Supporting the Statistical Analysis of Variability Models

Ruben Heradio
Universidad Nacional de Educacion a Distancia
Madrid, Spain
rheradio@issi.uned.es

David Fernandez-Amoros
Universidad Nacional de Educacion a Distancia
Madrid, Spain
david@issi.uned.es

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Variability models are broadly used to specify the configurable features of highly customizable software. In practice, they can be large, defining thousands of features with their dependencies and conflicts. In such cases, visualization techniques and automated analysis support are crucial for understanding the models. This paper contributes to this line of research by presenting a novel, probabilistic foundation for statistical reasoning about variability models. Our approach not only provides a new way to visualize, describe and interpret variability models, but it also supports the improvement of additional state-of-the-art methods for software product lines; for instance, providing exact computations where only approximations were available before, and increasing the sensitivity of existing analysis operations for variability models. We demonstrate the benefits of our approach using real case studies with up to 17,365 features, and written in two different languages (KConfig and feature models).**

***Index Terms*—Variability modeling, feature modeling, software product lines, software visualization, binary decision diagrams.**

I. Introduction

A common challenge in software engineering is enabling and coping with many variants of software products that are customized for different market segments or contexts of use. This is explored in paradigms such as *Software Product Lines* (SPLs) [1] or *Context-Aware Software* [2]. An essential tool to tackle this challenge are *Variability Models* (VMs), which specify the common and variable features available for the software products, together with the inter-feature conflicts and dependencies [3], [4].

Numerous visualization methods [5] and analysis operations [6] support the reasoning on non-trivial VMs. Introduced in 1990, *feature diagrams* [7] are the prevalent way to visualize VMs as graphs whose nodes and edges depict features and inter-feature relationships. Such representation works nicely for small VMs, but it becomes ineffective for large models because the resulting graphs are overly complicated. Many analysis operations are excessively rigid. For instance, current approaches for detecting dispensable features only identify those that, due to conflicts/dependencies with the remaining features, cannot be included in any product at all, overlooking thus features with a reusability insignificantly above zero.

This work has been supported by (i) the Spanish Ministry of Education and Vocational Training under the projects with reference DPI2016-77677-P and CAS17/00022, (ii) the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria; and (iii) the FFG, Contract No. 854184.

This paper proposes an alternative way to reason about VMs. The basic idea is adopting a method that, in many other knowledge domains, has proven to be successful for describing and interpreting variation in large samples/populations: *statistics*. For that, it presents two algorithms that compute the primary elements needed for the VM statistical analysis: (i) the *Feature Inclusion Probability* (FIP) algorithm determines the probability for a feature to be included in a valid product, and (ii) the *Product Distribution* (PD) algorithm determines the number of products having a given number of features.

SPL engineering typically distinguishes two roles: the *domain engineer* and the *application engineer* [8]. Whereas the domain engineer undertakes the product line development (i.e., she engineers *for* reuse), the application engineer obtains particular systems from the product line through a configuration process (i.e., she engineers *with* reuse). Our approach assists both roles.

Regarding the domain engineer, our method supports representing the feature and product variation using general statistical plots (e.g., histograms, box-plots, etc.), and summarizing the variation through descriptive statistics (e.g., mean, standard deviation, etc.). This way, the engineer receives information about the complexity of the software products, and the SPL itself. Moreover, our approach supports augmenting the sensitivity of binary analysis operations by redefining them into probabilistic terms, hence providing a continuous range of values instead of a simplistic yes/no categorization. Engineers may use this, for instance, to detect highly dispensable features whose reuse probability is close to zero, but not exactly zero.

Regarding the application engineer, our method provides information about the implications of her decisions (i) in terms of features (e.g., if feature f is selected, which other features become selected/excluded due to their dependencies/ conflicts with f?), and also (ii) in terms of the final product (e.g., if feature f is selected, what size will the final product probably have?). Moreover, some procedures have been proposed to guide the engineer through the configuration space by using the concept of feature probability [9], [10], [11], [12]. However, as existing methods for computing feature probabilities do not scale for large VMs [13], probabilities are often roughly approximated from samples of historical product configurations [12], [14] or set manually by the

engineer according to her beliefs [15]. This paper contributes to configuration guidance procedures by supporting the exact and scalable feature probability computation.

Most existing methods for automated reasoning on VMs convert the models into Boolean logic formulas for subsequent processing with logic engines [16]. This translation of VMs into Boolean logic is a well-studied problem, supported for most VM notations, such as *feature models* [16], *KConfig* [17], [18], or *CDL* [19]. Our algorithms work with practically every VM notation as they build on the *Binary Decision Diagram* (BDD) [20], [21] encoding of the VM Boolean formulas.

We demonstrate the feasibility and benefit of our approach with real VMs specified in two distinct languages (KConfig and feature models). The investigated VM examples differ in the number of features (ranging from small to huge with up to 17,365 features), and come from different application domains (open source software projects, the automotive industry, and web configurators). Among other issues, the experiments reveal that some models have a surprisingly high number of features with extremely low reusability.

The remainder of this paper is organized as follows: Section II motivates the statistical analysis of variability models, illustrating its benefits with a real example. Section III describes our algorithms in detail. Section IV reports the application of the approach to distinct case studies. Section V discusses related work. Finally, Section VI summarizes this paper's main conclusions and outlines directions for future research.

II. Motivating the Statistical Analysis of Variability Models

Most approaches for providing engineers with visualization assistance to understand non-trivial VMs use graphs (or trees), whose nodes and edges represent features and constraints, respectively [5]. *Feature models* are the most widespread graphical notation for VMs [22].

In practice, VMs can be huge [23] and for those cases, their visual graph representation becomes ineffective. For example, the EmbToolkit project (www.embtoolkit.org) eases the application development and firmware generation for highly customized embedded Linux products. Its VM is specified in a text-based language called KConfig, which is also used in other popular open source projects, such as the Linux Kernel, uClib, or axTLS. The KConfig specification of EmbToolkit encompasses 1,815 configurable features, together with 7,193 inter-feature constraints. Figure 1 shows the graph representation of the KConfig specification of EmbToolkit 1.7.0. Such visual representation offers little value even when zooming in to make the node labels readable.

In contrast, we propose a statistical approach to describe and interpret the variation of the features and products specified by a VM. In the following subsections, we outline how this method assists both the domain and the application engineers.

A. Domain engineer's support

Our algorithms provide the fundamental information to enable answering the following key questions:

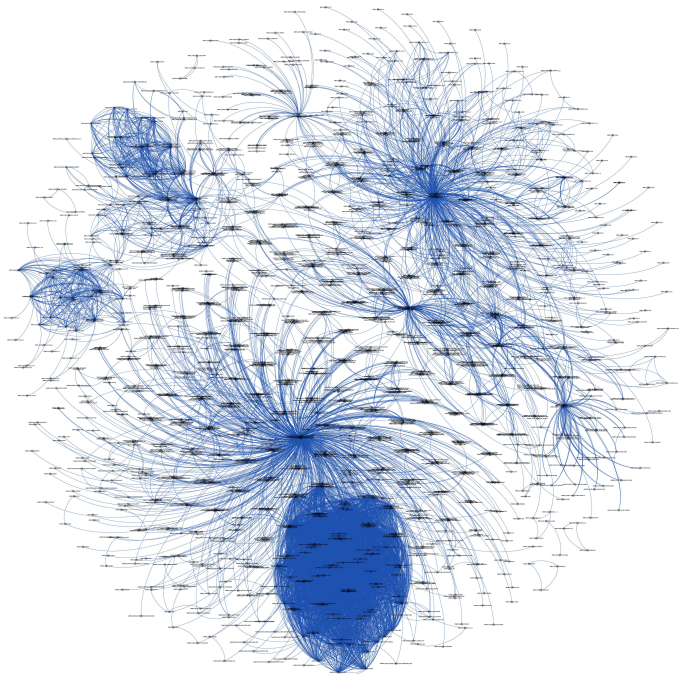


Fig. 1: Graph-representation of the EmbToolkit KConfig

1) How complex are the products?: The complexity of a product can be roughly measured by its number of features [24]. Our PD algorithm computes the products' distribution regarding their number of features. This distribution is the basis for distinct plots and descriptive statistics further characterizing products' complexity.

For instance, the density plot in Figure 2 and the descriptive statistics in Table I summarize the product distribution for the KConfig specification of EmbToolkit 1.7.0. This way, the engineer becomes aware that the most frequently occurring number of features for a product is 773, that the smallest and largest products have 19 and 1398 features, respectively, etc.

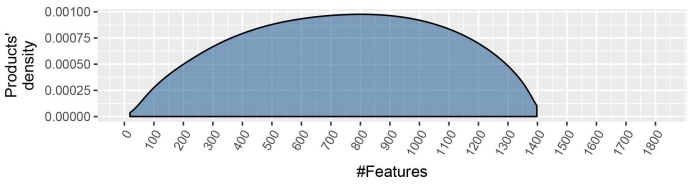


Fig. 2: EmbToolkit product distribution

Mean	Standard deviation	Median	Median absolute deviation	Mode	Min	Max	Range
741.49	330.91	748	391.41	773	19	1398	1379

TABLE I: Product distribution descriptive statistics

2) How complex is the product line?: The complexity of a SPL may be characterized by the following three core metrics: the number of features the SPL manages, the number of valid

products that can be derived, and the resulting homogeneity of those products [24] (i.e., how much does one product differ from the others). The PD algorithm in combination with our *Feature Inclusion Probability* (FIP) algorithm provides a clear picture of the products' homogeneity. Figure 3 presents two extreme cases: one of extreme homogeneity (top), the other of extreme heterogeneity (bottom):

- The top row describes a SPL where products are very homogeneous because (i) most products contain a similar number of features (i.e., its distribution has low variance - see the plot on the left), and (ii) most features are nearly always included (i.e., the feature probabilities are close to one and have low variance - right plot).
- The bottom row describes a SPL where the products are very heterogeneous because (i) some products may contain only a few features while others may contain a high number of features (i.e., the product distribution has high variance), and (ii) most features are nearly never included in a product (i.e., the feature probabilities are close to zero and have low variance).

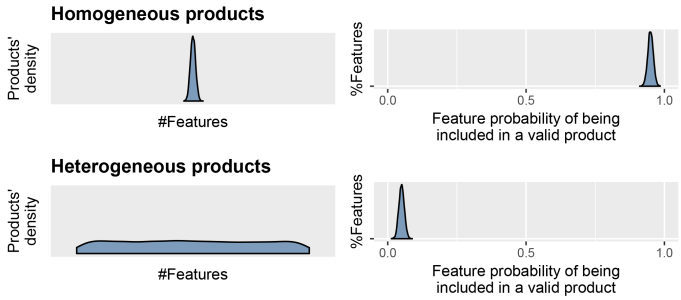


Fig. 3: Products' homogeneity characterization

3) Should the SPL be refactored to simplify its maintenance?: The histogram in Figure 4 depicts the feature probability distribution for EmbToolkit 1.7.0. Three zones have been highlighted in the plot, whose detailed information is summarized in Table II:

- The red shaded area (left) highlights the features with probability less than or equal to 0.05 of being included in a valid product. The extreme cases are those with zero probability, which are commonly called *dead* [6], [25].

Interestingly, 6.23% of the EmbToolkit features are dead, and thus they should be removed from the KConfig specification as they are completely without value.

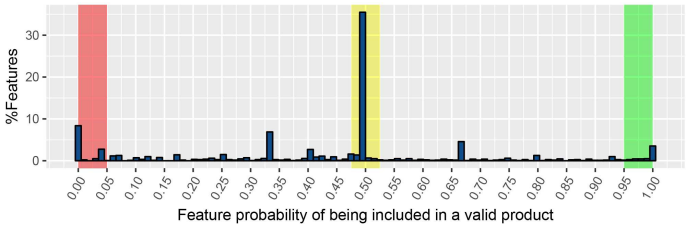


Fig. 4: EmbToolkit feature probability distribution

Dead features		Unconst. opt. features		Core features	
$p = 0$	$p \leq 0.05$	$p = 0.5$	$0.475 \leq p \leq 0.525$	$p = 1$	$p \geq 0.095$
6.23%	11.9%	25.73%	38.95%	1.21%	5.29%

TABLE II: Dead, core, and (potentially) optional features

- The green shaded area (right) emphasizes the features that are required by almost every valid product, being the extreme cases those with probability one, which are usually called *core* as they are present in all products.
- The yellow shaded zone (middle) identifies low-constraint features. In particular, those with probability 0.5 are typically pure optional features whose selection is unconstrained.

Our approach also provides assistance when historical data about the actual feature inclusion are available; e.g., the *Debian popularity contest* gathers information about how many times each Debian package has been installed (https://popcon.debian.org/). In this case, the domain engineer compares the VM statistics with the historical ones. If, for example, the actual products tend to be much smaller than the product distribution mode obtained from the VM, then perhaps the SPL is unnecessarily complex and could be simplified. Understanding the answers to these questions is thus of essential value for SPL and product testing, evolution, and reuse.

B. Application engineer's support

Our approach supports the application engineer's decision making by showing the impact that a decision has on:

1) The remaining features: For example, if the engineer selects the ARM architecture for EmbToolkit (`EMBTK_ARCH_ARM`), then our FIP algorithm will show that some other features will necessarily be excluded from the product (e.g., the probability of `KEMBTK_UCLIBC_TARGET_mips` becomes zero), and that the selection of other features will become difficult (e.g., the probability of `EMBTK_CLIB_GLIBC` decreases to $7.41 \cdot 10^{-35}$). It is worth noting that our approach determines feature exclusion beyond explicit constraints among two features by considering the overall set of constraints and currently selected features.

2) The product under configuration: For instance, our FIP and PD algorithms support providing plots such as the one in Figure 5, which shows how the configuration space shrinks with each engineer's decision about selecting/excluding features. Note that the product distribution variance decreases progressively until it becomes zero at the end of the configuration process.

Several heuristics have been proposed to speed up product configuration by taking advantage of the fact that, due to the inter-feature constraints, some decisions can be automatically derived from other decisions previously made. Some of those heuristics are based on approximating feature probabilities [10], [11]. Since our FIP algorithm computes those probabilities, it provides better support for the aforementioned heuristics.

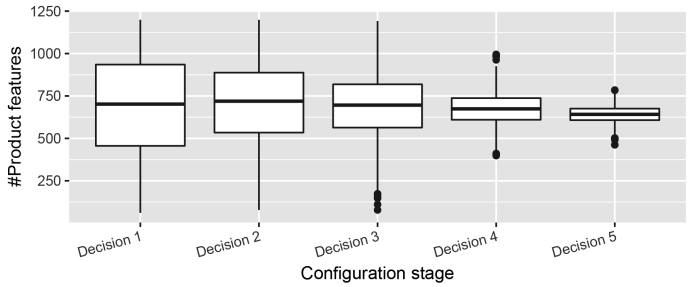


Fig. 5: Visualizing the product derivation progress

III. Computing feature and product distributions

This section describes a new method to compute the feature and product distributions from a VM. First, Section III-A introduces some probability definitions and the BDD technology our approach is built upon. Then, Sections III-B and III-C explain our algorithms in detail.

A. Preliminaries

The KConfig file in Figure 6 will be used throughout this section as running example. It is composed of several *configs* that specify three features A, B and C, and their interdependencies.

```
1 config A
2   bool "A value?"
3   select C if !B
4 config B
5   bool "B value?"
6   depends on A
7 config C
8   bool
```

Fig. 6: Running example: a KConfig file

All features are Boolean (Lines 2, 5 and 8), meaning that they can be either selected or deselected. Features can acquire their value from the user input, but also from other feature values. For instance, Configs A and B specify a *prompt* to request the user about their feature values (e.g., `"A value?"`). In contrast, C does not specify any prompt, and its value is derived as follows: C is selected whenever A is selected, but not B (Line 3). Finally, feature B depends on A, i.e., to be selected in a product, B requires that A is selected as well.

As a result, the configuration space encompasses only three valid products: $\{\overline{A}, \overline{B}, \overline{C}\}, \{A, \overline{B}, C\}, \{A, B, \overline{C}\}$, where f or $\overline{f}$ represents that feature f is selected or deselected, respectively. Therefore:

- The product distribution, regarding the number of features each product has, is: one product with zero features ($\{\overline{A}, \overline{B}, \overline{C}\}$), zero products with one feature, two products with two features ($\{A, \overline{B}, C\}$ and $\{A, B, \overline{C}\}$), and zero products with three features.
- The probability of A, B and C to be selected in a valid product is $2/3, 1/3$ and $1/3$, respectively.

1) Boolean representation of variability models: Most approaches for automated reasoning on VMs are based on converting the models into Boolean logic formulas, which are then processed with logic engines.

The details of this translation can be found in [16] and [18] for feature and KConfig models, respectively.

For instance, the VM in Figure 6 is equivalent to the formula $\Phi = ((A \wedge \overline{B}) \leftrightarrow C) \wedge (B \rightarrow A)$, whose truth table is summarized in Table III (1 and 0 means true and false, respectively).

A	B	C	Φ
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

TABLE III: Running example truth table

The truth table contains all possible configurations. The valid and invalid products are represented by rows where Φ is 1 and 0, respectively. For each row, the inclusion/exclusion of a feature is represented by 1/0 in its corresponding column. For example, the seventh row depicts the valid product $\{A, B, \overline{C}\}$.

The following probabilities are defined from the truth table:

- $p(\Phi)$ and $p(\overline{\Phi})$ are the probabilities of Φ to be 1 and 0, respectively; $p(\Phi)$ and $p(\overline{\Phi})$ are calculated as the number of rows where Φ is 1 and 0, respectively, divided by the total number of rows. In Table III, $p(\Phi) = 3/8$ and $p(\overline{\Phi}) = 5/8$.
- $p(x, \Phi)$ is the *joint probability* of x and Φ to be both 1; it is computed as the number of rows where both x and Φ are 1 divided by the total number of rows. For example, $p(A, \Phi) = 2/8$. It is worth noting that joint probabilities are symmetrical, i.e., $p(x, \Phi) = p(\Phi, x)$. Obviously, other joint probabilities can be defined negating x or Φ; e.g., $p(\overline{A}, \Phi) = 1/8$, $p(A, \overline{\Phi}) = 2/8$, etc.
- The *conditional probability* $p(x|\Phi)$ is the probability that x is 1 knowing beforehand that Φ is 1. In other words, it is the number of rows where both x and Φ are 1 divided by the number of rows where Φ is 1. For example, $p(A|\Phi) = 2/3$, $p(\overline{A}|\Phi) = 1/3$, etc.

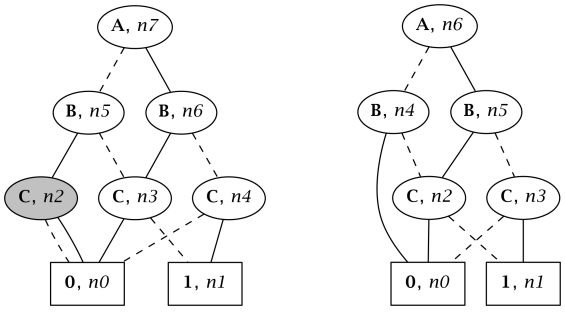
In this paper, we are especially interested in getting the probability each feature has to be included in a valid product, i.e., $p(x|\Phi)$. Nevertheless, this computation will be built upon other probabilities. In particular, by definition:

$$p(x|\Phi) = \frac{p(x, \Phi)}{p(\Phi)} \Rightarrow p(x, \Phi) = p(x|\Phi)p(\Phi)$$

Likewise, $p(\Phi|x) = \frac{p(\Phi, x)}{p(x)} \Rightarrow p(\Phi, x) = p(\Phi|x)p(x)$.

As joint probabilities are symmetrical, then $p(x, \Phi) = p(\Phi, x) \Rightarrow p(x|\Phi)p(\Phi) = p(\Phi|x)p(x) \Rightarrow p(x|\Phi) = \frac{p(\Phi|x)p(x)}{p(\Phi)}$. This last relation, which supports deriving $p(x|\Phi)$ from $p(\Phi|x)$, is known as *Bayes' rule*, and it will be used in Section III-B to get $p(x|\Phi)$.

2) Binary decision diagrams: Truth tables are convenient to understand the concepts we will handle to get the feature probabilities and product distribution, but not to make the computations because their size grows exponentially with the number of variables (a table with n variables has 2^n rows). In contrast, BDDs, which can be thought as compressed truth tables without redundancies, are by far more scalable [20], [26], [21]. An example that illustrates their compression power is reported in this paper experimental section: the KConfig specification of the uClibc library for developing embedded



(a) Non-reduced (b) Reduced

Fig. 7: BDD encoding for the KConfig example in Figure 6

Linux systems has 306 features and thus its truth table would have 2^{306} rows; nevertheless, its BDD encoding has only 3,085 nodes.

A BDD is a rooted directed acyclic graph where (i) all terminal nodes are labeled with 0 or 1, and (ii) all non-terminal nodes are labeled with a Boolean variable. Each non-terminal node has exactly two edges from that node to others: one labeled 0 and the other 1. They are called the *low* and *high* edges, and are usually represented graphically with dashed and solid lines, respectively. A BDD is *ordered* if the variables always appear in the same order for all the paths from the root to the terminal nodes. For instance, Figure 7a represents a BDD with the ordering $[A, B, C]$ for our running example. It has eight nodes, two terminals n_0 and n_1, and six non-terminals $n_2, n_3, \ldots, n_7$.

Likewise rows in truth tables, paths in BDDs represent variable assignments. In a path, x is assigned to 0 (or 1) if it goes through the low (or high) outgoing edge of a node labeled x, and the resulting evaluation is 0 (or 1) if the path ends up in the terminal 0 (or 1). For example, the 6^{th} row in Table III ($A, \overline{B}, C, \Phi$) corresponds to the path $\textcircled{n_7} \rightarrow \textcircled{n_6}$ $\dashrightarrow \textcircled{n_4} \rightarrow \boxed{n_1}$ in Figure 7a.

To save memory, BDDs are usually *reduced* by (R1) removing duplicated nodes (i.e., nodes that are the roots of structurally identical subBDDs), and (R2) deleting nodes with identical outgoing edges. In Figure 7a, R1 was performed but not R2, as the shaded node n_2 could be removed. Figure 7b shows a completely reduced BDD without these redundant nodes.

From here on, we will assume that BDDs are ordered and totally reduced. Thus, the algorithms we present in the next sections deal not only with the existing nodes in the BDD, but also with those removed due to R2.

In Section II-B, we saw that, in order to assist the application engineer to understand the impact of her decisions, it is convenient to restrict the configuration space according to a given set of selected/excluded features. Fortunately, most BDD libraries include a function called `restrict` that provides exactly this functionality [27].

Finally, Algorithm 1 shows Bryant's method [20] to traverse a BDD in a depth-first fashion, which will be used by our algorithms. `Traverse` is called at the top level with the BDD root as argument, and with a Boolean *mark* for every node being either all true or all false. `Traverse` visits all nodes by recursively visiting the low n_{LO} and high n_{HI} subBDDs rooted by n. Whenever a node is visited, its mark value is complemented. Comparing the marks of n and its children, it can be determined if they have already been visited. The method ensures that each node is visited exactly once and that, when traverse finishes, all node marks have the same value.

Algorithm 1. Bryant's method for BDD traversing

```
1 Function traverse(n)
2     mark(n) ← ¬mark(n)
3     if n is non-terminal then
4         if mark(n) ≠ mark(n_LO) then traverse(n_LO)
5         if mark(n) ≠ mark(n_HI) then traverse(n_HI)
6 traverse(ROOT)
```

B. Computing feature probabilities

Algorithm 2 (FIP) obtains, for each feature, the proportion of valid products that include it, i.e., $p(x|\Phi)$. To do so, it applies Bayes' rule to ultimately derive $p(x|\Phi)$ from $p(\Phi|n)$. First, the definition of conditional probability is used in Line 37: $p(x|\Phi) = \frac{p(x,\Phi)}{p(\Phi)}$; being $p(\Phi)$ and $p(x, \Phi)$ computed by the auxiliary Functions `getNodePr` and `getJointPr`.

1) Computing node probabilities: In a BDD, let us define the probabilities $p(n)$ and $p(\overline{n})$ for a node n as the number of paths that go from the root to the terminal nodes by traversing n through its high and low outgoing edges, respectively, divided by the total number of paths. Let us start reasoning on how to compute $p(n)$ when Reduction R2 has not been done yet. For instance, in Figure 7a, $p(n_6) = 2/8$ since there are eight paths in total from root to terminals, and two of them go through the high edge of n_6: $\textcircled{n_7} \rightarrow \textcircled{n_6} \rightarrow \textcircled{n_3} \rightarrow \boxed{n_0}$ and $\textcircled{n_7} \rightarrow \textcircled{n_6} \rightarrow \textcircled{n_3} \dashrightarrow \boxed{n_1}$

By construction, in a truth table every variable x is 1 half the rows, and it is 0 the other half. For instance, in Table III, there are four rows where B is 1, and there are other four rows where B is 0. This fact can be expressed as $p(x) = p(\overline{x}) = 1/2$. If R2 is not applied, $p(x) = p(\bigcup_{n \text{ labeled } x} n)$; being $p(\bigcup_{n \text{ labeled } x} n) = \sum_{n \text{ labeled }} p(n)$ because all BDD paths are mutually exclusive as they represent independent variable assignments. For example, in Figure 7a, $p(B) = p(n_5) + p(n_6) = 2/8 + 2/8 = 1/2$.

The first variable in the BDD ordering is represented by a single node: the root. So, $p(\text{ROOT}) = p(\overline{\text{ROOT}}) = 1/2$. The next variable in the ordering is encoded with two nodes ROOT_{HI} and ROOT_{LO} because every node has exactly two outgoing edges. Hence, the variable probability is shared out both nodes and thus $p(\text{ROOT}_{\text{HI}}) = p(\overline{\text{ROOT}_{\text{HI}}}) = \frac{1/2}{2}$, and $p(\text{ROOT}_{\text{LO}}) = p(\overline{\text{ROOT}_{\text{LO}}}) = \frac{1/2}{2}$. Proceeding this way, the node probabilities will be subsequently divided by two until the terminal nodes are reached. Finally, we need to be aware that whereas a node always has two outgoing edges, it may have any number greater than one of incoming

Algorithm 2. Feature Inclusion Probability (FIP)

```
1  Function getNodePr(n)
2      mark(n) ← ¬mark(n)
3      if n is non-terminal then
           // explore low
4          if n_LO is terminal then p(n_LO) ← p(n_LO) + p(n)
5          else p(n_LO) ← p(n_LO) + p(n)/2
6          if mark(n) ≠ mark(n_LO) then getNodePr(n_LO)
           // explore high
7          if n_HI is terminal then p(n_HI) ← p(n_HI) + p(n)
8          else p(n_HI) ← p(n_HI) + p(n)/2
9          if mark(n) ≠ mark(n_HI) then getNodePr(n_HI)
10 Function getJointPr(n)
11     mark(n) ← ¬mark(n)
12     if n is non-terminal then
           // explore low
13         if n_LO = n_0 then p(Φ|n̄) ← 0
14         else if n_LO = n_1 then p(Φ|n̄) ← 1
15         else
16             if mark(n) ≠ mark(n_LO) then getJointPr(n_LO)
17             p(Φ|n̄) ← p(Φ, n_LO ∨ n̄_LO) / (2p(n_LO))
18         p(n̄, Φ) ← p(Φ|n̄)p(n)
           // explore high
19         if n_HI = n_0 then p(Φ|n) ← 0
20         else if n_HI = n_1 then p(Φ|n) ← 1
21         else
22             if mark(n) ≠ mark(n_HI) then getJointPr(n_HI)
23             p(Φ|n) ← p(Φ, n_HI ∨ n̄_HI) / (2p(n_HI))
24         p(n, Φ) ← p(Φ|n)p(n)
           // combine both low and high
25         p(Φ, n ∨ n̄) ← p(Φ, n) + p(Φ, n̄)
26         p(var(n), Φ) ← p(var(n)) + p(n, Φ)
           // add joint probabilities of the removed nodes
27         foreach x_j between var(n) and var(n_HI) do
28             p(x_j, Φ) ← p(x_j, φ) + p(n, φ)/2
29         foreach x_j between var(n) and var(n_LO) do
30             p(x_j, Φ) ← p(x_j, φ) + p(n̄, φ)/2
31 p(ROOT) ← 1/2
32 p(n_i) ← 0 for all nodes n_i except the BDD root
33 getNodePr(ROOT)
34 p(x_j, Φ) ← 0 for all variables x_j
35 getJointPr(ROOT)
36 p(Φ) ← p(n_1)
37 foreach x_j do p(x_j|Φ) ← p(x_j, Φ) / p(Φ)
```

edges. Therefore, for a non-terminal node n with parents $u_1, u_2, \ldots u_s$, then $p(n) = \frac{\sum_{i=1}^{s} p(u_i)}{2}$; and for a terminal node, $p(n) = \sum_{i=1}^{s} p(u_i)$ (the parents' probability is not divided as the node has no outgoing edges).

Let us move now to realistic BDDs, where R2 is performed. In this case, we need to take into account the removed nodes:

$$p(x) = p\Big(\big(\bigcup_{n \text{ labeled } x} n\big) \cup \big(\bigcup_{\substack{n' \text{ labeled } x \\ \text{but removed}}} n'\big)\Big) = \sum_{n} p(n) + \sum_{n'} p(n')$$

Let us see how to compute the number of redundant nodes removed between any two nodes due to R2. If the variables follow the ordering $[x_1, x_2, \ldots, x_s]$, let $\text{var}(n)$ be the position of the variable that labels the node n in the ordering. For example, in Figure 7b, $\text{var}(n_4) = 2$ since n_4 is labeled B, and B is in the second position of the ordering $[A, B, C]$. Finally, let $\text{var}(n_0) = \text{var}(n_1) = s+1$. Then, $\text{var}(n_{\text{LO}}) - \text{var}(n) - 1$ is the number of nodes that have been removed between n and n_{LO}, and $\text{var}(n_{\text{HI}}) - \text{var}(n) - 1$ is the number of nodes that have been removed between n and n_{HI}. For example, as $\text{var}(n_0) - \text{var}(n_4) - 1 = 4 - 2 - 1 = 1$, it can be deduced that one node was removed in the high edge that goes from n_4 to n_0 (i.e., the shaded node n_2 in Figure 7a).

When a non-reduced BDD has a path $ⓤ \rightarrow ⓝ_1 \xrightarrow{\cdots} ⓝ_2 \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} ⓥ$, after applying R2 the path becomes $ⓤ \rightarrow ⓥ$ According to what was previously discussed above, $p(n_1) = p(u)/2$. For the rest of the nodes $n_2, n_3, \ldots, v$, the probability is not divided again since both the high and low edges go to the same node, e.g., $p(n_2) = \frac{p(n_{1\text{HI}}) + p(n_{1\text{LO}})}{2} = \frac{p(u)/2 + p(u)/2}{2} = p(u)/2$. To sum up, (i) the probability of the reduced nodes between any two nodes u and v is $p(u)/2$, and (ii) the probability of v is not affected by the amount of reduced nodes, being equal to $p(u)/2$ as well.

Function `getNodePr` combines the ideas discussed above with Bryant's traverse method. In Algorithm FIP, $p(\text{ROOT})$ is set to $1/2$, and $p(n)$ is initialized to 0 for the remaining nodes (Lines 31-32). Then, `getNodePr` traverses the BDD in pre-order to update $p(n)$. Finally, it is worth noting that $p(\Phi) = p(n_1)$ and $p(\overline{\Phi}) = p(n_0)$, being $p(\Phi)$ and $p(\overline{\Phi})$ the proportion of valid and invalid products of the VM, respectively.

2) Computing joint probabilities: Following the same argumentation line than in the previous section:

$$p(x, \Phi) = \sum_{n} p(n, \Phi) + \sum_{n'} p(n', \Phi)$$

Let us start first with the non-reduced nodes. By definition, $p(n, \Phi) = p(\Phi|n)p(n)$. As we rely on Bryant's recursive method to perform the computations, let us define $p(\Phi|n)$ in function of n high descendant (as the probability is conditioned to $n = 1$, in principle we only care about the high descendant). Two cases need to be considered:

1) When n_{HI} is terminal, (a) if $n_{\text{HI}} = n_0$ it means that the path is evaluated to 0, i.e., Φ is 0 for the variable assignment the path represents and so $p(\Phi|n) = 0$; (b) otherwise as $n_{\text{HI}} = n_1$ then $p(\Phi|n) = 1$.

2) When n_{HI} is non-terminal, $p(\Phi|n)$ is calculated as:

$$p(\Phi|n) = p(\Phi|n_{\text{HI}} \vee \overline{n_{\text{HI}}}) = \frac{p(\Phi, n_{\text{HI}} \vee \overline{n_{\text{HI}}})}{p(n_{\text{HI}} \vee \overline{n_{\text{HI}}})} = \frac{p(\Phi, n_{\text{HI}}) + p(\Phi, \overline{n_{\text{HI}}})}{p(n_{\text{HI}}) + p(\overline{n_{\text{HI}}})} = \frac{p(\Phi, n_{\text{HI}}) + p(\Phi, \overline{n_{\text{HI}}})}{2p(n_{\text{HI}})}$$

Equation 1 summarizes the cases above to compute $p(\Phi|n)$. As it needs knowing $p(\Phi, \overline{n_{\text{HI}}})$, Equation 2 is used (which is indeed the symmetrical case of Equation 1).

$$p(\Phi|n) = \begin{cases} 0 & \text{if } n_{\text{HI}} = n_0 \\ 1 & \text{if } n_{\text{HI}} = n_1 \\ \frac{p(\Phi, n_{\text{HI}}) + p(\Phi, \overline{n_{\text{HI}}})}{2p(n_{\text{HI}})} & \text{otherwise} \end{cases} \quad (1)$$

$$p(\Phi|\overline{n}) = \begin{cases} 0 & \text{if } n_{\mathrm{LO}} = n_0 \\ 1 & \text{if } n_{\mathrm{LO}} = n_1 \\ \frac{p(\Phi, n_{\mathrm{LO}}) + p(\Phi, \overline{n_{\mathrm{LO}}})}{2p(n_{\mathrm{LO}})} & \text{otherwise} \end{cases} \quad (2)$$

Function `getJointPr` in Algorithm FIP uses both Equations 1 and 2 to get the joint probability $p(x, \Phi)$ for non-removed nodes (Lines 13-26). Then, Equation 3 is applied to obtain $p(n', \Phi)$ for the removed nodes n' (Lines 27-30). It is worth noting that such equation follows the same reasoning presented in Section III-B1 to obtain $p(n')$.

$$p(n', \Phi) = \begin{cases} \frac{p(n,\Phi)}{2} & \text{if } n' \text{ was between } n \text{ and } n_{\mathrm{HI}} \\ \frac{p(\overline{n},\Phi)}{2} & \text{if } n' \text{ was between } n \text{ and } n_{\mathrm{LO}} \end{cases} \quad (3)$$

C. Computing product distribution

Algorithm 3 (PD) sketches the computation of the product distribution, accounting for how many products have no features, one feature, two features, ..., all features. It uses Bryant's method to traverse the BDD in post-order by calling the auxiliary Function `getProdDist` with the BDD root as argument. From the terminals to the root, it progressively obtains the partial distributions that correspond to the subBDDs rooted by each node, being the final distribution placed at the root.

Figure 8 shows each node's distribution for our running example, which is stored in different vectors *dist*. Starting from 0, the position i in a dist vector accounts for the number of products that have i features; e.g., $\mathrm{dist}(n_3) = [0, 1]$ because the subBDD with nodes n_0, n_1, and n_3 represents no products with zero features, and one product with one feature (i.e., product $\{C\}$).

`getProdDist`'s recursive base cases are node n_0, representing no products at all, and node n_1, representing a single product with no features. Accordingly, $\mathrm{dist}(n_0) = [\,]$ and $\mathrm{dist}(n_1) = [1]$ (Lines 24-25).

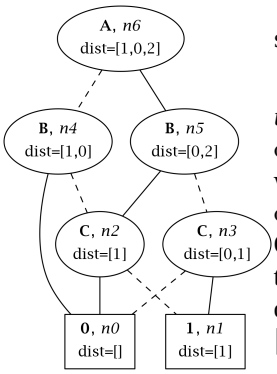


Fig. 8: *dist* vectors

To understand the more advanced recursive cases, three observations need to be done:

1) Including new features into all products is achieved by shifting the dist vector to the right (O1): Let us imagine that dist$= [1, 0, 4]$, i.e., there is 1 product with 0 features, 0 products with 1 feature, and 4 products with 2 features.

If no new features are added, dist remains the same. If one feature is added to all products, dist becomes $[0, 1, 0, 4]$, i.e., there are no products without features because all of them have at least the new feature, the product that had zero features now have 1 feature, and the 4 products that had 2 features now have 3 features. If two features are added to all products, dist becomes $[0, 0, 1, 0, 4]$, and so on.

Algorithm 3. Product Distribution (PD)

```
1  Function getProdDist(n)
2      mark(n) ← ¬mark(n)
3      if n is non-terminal then
           // traverse
4          if mark(n) ≠ mark(n_LO) then getProdDist(n_LO)
           // compute lowDist to account for the removed
              nodes through low
5          removedNodes ← var(n_LO) − var(n) − 1
6          let lowDist be a vector with removedNodes + length of
             dist(n_LO) zeros
7          for (i = 0; i ≤ removedNodes; i++) do
8              for (j = 0; j < length of dist(n_LO); j++) do
9                  lowDist[i + j] ←
                     lowDist[i + j] + dist(n_LO)[j] · C(removedNodes, i)
           // traverse
10         if mark(n) ≠ mark(n_HI) then getProdDist(n_HI)
           // compute highDist to account for the removed
              nodes through high
11         removedNodes ← var(n_HI) − var(n) − 1
12         let highDist be a vector with removedNodes + length of
             dist(n_HI) zeros
13         for (i = 0; i ≤ removedNodes; i++) do
14             for (j = 0; j < length of dist(n_HI); j++) do
15                 highDist[i + j] ←
                     highDist[i + j] + dist(n_HI)[j] · C(removedNodes, i)
           // combine low and high distributions
16         if lowDist is longer than highDist then
17             distLength ← length of dist(n_LO)
18         else distLength ← length of dist(n_HI) + 1
19         let dist(n) be a vector of length distLength filled with zeros
20         for (i = 0; j < length of lowDist; i++) do
21             dist(n)[i] ← lowDist[i]
22         for (i = 0; j < length of highDist; i++) do
23             dist(n)[i + 1] ← dist(n)[i + 1] + highDist[i]
24 dist(n_0) ← [ ] // no products
25 dist(n_1) ← [1] // one product with no features
26 getProdDist(ROOT)
27 return dist(ROOT)
```

In general, the addition of s features to all products means shifting dist s positions to the right.

2) Combining dist vectors is accomplished by adding them (O2): Let us think about how to get $\mathrm{dist}(n)$ from $\mathrm{dist}(n_{\mathrm{LO}})$ and $\mathrm{dist}(n_{\mathrm{HI}})$. First, let us suppose that no nodes were removed between n and its descendants. Imagine that $\mathrm{dist}(n_{\mathrm{LO}}) = [2, 0, 3]$ and $\mathrm{dist}(n_{\mathrm{HI}}) = [1, 2]$. According to O1, $\mathrm{dist}(n_{\mathrm{HI}})$ needs to be shifted one position to account for the additional feature that labels n. Then, both descendants distributions are combined by just adding them: $\mathrm{dist}(n) = [2, 0, 3] + [0, 1, 2] = [1, 1, 3]$.

3) Removed nodes require taking into account both observations O1 and O2, and blending them by means of combinatorial numbers (O3): If a non-reduced BDD had a path $u \dashrightarrow n_1 \overrightarrow{\dashrightarrow} n_2 \overrightarrow{\dashrightarrow} \ldots n_s \overrightarrow{\dashrightarrow} v$, R2 would remove the s redundant nodes, and thus the path would become $u \dashrightarrow v$. Hence, $\mathrm{dist}(u_{\mathrm{LO}})$ should be adjusted as any of the removed nodes could be set to 1, and so one new feature would be added to all products. Furthermore, any pair of redundant nodes $\binom{s}{2}$ could also be set to 1, any combination of three nodes $\binom{s}{3}$, ..., and finally the combination of s nodes $\binom{s}{s}$.

original dist(u_{LO})	1	0	4		
adding $\binom{2}{1}$ features	0	2	0	8	
adding $\binom{2}{2}$ features	0	0	1	0	4
adjusted final dist(u_{LO})	1	2	5	8	4

TABLE IV: Distribution adjustment

Let us see how the adjustment should work with an example: imagine that dist(n_{LO}) $= [1, 0, 4]$ and two nodes where removed between n and n_{LO}. Table IV summarizes the computations. The first and last rows represent the initial and adjusted distributions, respectively. The two intermediate rows represent the required adjustments.

First, adding one feature to all products implies shifting dist one position to the right (O1). As there are $\binom{2}{1} = \frac{2!}{1!(2-1)!} = 2$ different combinations of one feature, two shifted vectors should be added (O2). As a result, $[1, 0, 4]$ becomes $\left[0, 1 \cdot \binom{2}{1}, 0 \cdot \binom{2}{1}, 4 \cdot \binom{2}{1}\right] = [0, 2, 0, 8]$.

Second, there is only one possibility $\binom{2}{2} = 1$ to add two features to all products. So, $[1, 0, 4]$ becomes $[0, 0, 1, 0, 4]$.

Finally, all distributions are combined by adding them (O2): $[1, 0, 4] + [0, 2, 0, 8] + [0, 0, 1, 0, 4] = [1, 2, 5, 8, 4]$.

Lines 5-9 and 11-15 of Algorithm PD adjust the low and high distributions of the non-terminal nodes to account for the removed nodes. Then, Lines 16-23 combine both adjusted distributions.

D. Computational complexity

Both Algorithms FIP and PD traverse the whole BDD, and thus their complexity depends linearly on the number N of BDD nodes. Visiting each node requires (i) one loop on the number V of variables for FIP, and (ii) two nested loops on the variables for PD. As a result, the time complexities are $O(NV)$ and $O(NV^2)$ for FIP and PD, respectively.

IV. Experimental analysis of VMs

This section reports the analysis of seven VMs gathered from open source projects and academic repositories with the aim of illustrating the usefulness and generality of our approach. All the material described in this section (implementation of the FIP and PD algorithms, VM benchmark, BDD-encoding of the VMs, and results of the analysis) is available at the following public repository:

https://figshare.com/s/2f9f29494b16a0b88b87

A. Experimental setup

Our algorithms have been implemented as an extension of the library CUDD 3.0 for BDDs (https://github.com/vscosta/cudd). The benchmark is composed of VMs coming from different application domains and specified in distinct languages: (i) *axTLS*, *Fiasco*, *uClibc*, *Busybox*, and *EmbToolkit* are open source projects to enable the creation of highly customizable products, whose variability models are written in KConfig; (ii) the *Dell* feature model specifies a laptop configurator reverse-engineered from the DELL homepage; and (iii) *Automotive* is a feature model coming from the automotive industry. Table V summarizes (i) the models, (ii) the size of the BDDs that encode them, (iii) and our algorithms' running times on an HP ProLiant DL360 G9 with an Intel Xeon E5-2660v3.

VM name	VM notation	Reference	#Features	#Clauses	BDD #nodes	Running time FIP	Running time PD
axTLS 1.5.3	KConfig	http://axtls.sourceforge.net/	64	119	108	0.070s	0.035s
Dell Laptops	Feature Model	[11]	118	2,304	1,876	0.287s	0.273s
Fiasco 2014092821	KConfig	https://os.inf.tu-dresden.de/fiasco/	125	4,717	1,235	0.139s	0.104s
uClibc 201 50420	KConfig	https://www.uclibc.org/	306	903	4,862	0.506s	0.492s
Busybox 1.23.2	KConfig	https://busybox.net/	677	572	1,036	0.514s	0.582s
EmbToolkit 1.7.0	KConfig	https://www.embtoolkit.org/	1,815	7,193	263,636	15.117s	14.647s
Automotive 02	Feature model	[28]	17,365	321,933	30,432	3m 22.007s	3m 35.713s

TABLE V: VM benchmark

VM name	Mean	SD	Min	Max	$p = 0$	$p \leq 0.05$	$p = 0.5$	$0.475 \leq p \leq 0.525$	$p = 1$	$p \geq 0.095$
axTLS	25.46	10.46	3	46	0	9.38	6.25	37.50	0	3.12
Dell	17.50	2.24	14	21	0	47.46	0	2.54	0.85	0.85
Fiasco	24.84	9.70	4	44	31.20	46.40	15.20	24.80	0	1.60
uClibc	106.49	46.13	8	200	2.61	23.86	25.49	35.29	0	2.94
Busybox	324.44	149.05	5	635	2.95	3.55	37.81	53.91	0.44	3.10
EmbToolkit	741.49	330.91	19	1,398	6.23	11.9	25.73	38.95	1.21	5.29
Automotive	4,048.48	778.7	2,562	5,472	0.03	57.31	13.92	18.66	9.71	10.39

TABLE VI: Descriptive statistics for product distribution, and percentage of dead, core, and unconstrained optional features

B. Results

Our approach enables reasoning on VMs under two perspectives:

- *The products' view*. Table VI provides descriptive statistics for the VMs' product distribution regarding their number of features, and Figure 9 visualizes that distribution.
- *The features' view*. Figure 10 shows the feature probability distribution, and colored columns in Table VI detail the number of features in the zones *dead*, *unconstrained optional*, and *core*.

The product distribution graphs (Fig. 9) and feature probability distribution graphs (Fig. 10) (respectively Table VI) highlight the existence of two rough VM groups. In the first group, axTLS, uClibc, Busybox, and EmbToolkit represent families of loosely constraint products. Valid products may range from consisting of only a few features (as low as three features for axTLS), to close to all features (e.g., over 90% of all features in the case of Busybox). Hence, also the feature probability distribution graphs for these models show more features in the range $0.475 \leq p \leq 0.525$ compared to the range $p \leq 0.05$. In contrast, the second group consisting of Dell Laptops, Fiasco, and Automotive, represents SPLs with rather restricted products. Valid products may contain at a maximum 18%, 35%, and 32%, respectively, of available features compared to the first group with 72%, 65%, 94%, and 77% respectively. SPLs in the second group also tend to come with highly rare features. Between 46% and 57% of all features have a reusing probability less or equal than 0.05. A detailed list of all feature probabilities for every VM in the benchmark is published at our repository. This list will help domain engineers to polish their VMs, especially for Fiasco,

which has a surprisingly high percentage of dead features: 31.2%. For Dell Laptops and due to the sensitivity augment that our FIP algorithm provides, some low reusable features are discovered where current approaches do not detect any problem at all: although there are no dead features, 17.8% of Dell's features are allowed in at most 0.001% of the valid products.

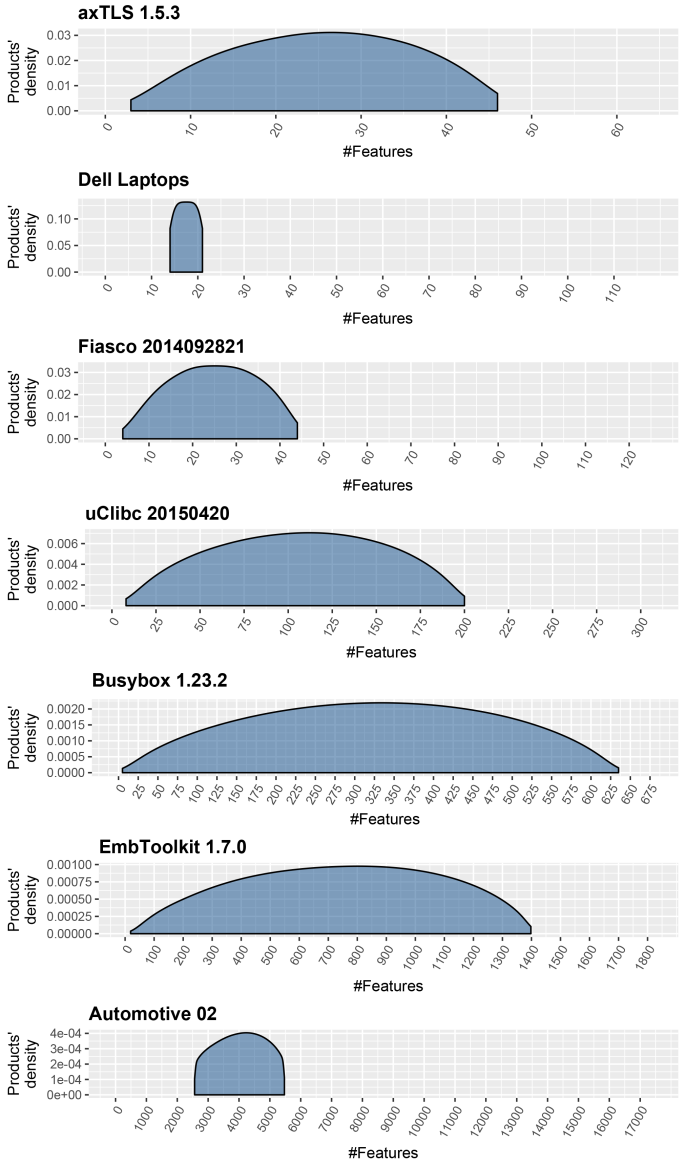


Fig. 9: Product distribution

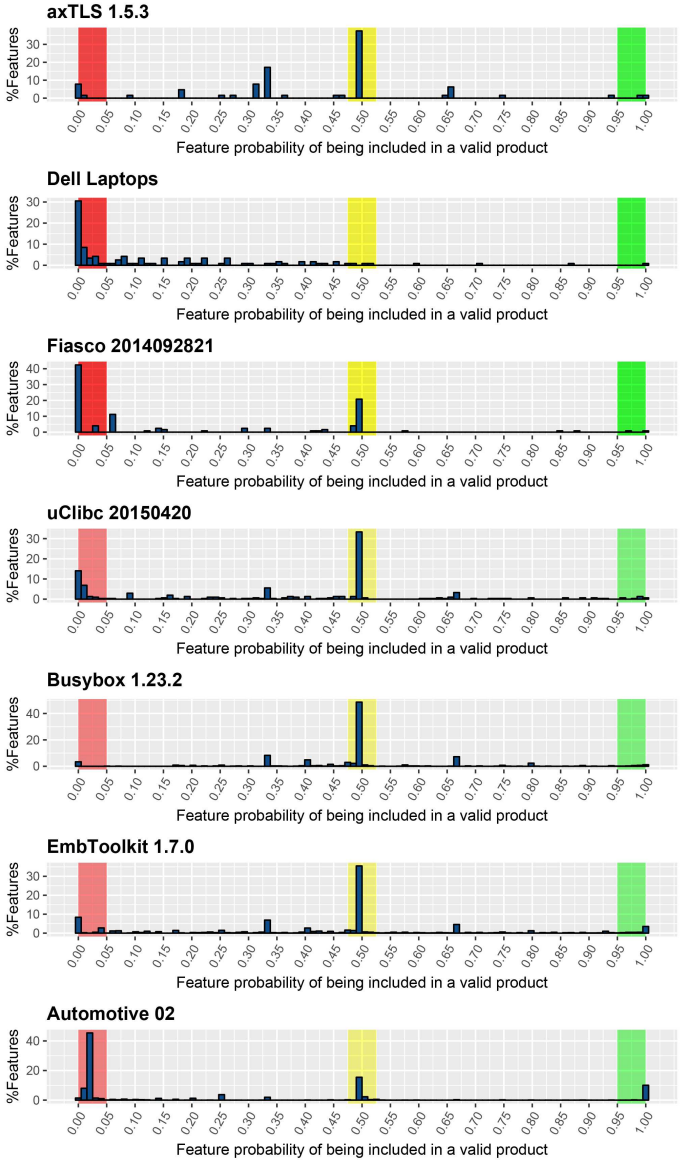


Fig. 10: Feature probability distribution

V. Related work

The seminal work by Kang et al. [7] established what has been the mainstream for visually representing VMs from 1990 to nowadays: graphs whose nodes depict features, and whose edges represent inter-feature constraints. The most popular notation is feature modeling [5], which puts the emphasis on those constraints that enable arranging the features hierarchically as a tree [3]. There are also other graph notations, e.g., decision diagrams [29], the OVM language [30], etc. Nevertheless, the differences among notations are minor, and so most approaches can be considered equivalent [4].

As in practice variability models can include thousands of features [23], some efforts have been made to clarify the visualization of large graph VM representations: applying 3D techniques to visualize the graphs in the space instead of the plane [31], supporting zooming on different graph areas [32], focusing the visualization on a selected feature [33], decomposing the graphs [34], etc. Our work complements existing research by introducing an alternative way to look at VMs through statistics, supporting thus the use of centrality/spread measures, plots, etc.

Sections V-A and V-B discuss related work that aims to assist domain and application engineering, respectively.

A. Domain engineer's assistance

A literature review by Benavides et al. [6] reports thirty analysis operations on VMs, most of them oriented to domain engineering. This paper supports augmenting the sensitivity of some of those operations. For instance, a feature is typically considered dead if it cannot appear in any product at all. The main reason why most approaches stick to this definition for detecting dispensable features is due to the current limitations of the technology they are built upon, as they detect whether a feature f in a VM Φ is dead by checking with a SAT solver if $f \wedge \Phi$ is unsatisfiable [28]. In contrast, our algorithms support a more flexible definition, detecting features with an extremely low probability of being selected.

Beek et al. [15], [35], [36] point out the convenience of providing the domain engineer with information about the product distribution regarding distinct quantitative attributes (e.g., number of features, product cost, failure probability, etc.). To do so, their approach requires (i) that the domain engineer sets manually the feature probabilities, or (ii) that the feature probabilities are derived from historical data. Then, the product distribution is estimated by generating multiple samples through a simulation process. Compared to Beek et al.'s method, our procedure provides the exact product distribution instead of an approximation. Nevertheless, Algorithm PD currently supports only one quantitative attribute, the number of features, and could be extended to consider domain specific properties.

B. Application engineer's assistance

There are several approaches to guide the application engineer through product configuration. Some of them are built upon historical data about previous configurations. For instance, Pereira et al. [12], [14] proposes a recommender system that limits the engineer's decision space towards configurations included in historical data. In addition, Martinez et al. [37] provide the engineer with feedback on the impact of her decisions by estimating the feature probabilities from historical data. These approaches have several shortcomings: first, the historical data may not be a representative sample of the product population, especially if the sample size is small and its variance is high; and most important, feature selectivity cannot be strictly constrained to a sample. For example, if a non-dead feature is not included in any configuration of the historical data, then the system could conclude erroneously that the engineer should never select such feature.

Other approaches, instead of relying on previous configurations, work directly with the VM. For example, Czarnecki et al. [9] suggest the application of the *entropy* measure to guide the VM configuration process, which is calculated from the feature probabilities. In addition, Nöhrer et al. [10], [11] propose an alternative heuristic, also based on the feature probabilities. However, none of those works scale to large VMs.

Fernandez-Amoros et al. [38] provide an algorithm to compute the feature probabilities from a feature model. However, the algorithm is specific for feature models and it does not scale when many constraints cross the tree structure of the feature model.

To the extent of our knowledge, Algorithm FIP is the most scalable and general approach to compute the feature probabilities from a VM. This way, our work not only supports the configuration heuristics that rely on the feature probabilities obtained from the VM, but also the ones based on historical data. In the latter case, our algorithms can be used to overcome the limitations of reasoning exclusively on the basis of a single product sample by applying Bayesian inference [39] to combine both the *prior* probabilities coming from the VM with the *posterior* probabilities coming from historical data.

VI. Conclusions and future work

In this paper, the algorithmic foundation for analyzing VMs from an innovative perspective has been presented, where the features' and products' variation is visualized and described using statistics. We have justified why this approach benefits both, the domain and the application engineer, exemplifying such benefits on real models gathered from open source projects and academic repositories. We have shown that our approach not only enables new ways to reason about VMs, but also supports the improvement of current VM-related methods: increasing the sensitivity of existing analysis operations on VMs, and providing exact computations for approaches that currently work with approximations.

We believe that our work opens a range of additional opportunities for future research. Applied to other product line related activities, such as testing, our work enables checking whether current methods for SPL testing are able to generate suites covering the whole product distribution range, and thus avoid missing any rare boundary cases. Also, our approach may be of assistance during maintenance of projects for highly customizable software; e.g., we have reported that the VMs of some relevant open source projects have an alarming amount of dead features. The causes of those useless features need investigation. A longitudinal study would provide insights under which circumstances these projects exhibit these problems, and whether they are corrected or stay in the successive versions of the VMs. Finally, our algorithms rely on the BDD encoding of VMs. It is well-known that a BDD's size is extremely sensitive to its variable ordering, and that finding an optimal ordering is an NP-complete problem. Therefore, our approach's scalability greatly depends on the performance of existing heuristics for variable ordering. Hence, future research might look for adapting our algorithms to other alternative logic technologies that also support model counting, such as *#SAT solvers* [40] or *Sentential Decision Diagrams* (SDDs) [41].

Acknowledgments

We thank Armin Biere and Tom van Dijk for their insight and helpful comments about the strengths and weaknesses of BDDs, and other logic related technologies in the earlier stages of this work.

REFERENCES

[1] R. Heradio, H. Perez-Morago, D. Fernandez-Amoros, F. J. Cabrerizo, and E. Herrera-Viedma, "A bibliometric analysis of 20 years of research on software product lines," *Information and Software Technology*, vol. 72, pp. 1 – 15, 2016.

[2] W. Cazzola and A. Shaqiri, "Context-aware software variability through adaptable interpreters," *IEEE Software*, vol. 34, no. 6, pp. 83–88, November 2017.

[3] P. Heymans, P. . Schobbens, J. . Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen, "Evaluating formal properties of feature diagram languages," *IET Software*, vol. 2, no. 3, pp. 281–302, June 2008.

[4] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: a comparison of variability modeling approaches," in *6th Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, Leipzig, Germany, 2012, pp. 173–182.

[5] R. E. Lopez-Herrejon, S. Illescas, and A. Egyed, "A systematic mapping study of information visualization for software product line engineering," *Journal of software: evolution and process*, vol. 30, no. 2, pp. 1–18, 2018.

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon University/Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

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

[9] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," in *12th Int. Software Product Line Conference (SPLC)*, Limerick, Ireland, Sept 2008, pp. 22–31.

[10] A. Nöhrer and A. Egyed, "Optimizing user guidance during decision-making," in *15th Int. Software Product Line Conference (SPLC)*, Munich, Germany, Aug 2011, pp. 25–34.

[11] A. Nöhrer and A. Egyed, "C2O configurator: a tool for guided decision-making," *Automated Software Engineering*, vol. 20, no. 2, pp. 265–296, Jun 2013.

[12] J. A. Pereira, P. Matuszyk, S. Krieter, M. Spiliopoulou, and G. Saake, "A feature-based personalized recommender system for product-line configuration," in *ACM SIGPLAN Int. Conference on Generative Programming: Concepts and Experiences (GPCE)*, New York, NY, USA, 2016, pp. 120–131.

[13] R. Heradio, D. Fernandez-Amoros, J. A. Cerrada, and I. Abad, "A literature review on feature diagram product counting and its usage in software product line economic models," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 8, pp. 1177–1204, 2013.

[14] J. A. Pereira, J. Martinez, H. K. Gurudu, S. Krieter, and G. Saake, "Visual guidance for product line configuration using recommendations and non-functional properties," in *33rd Annual ACM Symposium on Applied Computing (SAC)*, New York, NY, USA, 2018, pp. 2058–2065.

[15] M. H. ter Beek, A. Legay, A. Lluch-Lafuente, and A. Vandin, "Quantitative analysis of probabilistic models of software product lines with statistical model checking," in *6th Workshop on Formal Methods and Analysis in SPL Engineering (FMSPLE@ETAPS)*, London, UK, Apr. 2015, pp. 56–70.

[16] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas," in *9th Software Product Lines Conference (SPLC)*, Rennes, France, Sep. 2005, pp. 7–20.

[17] T. Berger and S. She, "Formal Semantics of the CDL Language," University of Leipzig, Tech. Rep., 2010.

[18] R. Tartler, "Mastering Variability Challenges in Linux and Related Highly-Configurable System Software," Ph.D. dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[19] S. She and T. Berger, "Formal semantics of the kconfig language," University of Waterloo, Tech. Rep., 2010.

[20] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug 1986.

[21] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 675–696, Nov 2017.

[22] S. Apel, D. Batory, and C. Kastner, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[23] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, Dec 2013.

[24] P. C. Clements, J. D. McGregor, and S. G. Cohen, "The Structured Intuitive Model for Product Line Economics (SIMPLE)," Carnegie Mellon University/Software Engineering Institute, Tech. Rep. CMU/SEI-2005-TR-003, 2005.

[25] H. Perez-Morago, R. Heradio, D. Fernandez-Amoros, R. Bean, and C. Cerrada, "Efficient identification of core and dead features in variability models," *IEEE Access*, vol. 3, pp. 2333–2340, 2015.

[26] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer, 1998.

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

[28] S. Krieter, T. Thüm, S. Schulze, R. Schröter, and G. Saake, "Propagating configuration decisions with modal implication graphs," in *40th Int. Conference on Software Engineering (ICSE)*, New York, NY, USA, 2018, pp. 898–909.

[29] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, New York, NY, USA, 2011, pp. 119–126.

[30] K. Pohl, F. V. D. Linden, and G. Bockle, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, Ed. Springer, 2005.

[31] P. Trinidad, A. R. Cortés, D. Benavides, and S. Segura, "Three-dimensional feature diagrams visualization," in *12th Int. Software Product Lines Conference (SPLC)*, Limerick, Ireland, Sep. 2008, pp. 295–302.

[32] M. Stengel, M. Frisch, S. Apel, J. Feigenspan, C. Kastner, and R. Dachselt, "View infinity: a zoomable interface for feature-oriented software development," in *33rd International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, May 2011, pp. 1031–1033.

[33] M. Garba, A. Noureddine, and R. Bashroush, "Musa: A scalable multi-touch and multi-perspective variability management tool," in *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Venice, Italy, April 2016, pp. 299–302.

[34] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser, "A visual support for decomposing complex feature models," in *IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, Bremen, Germany, Sept 2015, pp. 76–85.

[35] M. H. ter Beek, A. Legay, A. Lluch-Lafuente, and A. Vandin, "Statistical model checking for product lines," in *7th Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Corfu, Greece, Oct. 2016, pp. 114–133.

[36] M. T. Beek, A. Legay, A. L. Lafuente, and A. Vandin, "A framework for quantitative modeling and analysis of highly (re)configurable systems," *IEEE Transactions on Software Engineering*, p. (Early Access), 2018.

[37] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyand, J. Klein, and Y. L. Traon, "Feature relations graphs: A visualisation paradigm for feature constraints in software product lines," in *2nd IEEE Working Conference on Software Visualization (VISSOFT)*, Victoria, BC, Canada, Sept 2014, pp. 50–59.

[38] D. Fernandez-Amoros, R. Heradio, J. A. Cerrada, and C. Cerrada, "A scalable approach to exact model and commonality counting for extended feature models," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 895–910, Sept 2014.

[39] J. K. Kruschke, *Doing Bayesian Data Analysis, 2nd Edition: a Tutorial with R, JAGS, and Stan*. Academic Press/Elsevier, 2015.

[40] C. P. Gomes, A. Sabharwal, and B. Selman, *Handbook of Satisfiability*. IOS Press, 2009, ch. Model Counting, pp. 633–654.

[41] A. Darwiche, "SDD: A New Canonical Representation of Propositional Knowledge Bases," in *22nd Int. Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 819–826.