	0.56	177.34	689	31.51	1705	0.08	5	115.80	81
DELL	54	0.69	2.97	182.46	976	57.64	1924	0.49	27	168.62	275.04
Experimentation PL	56	0.16	0.44	178.34	0	26.87	1100	0.11	9	115.61	32
Letovanje	56	0.05	0.67	190.90	1205	38.91	2101	0.05	1	114.18	99.08
Face Animator	59	0.20	1.42	195.22	820	38.04	2967	0.10	4	122.70	90.34
Hotel PL	62	0	—	200.93	589	34.60	2539	0.03	0	115.14	44
Electronic Drum	63	0	—	212.04	1488	42.91	1488	0.03	0	118.45	77
Product Family Test	64	1	1.41	207.35	150	54.69	378	0.71	70	171	180.05
OW2 FraSCAti	65	0.68	1.05	219	1019	56.51	7954	1.95	257	192.15	1196.83
Phone Meeting	66	0	—	215.85	809	43.39	3610	0.03	0	121.48	66
Smart Home	66	0.06	0.50	217.84	854	41.27	3862	0.06	3	129.39	96
Arcade Game	70	0.57	0.85	238.14	1093	119.15	34334	0.25	7	166.23	483.81
HIS	73	0.11	0.50	232.95	753	46.26	1173	0.12	11	147.34	157.63
SimulES	73	0	—	240.62	1130	48.30	2534	0.04	0	126.30	71
Car Crisis Mgmt	74	0.04	0.67	240.75	903	50.26	2434	0.05	1	144.44	86
Video Player	76	0	—	244.01	282	64.11	11359	0.03	0	151.65	79
Database Tools	77	0.08	0.33	252.19	769	70.39	13308	0.06	4	151.70	103.75
E-Science	77	0	—	278.35	2935	73.68	6174	0.04	0	135.86	181
Eclipse Extensions	79	0.03	0.50	271.11	2327	59.22	4456	0.05	1	154.98	120
Web architectures	88	0	—	293.79	1855	67.23	3934	0.04	0	156.97	109
Billing	90	0.76	0.87	287.28	348	67.18	3907	48.73	5735	205.03	185.46
Car Selection	91	0.30	0.78	313.53	2829	87.50	4773	4.14	767	174.40	137.65
Ecologic Car	110	0.04	0.50	373.82	2816	112.44	4113	0.08	3	174.70	138
Model Transf	113	0	—	395.17	3144	124.09	7613	0.05	0	167.13	146
xtext	137	0.01	0.50	470.30	4401	159.55	17001	0.05	1	204.10	205
Printers	200	0	—	687.80	5259	443.07	63826	0.05	0	256.19	228
Electronic Shopping	326	0.11	0.60	1350.38	24900	1382.11	262389	65.22	6527	1479.53	23344.18

Table 5: SPLOT repository sample results. Times in milliseconds.

particular treatment. For all four systems, it marks the peak of both branching decisions and time. At a first glance, it does not seem a very hard test. With 326 features, it is the biggest model. However, the ECR is 0.11, not particularly high (the DELL model has 0.69) and the number of features present in the textual constraints is 35, less than the 40 features sported by an easy model like *Arcade Game*. The average number of literals in a clause is barely 2.05. The problem is that 27 features occur only once in the textual constraints and 9 clauses in textual constraints are formed from these variables and so are independent –the worst-case scenario for the hybrid systems. For splot, BDD size increases by two orders of magnitude. While splot's ordering heuristic (Pre-CL-MinSpan) is specifically tuned for use with FMs, BDD construction techniques are generic, so there is probably space for improvement. This exemplar is difficult for the logic systems and for splot because it is the biggest, and it is difficult for treecount because the textual constraints are largely independent of one another.

6.2.3 Third experiment. Computer-generated models without textual constraints

The four approaches have been measured against computer generated random models. Figure 4 displays the results of running the four programs with random FMs with no textual constraints ranging from 50 to 5000 features, with a branching factor of 6. Running times for cachet and relsat grow exponentially, although both hit the 1 minute timeout rather quickly, at sizes 3400 and 1300 respectively. Treecount (hardly visible in the graphic) is the clear winner, with 1.43 milliseconds at size 5000 against 23790 for splot. The *forcing* procedure clearly does not scale for cachet and relsat. It does for splot, although the absolute timings are much worse than those of treecount.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

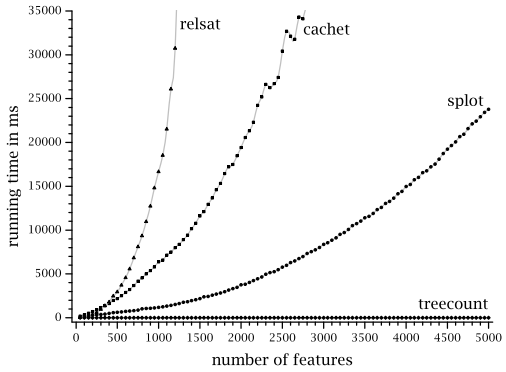


Figure 4: Performance without textual constraints

6.2.4 Fourth experiment. Computer-generated models with textual constraints

The next experiment tests satisfiable exemplars (ten of each) of sizes between 50 and 1000, ECR = 0.2, and clause density values ranging from 0.1, to 0.5, created using SPLOT generator with a branching factor of 6. We have set a timeout of 60 seconds for each test in this run of experiments and only show results when all 10 exemplars have been processed in time. Table 6 show the results.

Size	Clause density					
	0.1	0.2	0.3	0.4	0.5	
100	399.56	419.18	479.04	440.12	437.32	cachet
	182.12	312.40	677.27	343.80	553.62	relsat
	0.12	0.18	0.35	0.52	0.81	treecount
	182.11	185.78	190.11	191.97	199.08	splot
200	1022.03	1253.37	2714.42	3212.03	3196.15	cachet
	1486.73	7633.24	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	0.33	1.64	6.62	25.25	67.10	treecount
	325.84	376.18	445.28	568.00	744.36	splot
300	2084.60	3775.07	9427.70	18874.79	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	1.73	43.53	523.72	1368.78	6806.57	treecount
	473.95	779.60	1799.65	2548.33	≪timeout≫	splot
400	4247.83	14217.84	≪timeout≫	≪timeout≫	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	11.49	583.02	9165.96	≪timeout≫	≪timeout≫	treecount
	793.49	2309.78	17081.91	≪timeout≫	≪timeout≫	splot
500	9960.27	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	24.26	7872.75	≪timeout≫	≪timeout≫	≪timeout≫	treecount
	1269.74	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	splot
600	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	233.15	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	treecount
	3004.35	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	splot
700	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	1751.34	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	treecount
	8051.32	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	splot
800	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	cachet
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	relsat
	6072.60	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	treecount
	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	≪timeout≫	splot

Table 6: Performance in milliseconds with ECR 0.2 and various CD values

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

All four systems fail to finish the tests, especially with every increase in clause density. Relsat starts out faster than cachet but the situation quickly reverses. In fact, relsat times out at size 300 even for CD 0.1. Cachet lasts a bit longer but times out at size 600 for CD 0.1. Treecount is again the clear winner, and even so, it does not scale gracefully. Splot comes in second place. It is noteworthy that overconstraining the exemplars does not help any system, quite to the contrary, increasing the CD is a killer for all the approaches.

6.2.5 Fifth experiment. Computed generated models with textual constraints and group cardinality #2

The final experiment consisted in modifying the generated models in the last experiment to include group cardinality. Whenever there was a group node, we changed it randomly to a legal configuration (i.e. a [*low*, *high*] cardinality where $low \leq high$ and $high \leq$ the number of children of the node), which could be either FODA or a proper group cardinality configuration.

Run time increased 61% for cachet, 118% for relsat , 0.32% for treecount and 0.5% for splot.

The experiments indicate that group cardinality is a burden for cachet and relsat, even though the small branching factor limited the impact for these systems. Splot takes advantage of the fact that building the BDD takes a little longer but the bulk of the work traversing the BDD is very similar. Treecount performs well as expected.

6.3 Threats to Validity

Let us discuss threats to validity of the experimental evaluation:

1. **Choice of examples**. In the design of the experiments it was of paramount importance to address the performance of our approach and the baselines on large FMs since any method would do for small cases. As there are almost no publicly available industrial-sized FMs, this led to the use of random computer-generated examples within certain parameters. With these, the question is always whether the generated models can be considered realistic or not. We concur at this point with Mendonça's assertion: *From our experience in examining several FMs in the literature, we noticed that feature trees are frequently orders of magnitude larger than the extra constraints in terms of number of relations* [13]. For this reason we generated examples with no textual constraints in experiment 3 and also examples with ECR = 0.2 and CD between 0.1 and 0.5 in experiments 4 and 5.

   In [56], Berger et al. study the configuration models for the Linux (6320 features) and eCos (1244 features) kernels, which are examples of real big variability modeling systems, claiming an ECR of 0.86 in both cases, somewhat contradicting Mendonça's assumptions. It is worth mentioning that said systems go well beyond the usual FODA notation and its extensions, including feature attributes whose values may include a default and then change conditionally or unconditionally of other features' attribute values, including the use of arithmetic expressions for numbers and equality for strings. Another point of interest is that Kconfig (the Linux kernel configurator) allows illegal configurations to be selected and CDL (the eCos configuration language) is supported by an inference engine that is correct but not complete, that is, it proposes correct configurations to complete user selections, but not all the compatible ones. The authors also report that the maximum number of siblings is 158 and 29, respectively. These FMs resemble those described in [38] which integrate as many of the textual constraints as possible into a tree structure. Considering the differences and the fact that none of the algorithms used in this paper could process those FMs, we will stick to FMs with low ECR in the experiments while duly acknowledging that future work should address these bigger models.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

To further question the results in experiments 3-5, we included experiment 2 in which the FMs in the SPLOT repository, which are all real models (mostly from the academic world) albeit usually small. Since the results in all those experiments point in the same direction we consider the threat averted.

2. **Accurate and fair timing of the approaches**. In experiment 1, since some of the running times are so small and obviously prone to measuring errors, we have averaged the times over 10 runs. In experiment 2 and 3, we used 100 runs for each model. For experiments 4 and 5 we have averaged times over 10 exemplars for each combination of number of features and CD. To level up the field, we have edited the source code to measure only processing time, not I/O. Otherwise running times would exponentially rise when the input grows exponentially, such as for cachet, relsat and splot in experiment 1. Also, cachet and relsat heavily used temporary files to force the presence of the individual features.

3. **Comparison of C, C++ and Java implementations**. Cachet, relsat and treecount are C++ programs, while splot is written in Java. This suggests that splot timings could not be directly compared to those of the other approaches. The reality however, is complex. Cachet is built on top of zchaff, so it contains quite a bit of C code wrapped up as C++. Relsat and treecount are written in pure C++ while splot makes heavy use of JavaBDD, which in this case is linked to BuDDy, a pure C library with optional C++ wrappings. Only the ordering heuristic (which takes up typically less than 1% of runtime) is written in Java so language-wise all four approaches seem very comparable. In any case, treecount does not get an unfair advantage of being coded in C++.

7 Conclusions

Effective deployment of large-scale software product lines demands efficient support for automated analysis of expressively-rich FMs. In particular, this paper has shown that the computation of the number of products that implement each feature is helpful to describe the standarization/parameterization balance, detect flaws in the scope, assess the incremental development and improve the accuracy of economic models for product lines. Whereas there are alternative proposals for computing the total number of products, this paper contributes with an innovative algorithm to compute the feature commonalities.

Our algorithm is applicable for most FM notations since it takes into account textual constraints and the group cardinality construct. Theoretical considerations suggested that FM translation to propositional logic would be ineffective against widespread use of group cardinality. These considerations have been empirically tested and confirmed. As regards the group cardinality computation, we have presented a quadratic algorithm for a naturally exponential problem in Sections 5.3 and 5.4. The same quadratic complexity applies to the case in which no textual constraints are used. Furthermore, we have shown that commonality computation is possible with just a small overhead after computing the total number of products.

We have reported experimental evidence that the forcing technique to compute commonalities using models counters has serious scalability problems, especially for logic systems such as cachet and relsat. BDD-based approaches such as splot show a greater potential in that respect, although far behind treecount. None of the approaches is well suited for models with high clause density – a reminder of the difficulty of the task. Treecount's performance was superior in all test cases, often by several orders of magnitude over a variety of models, both real and computer-generated, making it a sound tool for automated analysis of FMs.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

8 Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful feedback. We also thank Roberto López Herrejón and Alexander Egyed at the Institute for Software Engineering and Automation (Johannes Kepler University of Linz, Austria) for their advice.

References

[1] K. Pohl, G. Bockle, F. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.

[2] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[3] F. Bachmann, P. C. Clements, Variability in software product lines, Tech. rep., CMU/SEI-2005-TR-012 (2005).

[4] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute (1990).

[5] A. Kübler, C. Zengler, W. Küchlin, Model counting in product configuration, in: 1st International Workshop on Logics for Component Configuration, Edinburgh, UK., 2010, pp. 44–53.

[6] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456–479.

[7] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, G. 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, 2007, pp. 243–253.

[8] K. Czarnecki, S. Helsen, U. W. Eisenecker, Formalizing cardinality-based feature models and their specialization., Software Process: Improvement and Practice 10 (1) (2005) 7–29.

[9] K. Czarnecki, S. Helsen, U. W. Eisenecker, Staged configuration using feature models., in: R. L. Nord (Ed.), Software Product Line Conference, Vol. 3154 of Lecture Notes in Computer Science, Springer, 2004, pp. 266–283.

[10] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: a literature review, Information Systems 35 (6).

[11] D. Fernandez-Amoros, R. Heradio, J. Cerrada, Inferring information from feature diagrams to product line economic models, in: Proceedings of the 13th International Conference on Software Product Lines, 2009, pp. 41–50.

[12] D. Benavides, On the automated analysis of software product lines using feature models. a framework for developing automated tool support, Ph.D. thesis, University of Seville (June 2007).

[13] M. Mendonça, Efficient reasoning techniques for large scale feature models, Ph.D. thesis, School of Computer Science, University of Waterloo (January 2009).

[14] J. C. Cleaveland, Program Generators with XML and Java, Prentice Hall, 2001.

[15] S. Q. Lau, Domain analysis of e–commerce systems using feature–based model templates, Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada (2006).

[16] D. Batory, Feature models, grammars, and propositional formulas, in: 9th international conference on Software Product Lines, Springer-Verlag, Rennes, France, 2005, pp. 7–20. doi:10.1007/11554844_3. URL http://dx.doi.org/10.1007/11554844_3

[17] K. C. Kang, J. Lee, P. Donohoe, Feature–oriented product line engineering, IEEE Software 19 (4) (2002) 58–65.

[18] G. Bockle, P. Clements, J. McGregor, D. Muthig, K. Schmid, Calculating roi for software product lines, IEEE Software 21 (3) (2004) 23–31.

[19] P. Clements, J. McGregor, S. Cohen, The structured intuitive model for product line economics, Tech. rep., CMU/SEI-2005-TR-003 (2005).

[20] K. Czarnecki, U. Eisenecker, Generative Programming: Methods Tools and Applications, Addison-Wesley, 2000.

[21] B. Boehm, A. W. Brown, R. Madachy, Y. Yang, A software product line life cycle cost estimation model, in: International Symposium on Empirical Software Engineering, 2004, pp. 156–164.

[22] S. B. A. B. Lamine, L. L. Jilani, H. H. B. Ghézala, Cost estimation for product line engineering using cots components, in: J. H. Obbink, K. Pohl (Eds.), Software Product Line Conference, Vol. 3714 of Lecture Notes in Computer Science, Springer, 2005, pp. 113–123.

[23] J. P. Nobrega, E. S. de Almeida, S. R. de Lemos Meira, Income: Integrated cost model for product line engineering, in: SEAA, IEEE, 2008, pp. 27–34.

[24] J. S. Poulin, The economics of software product lines, International Journal of Applied Software Technology 3 (1) (1997) 20–34.

[25] J. H. Wesselius, Software Product Lines Research Issues in Engineering and Management, Springer Berlin Heidelberg, 2006, Ch. Strategic Scenario-Based Valuation of Product Line Roadmaps, pp. 53–89.

[26] S. Cohen, Predicting when product line investment pays, Tech. rep., Software Engineering Institute (CMU/SEI-2003-TN-017) (2003).

[27] D. D. Galorath, M. W. Evans, Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves, Auerbach Publications, 2006.

[28] H. P. In, J. Baik, S. Kim, Y. Yang, B. Boehm, A quality-based cost estimation model for the product line life cycle, Communications of the ACM 49 (12) (2006) 85–88. doi:http://doi.acm.org/10.1145/1183236.1183273.

[29] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, ACM, New York, NY, USA, 1971, pp. 151–158.

[30] A. Biere, A. Biere, M. Heule, H. van Maaren, T. Walsh, Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications, IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[31] R. Bayardo, J. Pehoushek, Counting models using connected components, in: Proceedings of the National Conference on Artificial Intelligence, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2000, pp. 157–162.

[32] M. Thurley, sharpSAT–Counting Models with Advanced Component Caching and Implicit BCP, Theory and Applications of Satisfiability Testing-SAT 2006 (2006) 424–429.

[33] T. Sang, F. Bacchus, P. Beame, H. Kautz, T. Pitassi, Combining Component Caching and Clause Learning for Effective Model Counting, in: 7th International Conference on Theory and Applications of Satisfiability Testing, 2004, pp. 20–28.

[34] T. Sang, P. Beame, H. A. Kautz, Heuristics for fast exact model counting, in: 8th International Conference on Theory and Applications of Satisfiability Testing, 2005, pp. 226–240.

[35] M. Mendonca, A. Wasowski, K. Czarnecki, Sat-based analysis of feature models is easy, in: Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University, 2009, pp. 231–240.

[36] D. Benavides, A. Ruiz-Cortes, P. Trinidad, Automatic reasoning on feature models, in: Advanced Information Systems Engineering: 17th International Conference, CAiSE, 2005, pp. 491–503.

[37] D. Benavides, S. Segura, P. Trinidad, Using java csp solvers in the automated analyses of feature models, Techniques in Software 01 (2006) 399–408.

[38] K. Czarnecki, A. Wasowski, Feature diagrams and logics: There and back again, in: 11th International Software Product Line Conference, IEEE Computer Society, Washington, USA, 2007, p. 23–34. doi:10.1109/SPLC.2007.19.

[39] Y. Gil, S. Kremer-Davidson, Sans Constraints? Feature Diagrams vs. Feature Models, Lecture Notes in Computer Science I (6287) (2010) 271–285.

[40] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397. doi:http://doi.acm.org/10.1145/368273.368557.

[41] M. Mendonça, M. Branco, D. Cowan, S.P.L.O.T. - Software Product Lines Online Tools, in: 24th ACM SIGPLAN Conference on object oriented programming systems languages and applications - OOPSLA Companion, ACM Press, ACM Press, Orlando, Florida, USA, 2009, p. 761. doi:10.1145/1639950.1640002.

[42] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 8 (C-35) (1986) 677–691.

[43] M. Mendonça, A. Wasowski, K. Czarnecki, D. Cowan, Efficient compilation techniques for large scale feature models, in: Proceedings of the 7th international conference on Generative programming and component engineering, GPCE '08, ACM, New York, NY, USA, 2008, pp. 13–22. doi:10.1145/1449913.1449918.

[44] B. Bollig, I. Wegener, Improving the variable ordering of OBDDs is NP-Complete, IEEE Trans. Comput. 45 (1996) 993–1002. doi:http://dx.doi.org/10.1109/12.537122.

[45] F. Bacchus, S. Dalmao, T. Pitassi, Algorithms and Complexity Results for # SAT and Bayesian Inference, in: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, 2003, pp. 340–351.

[46] G. Tseitin, Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian), Steklov Mathematical Institute, 1968, Ch. On the complexity of derivation in propositional calculus, pp. 115–125.

[47] J. P. Marques-Silva, K. A. Sakallah, Grasp - a new search algorithm for satisfiability, in: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, ICCAD '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 220–227.

[48] N. Sörensson, N. Een, Minisat v1.13 - A SAT solver with conflict-clause minimization, in: System descriptions for the SAT competition, Vol. 2005, 2005, p. 53.

[49] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Proceedings of the 38th annual Design Automation Conference, DAC '01, ACM, New York, NY, USA, 2001, pp. 530–535. doi:http://doi.acm.org/10.1145/378239.379017.

[50] K. Pipatsrisawat, A. Darwiche, Rsat 1.03: Sat solver description, Tech. Rep. D–152, Automated Reasoning Group, Computer Science Department, UCLA (2006).

[51] Y. Hamadi, S. Jabbour, L. Sais, ManySAT: a parallel SAT solver, Journal on Satisfiability, Boolean Modeling and Computation 6 (2009) 245–262.

[52] D. Pretolani, Satisfiability and hypergraphs, Ph.D. thesis, Universitá di Pisa (1993).

[53] J. W. Freeman, Improvements to Propositional Satisfiability Search Algorithms, Ph.D. thesis, University of Pennsylvania (1995).

[54] T. Sang, P. Beame, H. A. Kautz, Solving bayesian networks by weighted model counting, in: 20^{th} National Conference of Artificial Intelligence (AAAI-05), Vol. 1, 2005, pp. 475–482.

[55] A. Biere, M. J. Heule, H. van Maaren, Toby, Walsh, Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

[56] T. Berger, S. She, R. Lotufo, Variability modeling in the real : A perspective from the operating systems domain, Measurement (2010) 73–82.

[57] J. Greenfield, K. Short, S. Cook, S. Kent, Software Factories: Assembling Applications with Patterns Models Framworks and Tools, Wiley, 2004.

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

A Algorithm to Compute Feature Commonality

This Appendix includes a detailed description of the algorithms and the attribute grammar presented in Section 5.

Algorithm 1: #DPLL adapted to FM

```
FM_DPLL(Φ, α, T, var products):
Data: Φ is a propositional logic formula in CNF
α is an assigment of the variables to the values {true, false, or
unassigned}
T : Is the FM tree structure
products : products[n] accumulates the number of products in which
feature n appears
begin
    apply UP to Φ and T and obtain Φ′ and α′
    if Φ′ is Satisfied then
        computeProducts(α, T, products)
        // See Figure 5
    else
        select an unassigned variable X in φ
        FM_DPLL(Φ′ |X=false, α′[X = false], T, products)
        FM_DPLL(Φ′ |X=true, α′[X = true], T, products)
```

Algorithm 2: Compute the type of a node

```
computeType(aSel, Desel) : type
Data: aSel is the augmented Sel attribute
      (Sel plus the mandatory attribute) and
      Desel is the attribute by the same
      name
Result: type is the type of node
begin
    if (aSel ∧ ¬ Desel) then
        type ← present
    else if (¬ aSel ∧ Desel) then
        type ← absent
    else if (aSel ∧ Desel) then
        type ← contradicting
    else
        type ← potential
    return type
```

Algorithm 3: Potential factor computation

```
#gCard(PChildrenVector, Low, High): v, SVector
Data: PChildrenVector includes the iv values of the
      potential children of the current node; cardinality
      limits of the node are [Low, High]
Result: v is the potential factor for parent ; SVector
        includes the node S values
begin
    if (node is leaf ∨ or ∨ xor) then
        v ← #Particular(PVector, Low, High)
        SVector ← nil
    else
        v, SVector ← #General(PVector, Low, High)
    return v, SVector
```

Algorithm 4: Potential factor computation for easier cases

```
#Particular(PVector, Low, High): V begin
    P ← 1
    if Low=1 ∧ High=NumberOfChildren then        // or
        forall the i such that 1 ≤ i ≤ NumberOfChildren
        do
            V ← V×(1 + VVector[i])
        V ← V − 1
    else if Low=1 ∧ High=1 then                  // xor
        forall the i such that 1 ≤ i ≤ NumberOfChildren
        do
            V ← V + VVector[i]
    // else leaf (do nothing)
    return V
```

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Algorithm 5: General case for potential factor

```
#General(PVector,Low,High): V, SVector begin
    if Low = 0 then
        V ← 1
    else
        V ← 0
    SVector[0] ← 1
    PowerSumVector[0] ← NumberOfChildren
    forall the k such that 1 ≤ k ≤ NumberOfChildren do
        PowerVector[k] ← 1
    forall the k such that 1 ≤ k ≤ High do
        ThisPowerSum ← 0
        forall the j such that 1 ≤ j ≤ NumberOfChildren do
            PowerVector[j] ← PowerVector[j] × PVector[j]
            ThisPowerSum ← ThisPowerSum + PowerVector[j]
        PowerSumVector[k] ← ThisPowerSum
        SVector[k] ← 0
        Parity ← 1
        forall the i such that 0 ≤ i < k do
            SVector[k] ← SVector[k] +
              Parity × SVector[k − i − 1]×
              PowerSumVector[i + 1]
            Parity ← −1×Parity
        SVector[k] ← SVector[k] ÷ k
        if k > Low−1 then
            V ← V + SVector[k]
    return V, SList
```

Algorithm 6: Contextual variability computation

```
TakeOneOut(PV, Low, High, SVector, NV): SV
Data: PV: Is the parent inner variability
Low and High are the limits of the cardinality range of the
parent
SVector is the variability vector computed in #gCard
NV is the inner variability of the calling node
Result: SV is the sibling variability (the variability due to
        the siblings of the calling node)
begin
    if NumberOfChildren=0 then                    // leaf node
        SV ← 1
    else if Low=NumberOfChildren ∧
    High=NumberOfChildren
    then                                          // mandatory
        SV ← PV ÷ NV
    else if Low=0 ∧ High=NumberOfChildren then
    // optional
        SV ← PV ÷ (NV + 1)
    else if Low=1 ∧ High=NumberOfChildren then  // or
        SV ← (VP + 1) ÷ (NV + 1)
    else if Low=1 ∧ High=1 then                   // xor
        SV ← PV − NV
    else                                  // the general case
        SVector′[0] ← 1
        if Low = 0 then
            SV ← 1
        else
            SV ← 0
        forall the k such that 1 ≤ k < High do
            SVector′[k] ← SVector[k] − NV×
            SVector′[k − 1]
            if k > Low −1 then
                SV ← SV+ SVector′[k]
    return SV
```

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

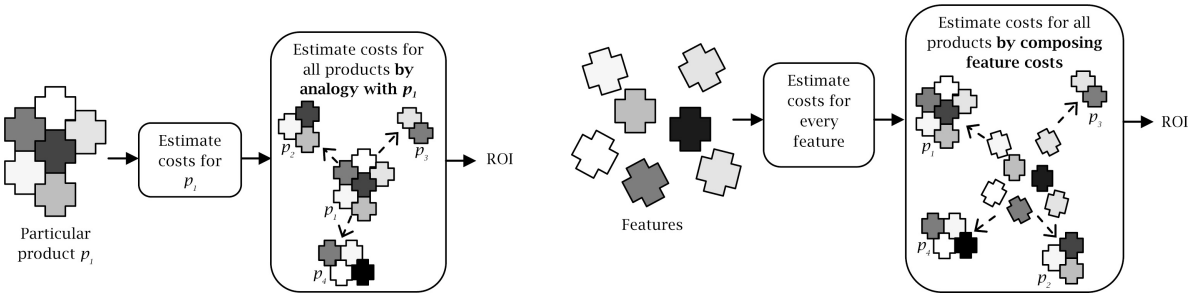
PRODUCTION	SEMANTIC RULES
R → F	(↓) F.cv = 1
F → G M **label (** FL **)**	(↑) F.sel = (α(F) = true) ∨ FL.sel (↑) F.desel = (α(F) = false) ∨ FL.c[pre] + FL.c[pot] < G.l ∨ FL.c[pre] > G.h ∨ FL.c[con] > 0; (↑) F.type = computeType(F.sel ∨ M.m, F.desel); (↓) if F.desel then FL.cv = 0 else FL.cv = F.cv; (↑) [F.potf, FL.s] = gCard(F.pot, G.l - FL.c[pre], G.h - FL.c[pre]); (↑) F.iv = FL.pref × F.potf; F.p = F.iv × F.cv; // : stands for list concatenation (↓)FL.extra = [F.p. G.l, G.h] : F.s; (↓) products[**label**.txt] += F.p
F → **leaf** M **label**	(↑) F.sel = (α(F) = true); (↑) F.desel = (α(F) = false); (↑) F.type = computeType(F.sel ∨ M.m, F.desel); (↓) F.p = F.cv; products[**label**.txt] += F.p
G → **group** $\mathbf{n}_1$ $\mathbf{n}_2$	(↑) G.l = $\mathbf{n}_1$.val; G.l = $\mathbf{n}_2$.val;
M → **mandatory**	(↑) M.m = true;
M → ϵ	(↑) M.m = false;
FL → F FL_1	(↑) FL.sel = FL_1.sel ∨ (α(F) = true); (↑) FL_1.c[F.type]= F.c[F.type] + 1; (↑) forall t in {pre, con, abs, pot} - {F.type} do FL.c[t] = FL_1.c[t] (↑) if F.type = pre then FL.pref = F.iv × FL_1.pref else FL.pref = FL_1.pref; (↓) if F.type = pre then F.cv = FL.cv * FL.iv / F.iv (↑) if F.type = pot then // : stands for list concatenation FL.pot = FL_1.pot : F.iv else FL.pot = FL_1.pot
FL → ϵ	(↑) forall s in {pre, con, abs, pot} do FL.c[s] = 0; (↑) FL.sel = FL.desel = false; FL.pref = 1; (↑) FL.pot = ∅

Figure 5: An attribute grammar to describe the computeProducts(α, products) function

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

B Increasing COPLIMO's accuracy by using $\#\mathcal{P}_f$

As depicted in Figure 6.a, COPLIMO starts estimating the development costs of a particular product p_1 (i.e., a domain *stereotype*). Then, supposing that all products in the scope of a SPL are quite similar (i.e., $p_1 \approx p_2 \approx ... \approx p_{\#\mathcal{P}}$), it extrapolates the costs of p_1 to compare the costs of building all the products under a SPL approach versus building them one by one. Describing COPLIMO [21, 28] in a top-down fashion, it estimates Return On Investment (ROI)[12] with Equations 10–13, where PLS stands for the Product Line Savings.



(a) Original COPLIMO (b) COPLIMO improved by using $\#\mathcal{P}_f$

Figure 6: Approaches to estimate the ROI of a SPL

$$\text{ROI}(\#\mathcal{P}) = \frac{\text{PLS}(\#\mathcal{P})}{\text{CCAB}(\#\mathcal{P})} = \frac{\text{PLS}(\#\mathcal{P})}{|\text{PLS}(1)|} \qquad (10)$$

$$\text{PLS}(\#\mathcal{P}) = \text{PMNR}(\#\mathcal{P}) - \text{PMR}(\#\mathcal{P}) \qquad (11)$$

COPLIMO relies on COCOMO II to estimate the amount of effort in *Person-Months* (PM) it will take to develop a software product. $\text{PMR}(\#\mathcal{P})$ and $\text{PMNR}(\#\mathcal{P})$ stand for the effort to develop $\#\mathcal{P}$ products with reuse and without it (i.e., SPL approach vs individual development). COPLIMO makes the following simplifying assumptions:

1. All domain products require the same amount of effort to be developed. Thus, $\text{PMNR}(\#\mathcal{P})$ is calculated by Equation 12, where p is any domain product and $\text{PM}(p)$ is the COCOMO II effort estimation of p.
$$\text{PMNR}(\#\mathcal{P}) = \#\mathcal{P} \times \text{PM}(p) \qquad (12)$$
2. All core assets are reused by all products; i.e., $\forall p \in \mathcal{P} \cdot \text{CAB} \subseteq p$. Hence, the Cost of developing the CAB (CCAB) is approximated by $|\text{PLS}(1)|$.
3. COPLIMO uses acronyms PFRAC, RFRAC and AFRAC to denote the proportions of a product that are unique, composed of black-box reused assets and composed of white-box reused assets, respectively[13]. COPLIMO assumes that all products have the same PFRAC, RFRAC and AFRAC; i.e., $\forall p, p' \in \mathcal{P} \cdot (\text{PFRAC}(p) = \text{PFRAC}(p')) \wedge (\text{RFRAC}(p) = \text{RFRAC}(p')) \wedge (\text{AFRAC}(p) = \text{AFRAC}(p'))$

As a result, $\text{PMR}(\#\mathcal{P})$ is calculated by Equation 13. Such equation distinguishes between the costs of reusing black-box and white-box features, which are set by the Assessment and Assimilation (AA) and the

[12] the SPL approach is not always the best economic choice for developing a family of related products. Since PLS may be negative, the absolute value of PLS(1) is used in Equation 10

[13] an asset is *black-box reused* if it is not necessary to understand the asset implementation to reuse it. Otherwise, the asset is *white-box reused* [57]

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

Adaptation Adjustment Modifier (AAM) parameters, respectively.

$$\text{PMR}(\#\mathcal{P}) = \begin{cases} \text{PMNR}(1) \times \Big(\text{PFRAC+RCWR} \times (\text{RFRAC+AFRAC})\Big) & \text{if } \#\mathcal{P} = 1 \\ \text{PMR}(1) + (\#\mathcal{P} - 1) \times \text{PMNR}(1) \times (\text{PFRAC} + \text{RFRAC} \times \frac{\text{AA}}{100} + \text{AFRAC} \times \text{AAM}) & \text{if } \#\mathcal{P} > 1 \end{cases} \quad (13)$$

Whenever the products are not extremely homogeneous, COPLIMO assumptions may produce problematic distortions in the estimates. Figure 6.b outlines our proposal to avoid them by using $\#\mathcal{P}_f$. First of all, the costs are estimated for each feature, i.e., COCOMO II is used to estimate PM(f) instead of PM(p). Then, the costs of each product are calculated by adding the costs of the features that it includes. Therefore, ROI is estimated with Equations 14-20, where CRCAB are the Cost of Reusing the CAB to build the products in $\mathcal{P}$, and CUNIQ($\#\mathcal{P}$) are the Cost of developing the UNIQue parts of the products in $\mathcal{P}$.

$$\text{ROI}(\#\mathcal{P}) = \frac{\text{PLS}(\#\mathcal{P})}{\text{CCAB}(\#\mathcal{P})} \quad (14) \qquad \text{PLS}(\#\mathcal{P}) = \text{PMNR}(\#\mathcal{P}) - \text{PMR}(\#\mathcal{P}) \quad (15)$$

$$\text{PMNR}(\#\mathcal{P}) = \sum_{f \in \mathcal{F}} (\text{PM}(f) \times \#\mathcal{P}_f) \quad (16) \qquad \text{PMR}(\#\mathcal{P}) = \text{CCAB}(\#\mathcal{P}) + \text{CUNIQ}(\#\mathcal{P}) + \text{CRCAB}(\#\mathcal{P}) \quad (17)$$

$$\text{CCAB}(\#\mathcal{P}) = \sum_{f \in \text{CAB}} (\text{PM}(f) \times \text{RCWR}(f)) \quad (18) \qquad \text{CUNIQ}(\#\mathcal{P}) = \sum_{a \notin \text{CAB}} \text{PM}(f) \quad (19)$$

$$\text{CRCAB}(\#\mathcal{P}) = \sum_{f \in \text{CAB}} \left(\text{PM}(f) \times \#\mathcal{P}_f \times \begin{cases} \frac{\text{AA(f)}}{100} & \text{if } a \text{ is BBR} \\ \text{AAM(f)} & \text{if } a \text{ is WBR} \end{cases} \right) \quad (20)$$

Note that our new approach provides the following improvements to original COPLIMO:

1. COPLIMO estimation of PMNR depends on the chosen stereotypical product p. Our improved model corrects such variability by providing an average value, i.e., $\text{PMNR}^{\text{Improved}}(\#\mathcal{P}) = \overline{\text{PMNR}^{\text{COPLIMO}}}(\#\mathcal{P})$, where $\overline{\text{PMNR}^{\text{COPLIMO}}}(\#\mathcal{P})$ stands for the arithmetic mean of the COPLIMO estimations of PMNR using each domain product as stereotype, i.e.,

$$\frac{\#\mathcal{P} \times \text{PM}(p_1) + \ldots + \#\mathcal{P} \times \text{PM}(p_{\#\mathcal{P}})}{\#\mathcal{P}}$$

Proof:

$$\overline{\text{PMNR}^{\text{COPLIMO}}}(\#\mathcal{P}) = \frac{\sum_{p \in \mathcal{P}} (\#\mathcal{P} \times \text{PM}(p))}{\#\mathcal{P}} = \sum_{p \in \mathcal{P}} \text{PM}(p) =$$
$$= \sum_{p \in \mathcal{P}} \Big(\sum_{f \in p} \text{PM}(f) \Big) = \sum_{f \in \mathcal{F}} (\text{PM}(f) \times \#\mathcal{P}_f) = \text{PMNR}^{\text{Improved}}(\#\mathcal{P}) \quad (21)$$

2. Original COPLIMO underestimates the cost of building the core asset base. Since ROI is estimated by dividing PLS into CCAB, COPLIMO overestimates it. Our COPLIMO reformulation corrects such problems.

   Proof: Our improved model provides Equation 18 to estimate CCAB, which takes into account the cost of all the features that are included into the CAB. On the contrary, original COPLIMO does not take into account the cost of all the core features to calculate CCAB. To show that COPLIMO underestimates CCAB, we will demonstrate that the following proposition holds: $\text{CCAB}^{\text{COPLIMO}} < \text{CCAB}^{\text{Improved}}$.

   In COPLIMO, the CAB development cost is approximated by |PLS(1)|, i.e., $\text{CCAB}^{\text{COPLIMO}} = |\text{PMNR}(1) - \text{PMR}(1)|$. So, the minimum value $\text{CCAB}^{\text{COPLIMO}}$ can get is 0 and it happens when PMNR(1) = PMR(1).

Draft version of the paper published in IEEE TSE, doi:10.1109/TSE.2014.2331073

From Equation 13, $\mathrm{PMR}(1) = \mathrm{PMNR}(1)$ when $\mathrm{PFRAC}(p) = 1$ and $\mathrm{RFRAC}(p) + \mathrm{AFRAC}(p) = 0$, since $\mathrm{RCWR}(p) \geq 1$. That is, $\mathrm{CCAB}^{\mathrm{COPLIMO}}$ is 0 when the stereotypical product p does not reuse any core feature. On the other hand, $\mathrm{CCAB}^{\mathrm{COPLIMO}}$ increases as $\mathrm{PMR}(1)$ gets bigger than $\mathrm{PMNR}(1)$, i.e., as p reuses more features. The maximum value of $\mathrm{CCAB}^{\mathrm{COPLIMO}}$ happens when $\mathrm{PFRAC}(p) = 0$ and $\mathrm{RFRAC}(p) + \mathrm{AFRAC}(p) = 1$, i.e., when p has been fully built by reusing core features. As Equation 22 shows (remember that $\mathrm{RCWR} \geq 1$), even the maximum value of $\mathrm{CCAB}^{\mathrm{COPLIMO}}$ does not account for the cost of all core features.

$$\begin{aligned}\mathrm{CCAB}^{\mathrm{COPLIMO}} &= \mathrm{PMR}(1) - \mathrm{PMNR}(1) = \mathrm{PM}(p) \times \mathrm{RCWR}(p) - \mathrm{PM}(p) = \\ &= \sum_{f \in \mathrm{CAB}} (\mathrm{PM}(f) \times \mathrm{RCWR}(f)) - \sum_{f \in \mathrm{CAB}} \mathrm{PM}(f) = \mathrm{CCAB}^{\mathrm{Improved}} - \sum_{f \in \mathrm{CAB}} \mathrm{PM}(f) < \mathrm{CCAB}^{\mathrm{Improved}}\end{aligned} \tag{22}$$