

Optimizing the Product Derivation Process

Sheng Chen
School of EECS
Oregon State University
chensh@eecs.oregonstate.edu

Martin Erwig
School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

Abstract—Feature modeling is widely used in software product-line engineering to capture the commonalities and variabilities within an application domain. As feature models evolve, they can become very complex with respect to the number of features and the dependencies among them, which can cause the product derivation based on feature selection to become quite time consuming and error prone.

We address this problem by presenting techniques to find good feature selection sequences that are based on the number of products that contain a particular feature and the impact of a selected feature on the selection of other features. Specifically, we identify a feature selection strategy, which brings up highly selective features early for selection. By prioritizing feature selection based on the selectivity of features our technique makes the feature selection process more efficient. Moreover, our approach helps with the problem of unexpected side effects of feature selection in later stages of the selection process, which is commonly considered a difficult problem. We have run our algorithm on the e-Shop and Berkeley DB feature models and also on some automatically generated feature models. The evaluation results demonstrate that our techniques can shorten the product derivation processes significantly.

Keywords—Feature Model, Feature Selection, Decision Sequence

I. INTRODUCTION

Software product lines (SPLs) are increasingly employed in the software development process since they support systematic reuse and, in particular, facilitate customization for particular needs [1], [2]. Basically, SPL engineering consists of two processes: domain engineering and application engineering. The first process is responsible for defining software products in terms of commonalities and variabilities and the constraints among them. An effective and commonly used way of doing this is feature modeling [2]. Figure 1 shows the feature model for a simple mobile phone product line. The second process involves the derivation of products from feature models.

Full exploitation of the benefits of the SPL paradigm highly depends on how efficient the second process is, because whenever we want to derive a new product, we need to go through this process. However, making decisions in a feature model to obtain a product is not always straightforward because (a) the feature model can be very complex (it may contain thousands of features [3]) and (b) the dependencies among features constrain the selection process

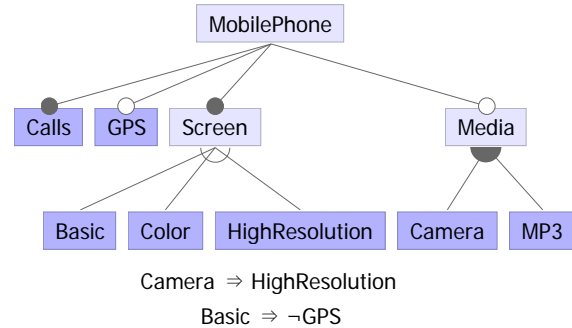


Figure 1. The feature model of mobile phone

since the selection of a particular feature may introduce some undesired features or exclude some expected features.

For these reasons, users always face the problem of where to start when deriving a product from a feature model. Some work has been done to address this issue, which is discussed in depth in Section V. However, no work seems to systematically exploit the fact that the selection of a feature that participates in constraints effectively triggers the automatic selection or deselection of other features. E.g. in Figure 1, feature *Camera* is more selective than feature *High resolution* in sense that if *Camera* is chosen, then *High resolution* is also selected. However, if a user makes decision between *Screen* first and chooses *High resolution*, then he still needs to decide whether he will take *Camera*.

We can generalize this idea by associating with each feature a value called *selectivity*, which indicates the impact of selecting that feature on the selection and removal of other features. The selectivity of a feature is determined by two factors: (1) the number of products that can be derived containing the feature (a.k.a. *commonality* [4]) and (2) the number of features that will be automatically selected or removed by selecting that feature due to constraint propagations. While previous work used the number of products (a.k.a. *variation degree* [5]) as an indicator for the flexibility and complexity of a feature model and commonality as an indicator for the importance of features, we employ the combination of these two measures to guide users in where to make decisions, which can accelerate the decision making process.

It should be clear that the most selective feature in a feature model is not necessarily the most important feature in a product. Selecting the most important feature first is a straightforward way of deriving a product, but it is usually not the most efficient way. Our technique only makes suggestions to accelerate product derivation; users can still select the most important feature first.

To calculate selectivity for each feature efficiently, we first transform a feature model into an algebraic expression, called *choice description*, which is a reduced representation of a feature model carrying sufficient information for computing selectivity. However, instead of manipulating the choice description transformed from a feature model as a whole, we employ a divide-and-conquer strategy that partitions a choice description into a set of smaller choice descriptions such that deriving a product from the original choice description can be achieved by deriving a smaller product from each of the smaller choice descriptions and combining them. This strategy is crucial for the efficiency of our approach in the face of huge variation degrees and commonalities. A side benefit of this strategy is that it supports parallel decision making. As SPLs are widely adopted and feature models get larger, staged configuration and parallel decision making will become more important [6], [7].

The main contribution of this paper is a selectivity-driven strategy to support the feature selection process that can shorten decision sequences considerably. Since in application engineering product derivation is repeated many times, whenever a new product is to be built, even a minor efficiency improvement for deriving each product leads to significant benefits over time. We report the results of an empirical study that shows noticeable efficiency improvement for deriving products. For example, we found that decision sequences can be shortened on average at least by 6% and up to 15% for the e-Shop [8] software product line. For the Berkeley DB feature model [9] the improvements were at least 12% and as large as 35%. The maximum improvements were even higher.

The remainder of the paper is organized as follows: In Section II, we present formal definitions and notations needed to describe our algorithm, which is then developed in Section III. We report results of an empirical study in Section IV. Section V compares our approach with related work, followed by Section VI, which concludes the paper and describes future work.

II. DEFINITIONS AND NOTATIONS

This section introduces the concept of choice description and introduces the notation used in the paper. A feature model concisely defines all valid configurations in a software product line. A feature model mainly consists of a feature tree and cross-tree constraints, which are usually expressed

with propositional formulas. A feature tree represents features hierarchically to express the child-parent relationship among features. The relationship among children can be AND, OR, and XOR. When AND is used, the child can be either Optional or Mandatory. A comprehensive and precise definition of feature models is provided in [10].

We will represent feature models as algebraic expressions, which facilitates a decomposition of expressions to make the computation of feature selectivity feasible. In the following we will first define the concepts of this algebraic structure and then define operations that will be needed in the rest of the paper.

A *choice algebra* $(A, \cdot, +)$ is given by a set A containing a unit element 1 plus two operations \cdot and $+$ such that:

- (i) (A, \cdot) is a semigroup,
- (ii) $+$ is associative, commutative and idempotent (that is, $(A, +)$ is a commutative, idempotent semiring),
- (iii) \cdot distributes over $+$, and
- (iv) $0 + e = e + 0 = e$ and $1 \cdot e = e \cdot 1 = e$.

Expressions over A are called *choice expressions*. Expressions formed using $+$ are called *choices*. An expression that does not contain any choice is called *plain*. Choice algebra is similar to feature algebra introduced in [11], although there are some important differences, which we will discuss in Section V.

Each choice-algebra expression describes a set of plain expressions that can be derived from it by selecting elements from choices. When we use choice algebra to represent feature models, the set A represents a set of features, and the set of plain expressions that can be derived from a choice expression is essentially the set of products that can be derived from a feature model.

The products or plain expressions derivable from a choice expression are computed by the following function π .

$$\pi(e) = \begin{cases} \pi(e_1) \times \dots \times \pi(e_n) & e = e_1 \cdot \dots \cdot e_n \\ \cup_{i=1}^n \pi(e_i) & e = e_1 + \dots + e_n \\ \{e\} & e \in A \end{cases}$$

We say that choice expression e is *linear* if each symbol appears at most once in e . For example, the expression $a + b$ is linear, while expression $a(a + b)$ is not. Later we will see that all choice expressions transformed from feature trees are linear.

A *choice description* is a pair $D = (e, c)$ where e is a choice expression and c is a propositional formula over $\varphi(e)$, called *constraint*. We say that a constraint c is *satisfiable* on a choice expression e if there is a plain expressions derivable from e that satisfies c . Similarly, c is a *tautology* on e if all plain expressions derivable from e satisfy c . For example, $a \rightarrow c$ and $b \rightarrow c$ are satisfiable and a tautology on $(a + b)c(d + e)$. For constraints in feature models, satisfiability can be determined efficiently.

We use $\varphi(e)$ to denote the set of features contained in choice expression e . Similarly, $\varphi(c)$ denotes the features contained in constraint c . For example, $\varphi(1+a) = \{a\}$ and $\varphi(a \rightarrow c) = \{a, c\}$. The *optional symbols* in a choice expression e are all features that are operands of the $+$ operator, while *necessary symbols* are the symbols that will appear in each plain expression derivable from e . For example, in $e = a(b+cd)$, optional symbols are $\{b, c, d\}$ and the only necessary symbol is a .

The functions $e \uparrow f$ and $e \downarrow f$ to select or drop a feature from an expression are inductively defined as follows.

$$e \uparrow f = \begin{cases} e_1 \dots (e_i \uparrow f) \dots e_n & f \in \varphi(e_i) \wedge e = e_1 \dots e_n \\ e_i \uparrow f & f \in \varphi(e_i) \wedge e = e_1 + \dots + e_n \\ f & f = e \\ 1 & f \notin \varphi(e) \end{cases}$$

$$e \downarrow f = \begin{cases} e_1 \dots (e_i \downarrow f) \dots e_n & f \in \varphi(e_i) \wedge e = e_1 \dots e_n \\ e_1 + \dots + e_i \downarrow f + \dots + e_n & f \in \varphi(e_i) \wedge e = e_1 + \dots + e_n \\ 1 & f = e \\ e & \text{otherwise} \end{cases}$$

When e contains each symbol only once, the operations are uniquely defined. Otherwise, we pick or drop the leftmost-outermost occurrence of f only.

A *rejected feature*, written as \bar{f} , is a feature that was not selected. A *literal* ℓ is a selected or rejected feature.

We can restrict a choice expression e by a literal ℓ using the operation $e|\ell$, which is defined as follows.

$$e|\ell = \begin{cases} e \downarrow f & \ell = \bar{f} \\ e \uparrow \ell & \text{otherwise} \end{cases}$$

Restricting a choice expression by a literal results in either a new choice expression or the unit expression 1 if the selected feature does not exist in the expression.

Restriction extends in a natural way to lists of literals. For example, restricting $(a+b)(c+d+e)$ by the literals $a\bar{e}$ works as follows.

$$\begin{aligned} & (a+b)(c+d+e)|a\bar{e} \\ &= ((a+b)(c+d+e)|a)|\bar{e} \\ &= a(c+d+e)|\bar{e} \\ &= a(c+d) \end{aligned}$$

The sets $\text{inclusion}_f(D)$ and $\text{inclusion}_{\bar{f}}(D)$ contain those features that are automatically selected after a feature f has been selected or dropped, respectively. Note that features can be implied either due to expression structure or due to constraints. Consider, for example, the following choice expression.

$$((a+b(c+d))(e+f)(g+h), \{b \rightarrow f, d \rightarrow h\})$$

Selecting d will imply the selection of h (because of the constraint $d \rightarrow h$) and b (due to the structure of the expression), which in turn implies f (again due to a constraint).

We write $\#D$ for the number of products that can be derived from D and $\gamma_f(D)$ for the commonality of feature f in the choice description D . We use $\sigma_f(D)$ to denote the selectivity of feature f in D , which is defined as follows.

$$\sigma_f(D) = \gamma_f(D)/\#D$$

A sequence of selected and rejected features that will yield a product from a choice description is called a *decision sequence*.

A *product derivation* (or *derivation* for short) is a sequence of picked, dropped, and automatically selected features that yield a product from a choice description. We write \check{f} for an automatically selected feature. Note that we can derive a decision sequence from a product derivation by simply removing all automatically selected features. A *selectivity-driven product derivation (SDPD)* is a product derivation in which the features appear in order of their selectivity (with respect to the given choice description).

As an example, consider the task of deriving the product $bdfh$ from the following choice description

$$((a+b(c+d))(e+f)(g+h), \{b \rightarrow f, d \rightarrow h\})$$

It turns out that even for this small example, there are 66 possible decision sequences to derive the product. Here are some sample sequences $d\check{b}\check{f}\check{h}$, $b\check{f}d\check{h}$, $fhbd$, $\bar{a}\check{b}\check{f}\check{c}\check{d}\check{h}$, $\bar{c}\check{d}\check{b}\check{f}\check{h}\bar{a}$, $b\check{f}\bar{c}\check{d}\check{h}$.

We can observe that the derivation $d\check{b}\check{f}\check{h}$ includes a decision sequence of length 1 (namely, d). In contrast, the derivation $fhbd$ entails the longest possible decision sequence of length 4. This shows that SDPD achieves a maximum improvement of 75% in this case. The average length of all 66 sequences is about 3.45, and hence the average efficiency improvement of SDPD for this example is $(3.45 - 1)/3.45$, that is, roughly 71%.

Note that while the length of the product derivation for $bdfh$ is not larger than the number of features in the product, this is not always the case. For instance, the product derivation $\bar{c}\bar{d}\check{a}\check{f}h$ for deriving product afh is longer than 3.

Finally, note that automatically *excluded* features are not relevant in this context and don't have to be considered since they don't appear in the final product and since they require no decision. The reason why we need to consider automatically included features is that they appear in the final product.

III. COMPUTING DECISION SEQUENCES

This section elaborates the process of supporting decisions on feature models to derive products. Given a feature model, we first show in Section III-A how to transform it into a choice description, which is the basis for our method. In Section III-B we describe the method of selectivity-driven

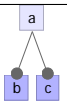
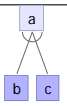
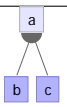
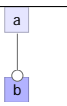
Feature Tree	Choice Description
	a requires both b and c (abc, \emptyset)
	a requires either b or c , but not both $(a(b+c), \emptyset)$
	a requires b or c or both $(a(1+b)(1+c), \{b \vee c\})$
	b is an optional feature of a $(a(1+b), \emptyset)$

Figure 2. Translating feature models to choice descriptions

product derivation on a high level. In the following sections, we then provide details on how to make this method feasible: In Section III-C we describe the computation of selectivity, and in Sections III-D and III-E we explain the concept of choice partitions and decompositions and how to compute them. Finally, in Section III-F we comment on the seemingly high overhead that our method incurs, which will also point to other potential uses of choice partitions.

A. Mapping Feature Models to Choice Descriptions

Feature models are frequently transformed into other representations to support reasoning about them. In our case, we map feature models to choice descriptions to support the efficient computation of feature selectivity.

The rules for mapping feature trees to choice expressions are summarized in Figure 2. Most of the rules are straightforward, except maybe for the OR relation in feature models. If $a = b \text{ OR } c$, then b or c or both of them must be selected. Hence we transform the tree into $(1+b)(1+c)$ plus a constraint $b \vee c$. The reason we don't simply transform the tree into the expression $(b+c+bc)$ is that this expression is not linear; it contains both features b and c twice. Linearity of choice descriptions makes the selection and removal operations more efficient. Moreover, our approach scales better than the seemingly shorter alternative. For example, if a has one more child d , the expression will contain 7 alternatives. Instead, our approach handles this with ease, we transform it into $a = (1+b)(1+c)(1+d)$ together with the constraint $b \vee c \vee d$.

The transformation of a feature tree to a choice expression then begins with the root of the feature diagram, and traverses the tree replacing all occurrences of abstract features by the choice expressions that represent their definition multiplied with the abstract feature. As a result, we obtain a single choice expression for the feature tree. Finally, we obtain the choice description for the feature model by

combining the choice expression and the constraints for the feature model.

Applying the transformation rules to the feature model from Figure 1, we obtain the following choice description.¹

$$(l(1+g)s(b+c+h)(1+d(1+a)(1+m)), \\ \{a \rightarrow h, b \rightarrow \neg g, a \vee m\})$$

Note that we have to keep abstract features in the translation to choice descriptions, because they can appear in constraints.

B. Selectivity-Driven Product Derivation

Next we present the algorithm for deriving products by incremental decision making for the currently most selective feature. To simplify the algorithm presentation by omitting the user interaction, it assumes an “intended” product p , which is the product that a user has in mind, but which has not been derived yet. This product p serves as a guideline to decide the selection or rejection of features that are presented by the algorithm. Since the algorithm accepts any p , this assumption is not restrictive in any way.

INPUT: Choice description (e, c) and intended product p

OUTPUT: A product derivation s

METHOD:

- (1) $s := []$
- (2) Find most selective feature f of e
- (3) **if** $f \notin p$ **then** $f := \bar{f}$
- (4) $s := s \cdot f \cdot \text{inclusion}_f(e, c)$
- (5) $e := e|f$
- (6) **if** $\phi(p) \not\subseteq \phi(s)$ **then** goto (2)
- (7) **return** s

The algorithm requires the repeated computation of selectivity and adds the feature with the highest selectivity to the product derivation s . If f is not chosen (because it is not in the envisioned product p), \bar{f} will be included as a rejected feature. (Note that \cdot denotes list concatenation.) Then the choice expression will be restricted, and the process is repeated until all features for the product have been selected.

C. Calculating Feature Selectivity

The choice description representation of feature models facilitates an inductive definition for the computation of selectivity. The computation of the number of products and the commonality of a feature in a choice expression are both easy to compute, but this is nontrivial in the presence of constraints.

Therefore, we next describe the computation of selectivity for choice expressions and show in the next two sections how to transform a choice description into a choice expression through the systematic removal of constraints.

¹Feature names are abbreviated by their first letter, except for Calls, Camera, and Media.

The computation of selectivity makes use of the number of products derivable from a choice expressions whose definition is straightforward.

$$\#(e) = \begin{cases} \prod_{i=1}^n \#(e_i) & e = e_1 \dots e_n \\ \sum_{i=1}^n \#(e_i) & e = e_1 + \dots + e_n \\ 1 & e \text{ is a feature} \end{cases}$$

The commonality for a feature then can be recursively computed as follows.

$$\gamma_f(e) = \begin{cases} \gamma_f(e_j) \prod_{i \neq j} \#(e_i) & f \in \Phi(e_j) \wedge e = e_1 \dots e_n \\ \gamma_f(e_j) & f \in \Phi(e_j) \wedge e = e_1 + \dots + e_n \\ 1 & f = e \\ 0 & e \text{ is a feature and } f \neq e \end{cases}$$

If e is a product and f is a feature of subexpression e_j , then the commonality of f is the product of number of products of all other subexpressions and the commonality of f in expression e_j because e_j itself can be a compound expression. On the other hand, if e is a sum and f is a feature of subexpression e_j , then the commonality of f in e is the same as that in e_j because the relationship between e_j and other subexpressions of e is alternative, which means if we make a choice in e_j , we can't make other choices in e .

As an example, consider $e = a + b(c + d)$ where $e' = b(c + d)$ and $e'' = (c + d)$. We get $\gamma_b(e) = \gamma_b(e') = \gamma_b(b) \times \#(e'') = 2$.

While it is quite simple to deal with choice descriptions without constraints, choice descriptions with constraints are harder to analyze. The constraint complicates the situation by ruling out some products from a choice expression. For example, in $((a + b)(c + d), a \rightarrow d)$ the constraint filters out the expression ac , which would be valid if there were no constraint. An obvious approach is to generate all products then ruling out those that conflict with constraints. In the example, we have four products, and ac is invalid. Although this approach works fine with small choice descriptions, it does not scale very well, because the need for (repeated!) generation of all products is prohibitive in large feature models.

Therefore, we decompose a choice description into a set of choice descriptions that are without constraints and compute selectivity for those choice expressions.

D. Choice Description Partitions and Decomposition

A set of choice descriptions $\{(e_1, c_1), \dots, (e_n, c_n)\}$ is called a *partition* of a choice description (e, c) if $e = e_1 \dots e_n$, $c = \cup_{1 \leq i \leq n} c_i$, and $\forall i, j: i \neq j \implies \Phi(e_i) \cap \Phi(c_j) = \emptyset$.

Choice description partitions allow us to derive a product in a divide-and-conquer manner by first deriving products from all sub-expressions and then merging the results.

Therefore, before processing a choice description derived from a feature model, we split it into smaller components

that are easier to manipulate. Consider, for example, the following choice description D .

$$D = ((a + b)(c + d)(e + f)(g + h), \{a \rightarrow h, d \rightarrow e\})$$

We can partition D into two smaller choice descriptions:

$$D_1 = ((a + b)(g + h), \{a \rightarrow h\})$$

$$D_2 = ((c + d)(e + f), \{d \rightarrow e\})$$

Making decisions in D is equivalent to making decision in both D_1 and D_2 because a plain expression e can be derived from D iff there exist e_1 and e_2 such that they can be derived from D_1 and D_2 , respectively, and e is the same as the product of e_1 and e_2 . This fact is an instance of the following lemma.

Lemma 1: If $\{cd_1, \dots, cd_n\}$ is a partition of D and each s_i is a decision sequence for D_i , $\Phi(D|(s_1 \dots s_n)) = \cup_{i=1}^n \Phi(D_i|s_i)$.

Here is how we can compute choice partitions. Assume $D = (e, c)$ where $e = e_1 \dots e_n$ and $c = \{c_1, \dots, c_n\}$. Then a choice partition for D can be computed as follows. For each choice expression e_i , construct a choice description $D_i = (ce_i, \emptyset)$. Next, for each D_i and each constraint c_j if $\Phi(D_i) \cap \Phi(c_j) \neq \emptyset$, add c_j to D_i . Finally, for each D_i and D_j if $\Phi(cd_i) \cap \Phi(cd_j) \neq \emptyset$, remove these two choice descriptions from the choice partition set and add their union to the set. We continue this iteration until no further simplification is possible. The resulting set is the choice partition set. Here we use $\Phi(D_i)$ for $\Phi(e_i) \cup \Phi(c_i)$ if $D_i = (e_i, c_i)$. The union of two choice descriptions $D_1 = (e_1, c_1)$ and $D_2 = (e_2, c_2)$ is $(e_1 \cdot e_2, c_1 \cup c_2)$.

A partition $\{cd_1, \dots, cd_n\}$ of choice description $D = (e, c)$ is called a *choice decomposition* if it satisfies the following conditions.

- 1) $\forall i: D_i = (e_i, \emptyset)$ where e_i satisfies constraint c .
- 2) $\pi(D) = \cup_{i=1}^n \pi(D_i)$.
- 3) $\forall i, j: i \neq j \implies \pi(cd_i) \cap \pi(cd_j) = \emptyset$.

The first condition ensures that the constraint c is faithfully represented by the collection of choice expressions e_i . The last two conditions ensure that we don't lose any valid products and that we don't count any valid products more than once.

E. Computing Choice Description Decompositions

Working with choice decompositions guarantees that our approach preserves the semantics of a choice description.

The basis of our approach to computing choice decompositions is the observation that the set of products represented by a choice description can be partitioned into sets of products with respect to the constraint. Consider the choice description $(e, \{a \rightarrow b\})$. The constraint $a \rightarrow b$ partitions the products into three sets: the products containing a and b , the products containing a but not b , and the products without a .

The second set is ruled out by the constraint. The first set is obtained by selecting both a and b from the original choice expression, while the third set is obtained by removing a from the original choice expression. Based on this idea, we can derive sets of literals from a constraint and restrict the expression with these sets to obtain its partitions.

Next we show how to derive sets of literals from a constraint, which are to be used by the restriction operation defined in Section II to compute simplified choice expressions.

We need to define two functions tru and fls to extract sets of literals from a constraint that make the constraint true or false, respectively. (Note that $S \bowtie S' = \{s \cup s' \mid s \in S \wedge s' \in S'\}$.)

$$tru(c) = \begin{cases} \{\{f\}\} & c = f \\ \{\{\bar{f}\}\} & c = \neg f \\ tru(c_1) \bowtie tru(c_2) & c = c_1 \wedge c_2 \\ tru(c_1) \cup fls(c_1) \bowtie tru(c_2) & c = c_1 \vee c_2 \\ tru(c_1) \bowtie tru(c_2) \cup fls(c_1) & c = c_1 \rightarrow c_2 \end{cases}$$

$$fls(c) = \begin{cases} \{\{\bar{f}\}\} & c = f \\ \{\{f\}\} & c = \neg f \\ fls(c_1) \cup tru(c_1) \bowtie fls(c_2) & c = c_1 \wedge c_2 \\ fls(c_1) \bowtie fls(c_2) & c = c_1 \vee c_2 \\ tru(c_1) \bowtie fls(c_2) & c = c_1 \rightarrow c_2 \end{cases}$$

Each set of literal in $tru(c)$ describes an alternative way of making c true. Restricting a choice expression with each such set yields all the different choice expressions that are valid instances satisfying the constraint.

A true assignment for an \vee expression is an assignment that makes any sub-formula evaluate to true. However, to simply take the union of all the true assignments for sub-formulas would violate the third property of a choice decomposition. For example, $tru(a \vee b)$ can't be $\{\{a\}, \{b\}\}$, because otherwise expressions obtained by restricting $\{a\}$ and $\{b\}$ will both contain plain expressions including both a and b , and the semantics will no longer be disjoint. Instead, the result should be $\{\{a\}, \{\bar{a}, b\}\}$, and the plain expression containing both a and b will appear in only one expression. Consider, for example, the following choice description.

$$((a+b)(c+d), \{a \vee c\})$$

Restricting $(a+b)(c+d)$ with a and b will produce $a(c+d)$ and $(a+b)c$, respectively. As we can see the product ac can be derived from both expressions. On the other hand, restricting with a and $\bar{a}c$ will give $a(c+d)$ and bc , respectively, whose products are disjoint.

Although $a \rightarrow b$ can be represented by $\neg a \vee b$, we include the rule for dealing with implication since it is the most common form of constraint in feature models. All true assignments for an implication are the combination of all false assignments to its premise and true assignments to

both its premise and conclusion, which is captured in the last case.

With these cases in hand, we can deal with all kinds of constraints in feature models. For instance:

$$tru(a \rightarrow c \vee d \vee f) = \{\{\bar{a}\}, \{a, b\}, \{a, \bar{b}, c\}, \{a, \bar{b}, \bar{c}, d\}\}$$

Obviously, the result is complementary in the sense that restricting with each set will produce expressions whose products are disjoint. For instance, the expressions obtained by restricting with the first and third set differ in whether they contain a .

This property about restriction and the tru function is captured in the following theorem.

Theorem 1: For a given choice description $D = (e, c)$, the choice partition $\{(e|f, \emptyset) \mid f \in tru(c)\}$ is a choice decomposition. \square

With these auxiliary functions, we can now define a function $elim$ to iteratively eliminate the constraints of a choice description.

$$elim(e, \{c\}) = \{e\}$$

$$elim(e, \{c\} \cup c') = elim\left(\bigcup_{e_i \in e} (\cup_{f \in tru(c)} (e_i|f)), c'\right)$$

Except for trivial constraints such as f and $\neg f$, eliminating each constraint in this way will results in at least two expressions if the constraint is satisfiable by the expression. Thus we potentially suffer from exponential blowup. Fortunately, in some cases, we don't need to decompose a choice expression into two, in particular, if the constraint is a tautology. For example, the approach described in this paper, when applied to $((a+b)c, \{a \rightarrow c\})$, produces $\{ac, bc\}$. However, in our prototype implementation, we detect that $a \rightarrow c$ is a tautology on $(a+b)c$ and simply remove the constraint. In fact, this approach dramatically reduces the size of expressions to a manageable size.

Let us take another look at the definition for tru . We find that there are three places where the \bowtie operator occurs, which are potential causes for the exponential size of choice expressions with respect to structure of a constraint and number of constraints. Luckily, the most common forms of constraints in feature models are $a \rightarrow b$ and $a \rightarrow \neg b$, which result in only two expressions.

F. Reducing Computational Overhead

It seems that our approach is computationally very costly. First, to find a feature with maximum selectivity, we need to compute the selectivity of all features in the feature model. Then, depending on whether that feature is picked or dropped, we obtain one of two new feature models in which several features have been removed. To select the next feature, we need to recompute the selectivities of all remaining features since they generally will have changed due to the changes in the feature model.

Thus, if a choice description has n features, we generally seem to be required to compute $O(kn) = O(n^2)$ selectivities with our approach to select a product of k features.

However, after a decision on a feature, say f , has been made, the choice partition can be often partitioned into several choice descriptions because one or more constraints may be removed due to the selection. This will, in fact, be very likely, because highly selective features are typically involved in constraints that cause the co-occurrence of that feature with others.

Suppose, for simplicity, that the removal of f splits a choice description D into two new choice descriptions D_1 and D_2 with n_1 and n_2 features, respectively. After we have recomputed all selectivities, we will find the next feature, say f' , in either D_1 or D_2 (since choice descriptions are linear). Say, f' is contained in D_1 , then we have to recompute selectivity only for the features in D_1 since the commonality of features in D_2 does not change.

We can continue this line of reasoning inductively, which shows that the need to recompute the selectivity of features is needed only for a fraction of the features. Under the assumption that the next selective feature is always in the largest block of the D partition and partitions are always split in half, we get as the number of recomputed selectivities the following.

$$\begin{aligned}
 & n + \\
 & (n-1)/2 + (n-2)/2 + \\
 & (n-3)/4 + (n-4)/4 + (n-5)/4 + (n-6)/4 + \dots + \\
 & \underbrace{(n-\log n)/(\log n) + \dots + (n-\log n)/(\log n)}_{\log n \text{ times}} \\
 & \leq n + (n-1) + \dots + (n-\log n) \\
 & = O(n \log n)
 \end{aligned}$$

This shows that we can expect the effort for recomputing selectivities to be less than $O(n^2)$ and that it decreases along with the selection process. This could be exploited by pre-computing and storing selectivities and for the first few most selective features and the feature models that result from their selection/rejection.

IV. EXPERIMENT AND EVALUATIONS

Next we report the results of empirical evaluations of our method on Berkeley DB [9] and the e-Shop example [8]. The major research question investigated was whether our approach could speed up product derivation. In particular, in how many cases would our approach be applicable and what improvements would it yield? We were also interested in how long the response time is to compute the next feature since the approach is intended to be used in an interactive environment.

Instead of running costly and time consuming user studies, we select a subset of the all products, which was done

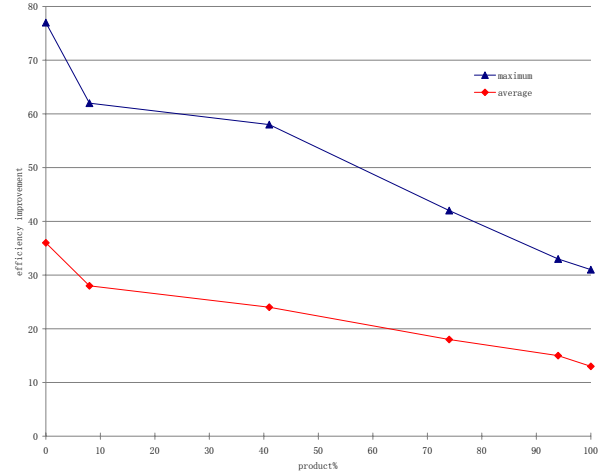


Figure 3. Derivation speed-up for BDB feature model

by randomly selecting or dropping particular features to simulate a hypothetical user. This works as follows. Given a feature model, all the features are ordered in ascending order according to their selectivity. Then a random binary value is generated to simulate the user's decision of whether or not to select the current feature. In either case a new feature model will be produced. We continue this process until we obtain a product.

Along with the obtained product, we also have the information about how many features we looked at and tried. This should be the minimal number of features we have to examine in order to get the product under our technique. Note that we also count the number of features that are looked at but rejected, because the user has spent effort on such features.

In order to calculate the longest decision sequence, we select features in increasing order of selectivity since this will eliminate fewest features along the way.

Having the longest and shortest decision sequences of lengths l and s , respectively, we can compute the maximum speed-up by $(l-s)/l$. However, this number is not very informative since the longest decision sequence reflects the worst case of product derivation and is not a very likely scenario. To obtain numbers for an average-case improvement, we randomly select features until we get the product we want. The number of all features looked at is then taken as the length of the decision sequence. To make this approach reliable, we repeat this process 500 times for each product and then average all these numbers, to yield a , the average number of features a user has to look at when deriving the product. The average efficiency improvement is then defined as $(a-s)/a$.

Figure 3 shows the maximum and average speed-up for deriving products from Berkeley DB [9] by using our technique. In the figure, the x axis denotes the percent

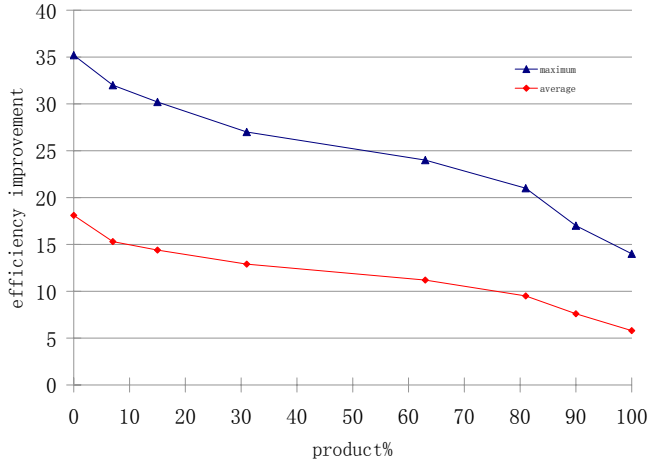


Figure 4. Derivation speed-up for e-Shop feature model

of products and the y axis denotes percent of efficiency improvement. A point (x,y) on the curve means that x percent of all products enjoy a speed-up of at least y percent. For example, the point $(41, 24)$ on the curve indicates that at least 41% percent of all products in the Berkeley DB feature model have an average speed-up of at least 24%. We can observe that the speed-up is significant, which is important taking into account the fact that products have to be derived many times. Specifically, the average speed-up is between the one and one third of the maximum speed-up. Moreover, there is no product for which we obtain a slow-down.

A similar trend can be observed for the decision sequences in the e-Shop example [8], see Figure 4.

Figure 5 presents the relationship between derivation speed-up, ECR (the ratio of number of features involved in extra constraints over number of all features) [12] and the occurrences of each feature involved in constraints. We empirically studied our approach by automatically generating many feature models with different sizes and different ECRs and ran our approach on them. The way we generated our feature models is the same as that described in [13]. We used the same parameters, such as the ratio of AND, OR and XOR features, the number of sub-features of a feature, the ratio of constraints and the size of feature models. In Figure 5, the curve 1avg (2avg) shows the average speed-up for different ECRs where each feature occurs once (twice) if it is involved in constraints, respectively. The curves 1max and 2max represent maximum speed-ups. A point on a curve denotes the speed-up for all products that can be derived from a feature model under corresponding parameters. For example, we have $(0.33, 11)$ on 1max, which means that when the ECR is 0.33, the maximum speed-up for deriving any product is about 11%.

Finally, we found that computing the next feature for selection took at most a few seconds, which is tolerable. In

any case, this can probably be improved considerably. First of all, our prototype was implemented in Haskell without any special consideration for efficiency. Reimplementing the tool in, say C, might lead to an improvement in the running time. Moreover, as we have indicated in Section III-F, there are techniques for pre-computation that can help further reduce the time to compute selectivities, in particular, at the beginning of the process where it is most needed because the choice expressions are still large and we need most selectivities.

V. RELATED WORK

Variability management has been identified as one of the most important parts of successful SPL practices. Number of products (a.k.a. variation degree) [5] and commonality [4], together with some other notions form important measurements for evaluating feature models [14]. Due to their significance, much work has been done on variation degree and commonality [12], [15]–[18]. However, to the best of our knowledge, we are the first to exploit the difference of selectivity among features to guide decision making process to improve the efficiency of application engineering.

The problem of computing the commonality for a feature can be reduced to computing the number of products by first selecting that particular feature, which leads to a new feature model, and then calculating the number of products of that new feature model. Czarnecki et al. [15] transform feature models to propositional formulas and employ a SAT solver to compute the number of products. However, SAT solvers are inefficient in counting satisfying instances, even though Mendonca et al. [13] empirically showed that satisfiability testing for feature models is easy. In [17], [19], Benavides and colleagues transform feature models to constraint programming models and use a constraint solver

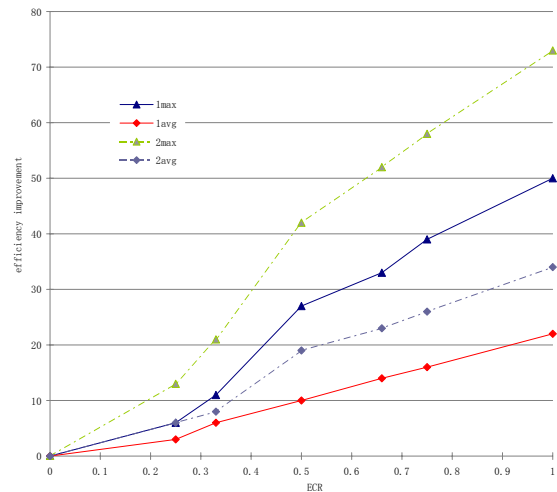


Figure 5. Relationship between derivation speed-up and ECR

to compute variation degree. Constraint solvers suffer from similar efficiency problem as SAT solvers.

Because of these efficiency concerns, we can't build our approach on these models. In [18], Von der Massen and Lichter use a mathematical method to calculate a rough approximation of variation degrees of feature models. However, their approach is relatively coarse and many features have the same selectivity under their method. Using BDDs [20] is the most efficient way for computing the number of products and commonality when compared to other approaches. The time complexity to compute variation degree is $O(|G|)$, where G is the BDD for a feature model and $|G|$ is the number of nodes in the BDD. To compute commonality, we first need to apply feature restriction to the BDD, then again use the method for computing the number of products to compute commonality. Thus, in a feature model with n features whose BDD is G , the complexity for computing commonality is $O(n|G|)$.

A disadvantage of the BDD approach is that the ordering of features has a critical impact on the size of the BDD, and computing the optimal ordering is NP-hard. In [12] Mendonca and colleagues propose two promising heuristics for computing the ordering for building BDD for feature models. By using these heuristics, the size of the BDD for feature models can be significantly reduced when compared to other heuristics. When the ECR is low, our choice algebra approach outperforms the BDD approach, and when the ECR is high, the two approaches have competitive performance. Moreover, when a feature model can be partitioned into clusters, our approach scales better.

It is generally recognized that product derivation is a challenging task due to the dependencies among features and increasingly complicated feature models. To address this issue, many tools and approaches have been proposed. Batory shows that feature models can be represented in propositional formulas [21]. He proposes a SAT-solver-based logic truth maintenance system to ensure that the derived product is free of inconsistencies. Mannion et al. [5] encode feature models in propositional formulas, which are then used to check the correctness of configurations. Benavides et al. [22] and Hadzic et al. [23] develop tools to support feature selection by auto-completion in the sense that when a feature is chosen, the selection of this feature is propagated to choose further features based on extra constraints, but they don't provide suggestions about the order in which features are to be decided.

Moreover, auto-completion does not help with the problem of inconsistencies, which can still occur, for example, in staged configurations [7]. The work done by White and colleagues tackles this problem [24]. Their approach is to transform feature models into constraint satisfaction problems and use a constraint solver to derive the minimal set of feature selection changes to fix errors in an invalid configuration. Incidentally, our approach helps avoiding in-

consistencies into a product by forcing early decisions for these features. Moreover, with the help of our approach, users are less likely to run into a situation in which they are unable to choose a specific feature because the features it requires are eliminated in earlier decisions.

With respect to decision making for product derivation, Mendonca et al. [25] focus on the coordination of decision making. Schmid and colleagues [26] compared different approaches of decision modeling for product derivation. Our work is complementary to those works in the sense that they pay attention to the information recorded during decision making while our work tries to accelerate the decision making process. Decision making for produces derivation based on other models can be found in [27].

VI. CONCLUSIONS AND FUTURE WORK

Application engineering is an important component of the SPL paradigm; it is repeated whenever a new product is derived and built. Thus, improving the efficiency of application engineering is critical to realize all advantages of the SPL paradigm. In this paper, we have introduced a selectivity-directed approach to improve the efficiency of decision making when deriving products. To calculate the selectivity for all features efficiently, we have developed choice algebra, a partitioning technique, and a decomposing technique that removes constraints in choice descriptions.

The evaluation results show that our approach can significantly speed-up the product derivation process without noticeable interactive response delay. The effectiveness of our technique increases with the ECR. Also, our technique helps avoiding feature-selection conflicts.

In future work, we will be developing an approximation methods for computing number of products, commonality, and selectivity. Since a short response time is critical for an interactive system that supports users in the decision-making process, we might not able to afford the computation of exact numbers when feature models are large or are not amenable to decomposition. In fact, for guiding the decision making, knowledge about the relative ordering of feature selectivity is sufficient. Another possibility for improving efficiency is to identify those features that cut across multiple subsystem in the feature model and ask for decisions about them first. After this, the monolithic feature models can be partitioned down. We have already started to investigate using the impact of features as an approximation for their selectivities, and while this works in many cases, it doesn't always provide the most selective features and thus does not always lead to as much efficiency improvements as our current approach. A detailed analysis of the trade-off will remain to be investigated in the future.

Our approach forces users to make high-impact decisions early on, which might lead to mistakes. Moreover, users might prefer to make easy decisions first and postpone hard, high-impact decisions. We plan to study how much of a

problem that is and whether we can employ strategies to entice users to make hard decisions first.

ACKNOWLEDGMENTS

This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

REFERENCES

- [1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 3rd ed. Addison-Wesley Professional, 2001.
- [2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Nak, and A. S. Peterson, "Feature-oriented domain analysis(foda) feasibility study," CMU/SEI, Technical Report CMU/SEI-90-TR-21, 1990.
- [3] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing pla at bosch gasoline systems: Experiences and practices," in *Software Product Lines, Third International Conference, SPLC 2004*, 2004, pp. 34–50.
- [4] P. Trinidad, D. Benavides, and A. Ruiz-corts, "Improving decision making in software product lines product plan management," in *CEUR Workshop Proceedings*, 2004.
- [5] M. Mannion, "Using first-order logic for product line model validation," in *Software Product Lines, Second International Conference, SPLC 2*, 2002, pp. 176–187.
- [6] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multi-level configuration of feature models," *Software Process Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.
- [7] —, "Staged configuration using feature models," in *Software Product Lines: Third International Conference, SPLC 2004*, 2004, pp. 266–283.
- [8] S. Q. Lau, "Domain analysis of e-commerce systems using feature-based model templates," Master's thesis, Dept. of ECE, University of Waterloo, Canada, 2006.
- [9] C. Kästner, "Aspect-oriented refactoring of berkeley db," 2007.
- [10] M. Riebisch, "Towards a more precise definition of feature models," in *Modelling Variability for Object-Oriented Product Lines*, 2003, pp. 64–76.
- [11] P. Höfner, R. Khedri, and B. Möller, "Feature algebra," in *LNCS, FM 2006: Formal Methods*, 2006, pp. 300–315.
- [12] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan, "Efficient compilation techniques for large scale feature models," in *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, 2008, pp. 13–22.
- [13] M. Mendonca, A. Wasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *SPLC '09: Proceedings of the 13th International Software Product Line Conference*, 2009, pp. 231–240.
- [14] D. Benavides, S. Segura, and A. Ruiz-Cortes, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, 2010.
- [15] K. Czarnecki and C. H. P. Kim, "Cardinality-based feature modeling and constraints: A progress report," in *International Workshop on Software Factories*, 2005.
- [16] M. Mannion and J. Camara, "Theorem proving for product line model verification," in *Software Product-Family Engineering*, ser. Lecture Notes in Computer Science, vol. 3014, 2004, pp. 211–224.
- [17] D. Benavides, P. Trinidad, and A. R. Cortés, "Using constraint programming to reason on feature models," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005)*, 2005, pp. 677–682.
- [18] T. von der Massen and H. Lichter, "Determining the variation degree of feature models," in *LNCS, Software Product Lines*, 2005, pp. 82–88.
- [19] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005*, 2005, pp. 491–503.
- [20] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, 1986.
- [21] D. S. Batory, "Feature models, grammars, and propositional formulas," in *Software Product Lines, 9th International Conference, SPLC 2005*, 2005, pp. 7–20.
- [22] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-corts, "Fama: Tooling a framework for the automated analysis of feature models," in *In Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VAMOS, 2007)*, pp. 129–134.
- [23] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Moller, and H. Hulgaard, "Fast backtrack-free product configuration using a precompiled solution space representation," in *PETO Conference, DTU-TRYK*, 2004, pp. 131–138.
- [24] J. White, D. Schmidt, D. B. P. Trinidad, and A. Ruiz-Cortes, "Automated diagnosis of product-line configuration errors in feature models," in *Proceedings of the Software Product Line Conference*, 2008.
- [25] M. Mendonça, T. T. Bartolomei, and D. Cowan, "Decision-making coordination in collaborative product configuration," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08, 2008, pp. 108–113.
- [26] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '11, 2011, pp. 119–126.
- [27] R. Rabiser, P. Grünbacher, and D. Dhungana, "Requirements for product derivation support: Results from a systematic literature review and an expert survey," *Information and Software Technology*, vol. 52, no. 3, pp. 324 – 346, 2010.