Augmenting Measure Sensitivity to Detect Essential, Dispensable and Highly Incompatible Features in Mass Customization

Ruben Heradio\textsuperscript{a}, Hector Perez-Morago\textsuperscript{a}, Mauricio Alférez\textsuperscript{b}, David Fernandez-Amoros\textsuperscript{a}, Germán H. Alférez\textsuperscript{c}

\textsuperscript{a}Dept. of Software Engineering and Computer Systems, Universidad Nacional de Educacion a Distancia, Madrid, Spain
\textsuperscript{b}CEA, LIST, Laboratory of Model-Driven Engineering for Embedded Systems. Point Courrier 174, Gif-sur-Yvette, 91191, France
\textsuperscript{c}Facultad de Ingeniería y Tecnología, Universidad de Monterrey, Mexico

Abstract

Mass customization is the new frontier in business competition for both manufacturing and service industries. To improve customer satisfaction, reduce lead-times and shorten costs, families of similar products are built jointly by combining reusable parts that implement the features demanded by the customers. To guarantee the validity of the products derived from mass customization processes, feature dependencies and incompatibilities are usually specified with a variability model. As market demand grows and evolves, variability models become increasingly complex. In such entangled models it is hard to identify which features are essential, dispensable, highly required by other features, or highly incompatible with the remaining features. This paper exposes the limitations of existing approaches to gather such knowledge and provides efficient algorithms to retrieve that information from variability models.

Keywords:
Mass Customization, Product Platform, Feature Incompatibility, Variability Modeling, Dead Feature, Core Feature, Binary Decision Diagram

Email addresses: rheradio@issi.uned.es (Ruben Heradio), hperez@issi.uned.es (Hector Perez-Morago), mauricio.alferez@cea.fr (Mauricio Alférez), david@lsi.uned.es (David Fernandez-Amoros), harveyalferez@um.edu.mx (Germán H. Alférez)
1. Introduction

Companies have shifted from *mass production* to *mass customization* in order to increase product variety, improve customer satisfaction, reduce lead-times, and shorten costs [1, 2, 3, 4]. For instance, Van der Linden et al. [5] reports successful experiences of large companies such as Bosch (Gasoline Systems), Nokia (Mobile Phones), Philips (Consumer Electronics Software for Televisions), Siemens (Medical Solutions), etc.

Mass customization enriches the mass production economies of scale with the flexibility of custom manufacturing by developing families of related products instead of single products. From this perspective, designing a product family requires developing a generic architecture, named *product platform*, that supports the creation of customized products, named *derivatives*, to satisfy different market niches. Derivatives are specified as combinations of *features* demanded by the customers (e.g., get a car with cruise control, speed limiter, directional stability control, etc) [5, 6].

Product platforms usually offer a high number of features whose combination can produce a large quantity of derivatives [7]. For instance, the BMW 7-Series platform supports $10^{17}$ derivatives [8]. Typically not all feature combinations are valid. There may be feature incompatibilities (e.g., “manual transmissions are not compatible with V8 engines”), feature dependencies (e.g., “sport cars require manual gearbox”), etc. As product platforms grow and evolve, the need for feature variability increases, and managing that variability becomes increasingly difficult [8]. *Variability models* (also known as *configuration models*, *feature models*, etc.) are widely used to support variability management by modeling the dependencies and incompatibilities among features [10, 11].

Operational research methods are currently being used to tackle several problems regarding the automated management of variability models. For instance, they are applied to search which derivative best fulfills the requirements of a given customer in terms of costs and/or utilities [12, 13], to diagnose and re-factor variability models [14] (e.g., by removing redundant constraints), etc. In particular, the goal of this paper falls into the model diagnosis domain.

Product variety creates both challenges and opportunities. Customers prefer broad product variety and, therefore, marketing managers are re-
warded with greater revenue when product platforms are extended [15]. Never-
theless, this may increase costs and reduce profits [8, 16, 2, 17, 18]. In order

to provide guidance on how maximize the profits and minimize the costs as-

sociated to product platforms, this paper describes how to analyze variability

to identify which features are essential, dispensable, highly required

by other features, or highly incompatible with the remaining features. That

is, features in a product platform have usually varying degrees of impor-
tance. Some features may be highly demanded by the market and so most

derivatives should include them. Other features may become dispensable as

the market demand evolves. In addition, there may be features that indi-

directly become of key importance because other essential features need them.

Finally, product platforms may include highly incompatible features whose

presence disable many other features, hindering feature combinability.

Since variability models specify how features can be combined to get the

valid derivatives, it is possible to identify the incompatibility and the relative

importance of the features by directly inspecting the models. Neverthe-

less, existing approaches to carry out such identification have the following limi-

tations:

1. **Measures are rigid**. Measures are needed to account for feature incom-

patibility and relative importance. Although some measures have been pro-

posed [19, 20, 21, 22, 23], their sensitivity is not adjustable and thus

they are often too rigid in practice. For instance, the dead measure is

commonly used to detect if a feature is expendable for a product pla-

tform [21]. Its traditional definition is: “a feature is dead if, due to

its dependencies and incompatibilities with the remaining features, it

cannot be included in any derivative”. Imagine a feature that can only

be included in 1% of the derivatives. The high dispensability of such

feature would go unnoticed for the current definition of dead feature.

2. **Algorithms to compute the measures are inefficient**. Calculating the

measures by hand is unfeasible for all but the most trivial variability

models, so their automated computation is required. The usual way to

perform that computation is to translate the models into Boolean for-

mulas and use off-the-self logic tools, such as SAT solvers [24] or Binary

Decision Diagrams (BDDs) [25], to get the measures. Unfortunately,

current algorithms have poor time performance [25, 26, 27].

To overcome the aforementioned limitations, this paper contributes with:
1. The redefinition, by taking into account a sensitivity parameter, of the existing measures. In particular, the following ones have been extended: dead and core features [21], impact and exclusion sets [23], feature necessity and incompatibility [23]. As we will see, sensitivity is a number between 0 and 1. For instance, if the dead sensitivity is set at 0.01, a feature passes to be considered dead if its reusability is restricted to 1% of the derivatives.

2. Algorithms to efficiently compute the measures from a variability model. The input of our algorithms is the Propositional logic codification of a variability model. Since more complex logics than the Propositional one, which include integer arithmetic, transitive closure, etc., can be reduced to Boolean functions [28] [29], our algorithms are general enough to support most variability model notations. In fact, the paper includes an experimental validation of our algorithms processing models specified in three different notations: the Configit language (http://configit.com/), feature models [30], and decision models [31].

This paper provides empirical evidence of the usefulness of our measure redefinition to detect essential, dispensable and highly incompatible features that go unnoticed using current measures. It also shows that our algorithms not only support the efficient computation of the flexibilized measures, but also have better time performance than existing algorithms when computing the rigid measures.

The remainder of this paper is structured as follows. Section 2 introduces the background required to understand our work, i.e., variability models and current measures. Section 3 summarizes related work to our approach, identifying its limitations. Section 4 describes our approach, defining the flexibilization of the considered measures thanks to the sensitivity parameter, and presenting the algorithms that support the computation of the measures. Section 5 reports the experimental validation of our approach. Finally, Section 6 outlines the conclusions of our work.

2. Background

2.1. A Brief Introduction to Variability Models

Figure 1 represents a variability model written as a directed graph where the nodes represent features and the edges represent constraints related to
dependencies or incompatibilities between features. A dashed-line edge depicts that the two connected features are incompatible, while a solid-line edge from a feature to another represents that the first feature requires the second one.

![Diagram](image)

Figure 1: A small variability model represented as a directed graph. Each node represents a feature and arrows depict inter-feature constraints.

Each derivative is characterized by a selection of features from a variability model. From all possible combinations of 6 features, i.e. \(2^6 = 64\), the set of valid derivatives is reduced to the six ones enumerated in Equation 1 due to the feature interdependencies. For instance, \(\{f_1, f_3, f_4, f_5\}\) is discarded because it violates the constraint \(f_4 \leftrightarrow f_5\).

\[
\text{valid derivatives} = \{\{f_1\}, \{f_1, f_3\}, \{f_1, f_3, f_6\}, \\
\{f_1, f_3, f_5\}, \{f_1, f_3, f_5, f_6\}, \{f_1, f_3, f_4\}\} 
\]  

(1)

### 2.2. Measures to Identify Essential, Dispensable and Highly Incompatible Features

As product platforms grow, variability models become bigger and harder to understand. So, there is a need for an automated mechanism that provides information regarding which role each feature plays according to the variability model. Two complementary approaches are found in the literature to identify the essential, dispensable and highly incompatible features:

1. “Look at the feature distribution in all valid derivatives” \([19, 20, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]\). In particular, features that appear in
all or none the valid derivatives are absolutely essential or dispensable, respectively. In the Software Product Line literature [21], those features are known as core and dead, respectively.

According to Equation 1, $f_1$ is included in all derivatives and so it is a core feature. As $f_2$ is missing in every derivative, it is a dead feature.

2. “Look at how features interact with each other in all valid derivatives” [42, 22, 43, 23, 44]. If a feature is required by many others then it is highly necessary. If a feature excludes many others then it is highly incompatible.

In particular, Boender [23] provides the following measures to quantify feature necessity and incompatibility:

(a) The impact set of a feature $f$ is composed of all the features $f'$ that require $f$ to be enabled whenever they are included in a valid derivative, i.e.,

\[ \text{Impact Set}(f) = \{ f' : f' \Rightarrow f \} \]

(b) The exclusion set of a feature $f$ is composed of all the features that are required to be disabled whenever $f$ is included in a valid derivative, i.e.,

\[ \text{Exclusion Set}(f) = \{ f' : f \Rightarrow \neg f' \} \]

(c) The necessity and incompatibility of a feature $f$ is the cardinal of its impact and exclusion sets, respectively, divided by the total number of features $\#F$, i.e.,

\[ \text{Necessity}(f) = \frac{\#\text{Impact Set}}{\#F}, \quad \text{Incompatibility}(f) = \frac{\#\text{Exclusion Set}}{\#F} \]

For instance, Table 1 summarizes feature co-occurrences in Figure 1. According to Equation 1, all products that include $f_4$, $f_5$, or $f_6$, also include $f_3$. So the impact set of $f_3$ is $\{f_3, f_4, f_5, f_6\}$, and its necessity is $\frac{4}{6}$ (it is required by 4 of the 6 features).

Unfortunately, current measures in both approaches are often too rigid in practice. If the focus is set on the feature reusability distribution throughout the derivatives, a feature is considered as core or dead if it is included in ALL or NONE of the derivatives, respectively. So a feature included in
<table>
<thead>
<tr>
<th>Feature</th>
<th>Impact set</th>
<th>Necessity</th>
<th>Exclusion set</th>
<th>Incompatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>{$f_1, f_3, f_4, f_5, f_6$}</td>
<td>1</td>
<td>{$f_2$}</td>
<td></td>
</tr>
<tr>
<td>$f_2$</td>
<td>$\emptyset$</td>
<td>0</td>
<td>{$f_3, f_4, f_5, f_6$}</td>
<td></td>
</tr>
<tr>
<td>$f_3$</td>
<td>{$f_3, f_4, f_5, f_6$}</td>
<td>1</td>
<td>{$f_2$}</td>
<td></td>
</tr>
<tr>
<td>$f_4$</td>
<td>{$f_4$}</td>
<td>1</td>
<td>{$f_2, f_5, f_6$}</td>
<td></td>
</tr>
<tr>
<td>$f_5$</td>
<td>{$f_5$}</td>
<td>1</td>
<td>{$f_2, f_4$}</td>
<td></td>
</tr>
<tr>
<td>$f_6$</td>
<td>{$f_6$}</td>
<td>1</td>
<td>{$f_2, f_4$}</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Summary of feature co-occurrences in Figure 1

MOST or ALMOST NONE products is overlooked. For instance, according to Equation 1, $f_3$ is not identified as core feature although it appears in all products except one. Similarly, $f_4$ is not considered a dead feature although it is included in just one of the valid derivatives.

In the same way, if the focus is changed to looking at how features interact with each other in all derivatives, a feature $f$ belongs to $f'$ exclusion set if there is NO product including $f$ and $f'$. Figure 2 depicts a variability model where feature $f_1$ is included in 32 valid derivatives. Among them, just one includes $f_7$ and so it might be convenient to consider $f_7$ as part of $f_1$’s exclusion set.

Figure 2: Highly incompatible feature undetected by the current definition of exclusion set and incompatibility

In Section 4, we propose a measure redefinition where ALL and NONE can be flexibilized to MOST OF and ALMOST NONE. Moreover, we provide an algorithm that supports that flexibilization.
3. Related Work

To model the configurable options of a product family, a number of different notations are available. For instance, Feature Diagrams (FD) \[30\], Decision Diagrams \[45\], the Configit language\[1\], the SAP Product Configurator language\[2\], the Oracle Configurator language\[3\], etc. Interestingly, most of those notations are semantically equivalent \[46, 47\]. In fact, instead of processing models directly, automated tools for variability management usually translate them into a propositional logic representation, such as a logic formula in conjunctive normal form, a BDD, etc. That logic representation is then processed using off-the-self tools, such as SAT solvers, BDD engines, etc. \[48\]

For instance, Equation 2 shows the Boolean encoding of Figure 1. The first row means that at least one of the six features has to be selected. The second and third rows encode constraints between those features. The second row encodes five dependencies, for example \(\neg f_2 \lor f_3\) means that \(f_2\) requires \(f_3\). The third row encodes three incompatibilities, for example \(\neg f_1 \lor \neg f_2\) means that \(f_1\) is incompatible with \(f_2\).

\[
\psi = (f_1 \lor f_2 \lor f_3 \lor f_4 \lor f_5 \lor f_6) \land \\
(\neg f_2 \lor f_3) \land (\neg f_3 \lor f_1) \land (\neg f_4 \lor f_3) \land (\neg f_5 \lor f_3) \land \\
(\neg f_1 \lor \neg f_2) \land (\neg f_4 \lor \neg f_5) \land (\neg f_4 \lor \neg f_6)
\] (2)

The usual way to compute the core and dead features of a formula \(\psi\) with \(n\) features is described by Algorithm 1 \[11, 12, 20, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 49, 26\]. The idea is to repeatedly call the \texttt{sat\_count} function, one time for each feature \(f_i\), to get the number of satisfying assignments of \(\psi \land f_i\). The most common approaches to compute \texttt{sat\_count} are:

1. Using a \#SAT solver. A SAT solver is a program that tries to determine if a Boolean formula is satisfiable. A \#SAT model counter is a program that tries to determine how many models (i.e., how many satisfying assignments) a formula has. A simple SAT solver can be easily

\[\text{http://configit.com/}\]
\[\text{https://scn.sap.com/docs/DOC-25224}\]
modified to act as a model counter, and even as an explicit model generator. Nevertheless, as SAT solver techniques grow increasingly specialized, they become useless for these other problems, which demand their own techniques. For instance, it is now customary for solvers to implement timed restarts; if no answer to the SAT problem is found, then the search is interrupted and continued elsewhere. For satisfiable cases, it suffices to find one model, so the technique seems to speed up the process. However, it does not carry over to efficient counting or enumerating of the models. While the SAT problem is known to be NP-complete [50], it is widely believed that the #SAT problem is even harder [51].

2. Using a BDD engine. BDDs are an optimized way of representing Boolean functions by mean of rooted, directed, acyclic graphs [52, 53]. The Achilles heel of BDDs is its size, which depends on the variable ordering the BDD uses to encode $\psi$. Ordering heuristics can take remarkably long, and it is known that finding an ordering that produces an optimal BDD (i.e. a BDD with the minimal number of nodes) is an NP-complete problem [54]. For a BDD with $m$ nodes, computing $\text{sat}\_\text{count}$ has complexity $O(m)$ [52]. So, the time complexity of Algorithm 1 for a model with $n$ features is $O(mn)$.

**Algorithm 1**: get core and dead features (straightforward approach)

```
1  Input Boolean formula $\psi$; set $F$ of all features
2  Output a list with all core features; a list with all dead features
3  var core_features, dead_features: list; count: int;
4 begin
5   core_features = {}
6   dead_features = {}
7   count = sat_count($\psi$)
8   forall $f \in F$ do
9     count' = sat_count($\psi \land f$)
10    if count' == count then
11       core_features.insert($f$)
12     else if count' == 0 then
13       dead_features.insert($f$)
14   return core_features, dead_features
```

The straightforward approach to compute the impact and exclusion sets is described by Algorithm 2. It checks the following two conditions for all pairwise combinations of features $f$ and $f'$:

```
Algorithm 2: `get_impact_and_exclusion_sets` (straightforward approach)

1. **Input** Boolean formula $\psi$; set $F$ of all features
2. **Output** two hash tables: one including the impact sets for all features, and another one including the exclusion sets for all features
3. var `impact_sets`, `exclusion_sets`: hash; count, count': int;
4. begin
5.   `impact_sets` = Hash.new
6.   `exclusion_sets` = Hash.new
7. forall $f \in F$ do
8.   count = sat_count($\psi \land f$)
9.   if count == 0 then
10.      `impact_sets`[$f$] = {}
11.      `exclusion_sets`[$f$] = {f}
12. else
13.      `impact_sets`[$f$] = {f}
14.      `exclusion_sets`[$f$] = {}
15. forall $f' \in F\setminus\{f\}$ do
16.   count' = sat_count($\psi \land f \land f'$)
17.   if count' == 0 then
18.      `exclusion_sets`[$f'$].insert(f')
19.   else if count == count' then
20.      `impact_sets`[$f'$].insert(f)
21. return `impact_sets`, `exclusion_sets`
1. If sat_count(ψ ∧ f) and sat_count(ψ ∧ f ∧ f’) coincide then f belongs to the f’ impact set.

2. If sat_count(ψ ∧ f ∧ f’) is zero then f’ belongs to the f exclusion set.

This algorithm requires calling sat_count n · (n − 1) times. Therefore, if the computation is performed with a BDD the complexity is $O(mn^2)$.

Boender [43, 23, 44] proposes the alternative Algorithm 3 which includes the following shortcut: for each feature f, a satisfying assignment a of ψ ∧ f is computed using the sat_one function (line 9), which has time complexity $O(m)$ [52]. If a feature f’ is true in a, it cannot belong to f exclusion set (i.e., since there is at least one derivative than include both f and f’, they cannot be incompatible). So line 18 avoids the unnecessary computation sat_count(ψ ∧ f ∧ f’) for such f’ (in lines 18 and 23, the symbol \ represents set difference). Analogously, if a feature f’ is false in a, f cannot be part of f impact set, and so line 22 avoids calling sat_count(ψ ∧ f ∧ f’). Although the time complexity for Algorithm 3 is the same that for Algorithm 2 in practice it usually saves steps and thus runs faster.

4. Automated Approach to Detect Essential, Dispensable and Highly Incompatible Features

This section is organised in four subsections. First, Subsection 4.1 redefines the measures introduced in Section 2.2 to increase their sensitivity. Then, Subsection 4.2 describes two algorithms that compute the measures. Those algorithms require the auxiliary computation of feature probability, which is described in Subsection 4.3. Finally, Subsection 4.4 discusses the computational cost of all the algorithms considered in this paper.

4.1. Measure Flexibilization

The sensitivity of existing measures can be augmented using the concept of feature probability, which accounts for the likeliness of a feature to be included in a derivative.

**Definition 1.** The probability of a feature f is calculated as:

\[
\Pr(f) = \frac{\text{Number of valid derivatives that include } f}{\text{Total number of valid derivatives}}
\]
Algorithm 3: get_impact_and_exclusion_sets (Boender’s approach)

Input Boolean formula $\psi$; set $F$ of all features
Output two hash tables: one including the impact sets for all features, and another one including the exclusion sets for all features

var impact_sets, exclusion_sets: hash;
count, count': int; true_features, false_features: list;

begin
  impact_sets = Hash.new
  exclusion_sets = Hash.new
  forall $f \in F$ do
    count = sat_count($\psi \land f$)
    assignment = sat_one($\psi \land f$) // get a satisfying feature assignment
    true_features = get_features_with_true_value(assignment)
    false_features = get_features_with_false_value(assignment)
    if count == 0 then
      exclusion_sets[$f$] = {$f$}
    else
      impact_sets[$f$] = {$f$}
      exclusion_sets[$f$] = {}  
   forall $f' \in F\setminus \{f\}$ do
      count' = sat_count($\psi \land f \land f'$)
      if count' == 0 then
        exclusion_sets[$f$].insert($f'$)
        exclusion_sets[$f'$].insert($f$)
   forall $f' \in F\setminus \{f\}$ do
      count' = sat_count($\psi \land f \land f'$)
      if count' == count then
        impact_sets[$f'$].insert($f$)
  end
return impact_sets, exclusion_sets

<table>
<thead>
<tr>
<th>Feature</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
<th>$f_5$</th>
<th>$f_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Pr(f)$</td>
<td>1</td>
<td>0</td>
<td>$\frac{1}{3}$</td>
<td>$\frac{1}{3}$</td>
<td>$\frac{1}{3}$</td>
<td>$\frac{1}{3}$</td>
</tr>
</tbody>
</table>

Table 2: Feature probabilities for Figure 1
Table 2 summarizes feature probabilities for Figure 1. For instance, looking at Equation 1 it can be checked that feature $f_3$ is included in 5 of the 6 valid derivatives, so its probability is $\frac{5}{6}$.

Let sensitivity be a number between 0 and 1. The measures this paper deals with can be redefined as follows.

**Definition 2.** A feature $f$ is core or dead under sensitivity $\alpha$ if Equation 3 or 4 holds, respectively.

$$
\Pr(f) \geq (1 - \alpha)
$$

$$
\Pr(f) \leq \alpha
$$

For example, under sensitivity 0.2, $f_3$ is considered core (i.e., $\frac{5}{6} \geq 1 - 0.2$), and $f_4$ dead (i.e., $\frac{1}{6} \leq 0.2$).

**Definition 3.** The probability $\Pr(f|f')$ of a feature $f$ conditioned to another feature $f'$ is calculated as:

$$
\Pr(f) = \frac{\text{Number of those derivatives that including } f' \text{ also include } f}{\text{Number of derivatives that include } f'}
$$

For instance, from all the derivatives summarized by Equation 1, only the following two include $f_5$: $\{f_1, f_3, f_5\}, \{f_1, f_3, f_5, f_6\}$. As just one of them includes $f_6$, $\Pr(f_6|f_5) = \frac{1}{2}$.

**Definition 4.** The impact set of a feature $f$ under sensitivity $\alpha$ is composed of all the features $f'$ that require $f$ to be enabled whenever they are included in at least $1 - \alpha$ of the derivatives, i.e,

$$
\text{Impact Set}_\alpha(f) = \{f' \cdot \Pr(f'|f) \geq (1 - \alpha)\}
$$

Table 3 summarizes the conditional probabilities for all features in Figure 3. Let us suppose sensitivity is 0.2, then the impact set of $f_3$ is $\{f_1, f_3, f_4, f_5, f_6\}$ (i.e., those features whose rows are $\geq 0.8$ in column $f_3$).

**Definition 5.** The exclusion set of a feature $f$ under sensitivity $\alpha$ is composed of all the features $f'$ that are required to be disabled in at least $1 - \alpha$ of the derivatives that include $f$, i.e,

$$
\text{Exclusion Set}_\alpha(f) = \{f' \cdot \Pr(f'|f) \leq \alpha\}$$
Table 3: Feature conditional probabilities for Figure 1

For example, according to Table 3, the exclusion set of $f_3$ with sensitivity 0.2 is \{${f_2, f_4}$\} (i.e., those features whose columns are $\leq 0.2$ in row $f_3$).

Note that Definitions 2, 4, and 5 coincide with the rigid ones given in Section 2.2 for the extreme case when sensitivity is 0.

4.2. Algorithms to Detect Core & Dead Features, and to Compute Impact & Exclusion Feature Sets

The following Subsection 4.3 presents the algorithm get_prob that computes the feature probabilities of a variability model encoded as a BDD. Using that algorithm:

1. The identification of all core and dead features in a variability model considering a given sensitivity is performed by Algorithm 4. To do so, the algorithm builds the BDD that encodes the formula $\psi$ according to an input variable ordering. Remember from Section 3 that $\psi$ is the Boolean representation of the input variability model. The variable ordering is specified using an array which sets how variables are arranged in the BDD (see Section 4.3.2). Then, feature probabilities are computed by calling Algorithm 6 using the BDD as input parameter (line 8). Finally, Definitions 3 and 4 are applied to check if the features are core or dead according to their probability (lines 11-14).

2. The impact and exclusion sets of all features in a variability model considering a given sensitivity is computed by Algorithm 5. To do so, for each feature $f$, the algorithm builds a BDD that encodes the formula $\psi \land f$ (line 11). Afterwards, feature probabilities are computed by calling Algorithm 6 (line 12), and those probabilities are used to compute the impact and exclusion sets according to Definitions 4 and 5 (lines 15-18).
Algorithm 4: get_core_and_dead_features (flexible approach)

1 Input Boolean formula $\psi$; var_ordering: array[0..n-1] of string; sensitivity $\in [0,1]$
2 Output a list with all core features; a list with all dead features
3 var core_features, dead_features: list; i: int;
4 begin
5     core_features = {}
6     dead_features = {}
7     bdd = get_bdd($\psi$, var_ordering)
8     feature_probabilities = get_prob(bdd, var_ordering)
9     i = 0
10    while $i < \text{length(var_ordering)}$ do
11       if feature_probabilities[$i$] $\geq (1 - \text{sensitivity})$ then
12          core_features.insert(var_ordering[$i$])
13       else if feature_probabilities[$i$] $\leq \text{sensitivity}$ then
14          dead_features.insert(var_ordering[$i$])
15       i += 1
16    return core_features, dead_features

Algorithm 5: get_impact_and_exclusion_sets (flexible approach)

1 Input Boolean formula $\psi$; var_ordering: array[0..n-1] of string; sensitivity $\in [0,1]$
2 Output two hash tables: one including the impact sets for all features, and another one
3 including the exclusion sets for all features
4 var impact_sets, exclusion_sets: hash; i, j: int;
5 begin
6     impact_sets = Hash.new
7     exclusion_sets = Hash.new
8     i = 0
9     while $i < \text{length(var_ordering)}$ do
10        impact_sets[var_ordering[$i$]] = {}
11        exclusion_sets[var_ordering[$i$]] = {}
12        bdd = get_bdd($\psi \wedge \text{var_ordering}[i]$, var_ordering)
13        feature_probabilities = get_prob(bdd, var_ordering)
14        j = 0
15        while $j < \text{length(var_ordering)}$ do
16           if feature_probabilities[$j$] $\geq (1 - \text{sensitivity})$ then
17              impact_sets[var_ordering[$j$]].insert(var_ordering[$i$])
18           else if feature_probabilities[$j$] $\leq \text{sensitivity}$ then
19              exclusion_sets[var_ordering[$j$]].insert(var_ordering[$i$])
20           j += 1
21        i += 1
22    return impact_sets, exclusion_sets
4.3. A BDD Algorithm to Compute Feature Probabilities

BDDs are a way of representing Boolean functions. They are rooted, directed, acyclic graphs, which consist of several decision nodes and terminal nodes [52]. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node \( v_i \) is labeled by a Boolean variable \( x_k \) and has two child nodes called low and high (which are usually depicted by dashed and solid lines, respectively). The edge from node \( v_i \) to a low (or high) child represents an assignment of \( v_i \) to 0 (resp. 1). Such a BDD is called ordered if different variables appear in the same order on all paths from the root. A BDD is said to be reduced if the following two rules have been applied to its graph: (i) isomorphic subgraphs are merged, and (ii) nodes whose two children are isomorphic are eliminated.

Let us use formula \( \psi \equiv (x_1 \land x_2) \lor (x_3 \land x_4) \) as a running example for this subsection. Table 4 is the truth table for \( \psi \). Figure 3 is its BDD representation using the variable ordering \( x_1 \prec x_2 \prec x_3 \prec x_4 \).

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( \psi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Truth table for \( \psi \equiv (x_1 \land x_2) \lor (x_3 \land x_4) \)

The remainder of this subsection is structured as follows. First, some definitions required to understand our algorithm are given. Next, the data

---

4in popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram [28]. In this paper, we will follow that convention as well.

5note that a logic formula may be encoded with different BDDs according to the variable ordering used to synthesize the BDD. Obviously, our algorithm produces the same results for equivalent BDDs (i.e., BDDs that encode the same formula).

---

16
structures the algorithm uses are described from a theoretical perspective. Then, the algorithm is presented. Finally, the algorithm computational cost is discussed.

4.3.1. Definitions

Definition 6. The satisfying set of a Boolean formula $\psi(x_1, ..., x_n)$, denoted $S_\psi$, is defined by Equation 5.

$$S_\psi = \{(x_1, ..., x_n) : \psi(x_1, ..., x_n) = \text{true}\}$$

(5)

Definition 7. The satisfying set of the variable $x_i$ of a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $S_\psi|_{x_i=\text{true}}$, is defined by Equation 6.

$$S_\psi|_{x_i=\text{true}} = \{(x_1, ..., x_{i-1}, \text{true}, x_{i+1}, ..., x_n) : \psi(x_1, ..., x_{i-1}, \text{true}, x_{i+1}, ..., x_n) = \text{true}\}$$

(6)

For instance, according to Table 4, $\#S_\psi = 7$ since there are 7 rows where $\psi$ evaluates to true, and $\#S_\psi|_{x_4=\text{true}} = 5$ because $x_4 = 1$ in 5 of the 7 rows where $\psi = 1$.

Definition 8. The satisfying probability of a Boolean formula $\psi(x_1, ..., x_n)$, denoted $Pr(\psi)$, is defined by Equation 7.

$$Pr(\psi) = \frac{\#S_\psi}{2^n}$$

(7)

Definition 9. The satisfying marginal probability of a variable $x_i$ in a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $MPr(\psi|_{x_i=\text{true}})$, is defined by Equation 8.

$$MPr(\psi|_{x_i=\text{true}}) = \frac{\#S_\psi|_{x_i=\text{true}}}{2^n}$$

(8)

Definition 10. The satisfying probability of a variable $x_i$ in a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $Pr(\psi|_{x_i=\text{true}})$, is defined by Equation 9.

---

"throughout this paper 0/1 and false/true are used interchangeably"
Pr(ψ|\(x_i\)=true) = \#S_{ψ|\(x_i\)=true} / \#S_{ψ} \hspace{1cm} (9)

For instance, looking at Table 4, it is easy to see that Pr(ψ) = \(\frac{7}{24}\), MPr(ψ|\(x_4\)=true) = \(\frac{5}{24}\), and Pr(ψ|\(x_4\)=true) = \(\frac{5}{7}\).

For convenience, in the remainder of the paper we denote Pr(ψ|\(x_i\)=true) and MPr(ψ|\(x_i\)=true) as Pr(\(x_i\)) and MPr(\(x_i\)), respectively, which is consistent with the notation introduced in Definition 4 (i.e., if a variability model is encoded by ψ then: \#S_{ψ|f_i=\text{true}}) represents the number of derivatives that include \(f_i\), \#S_{ψ} is the total number of derivatives, and consequently, Pr(ψ|\(f_i\)=true) = Pr(\(f_i\)).

Figure 3: BDD for ψ according to the variable ordering \(x_1 < x_2 < x_3 < x_4\). Nodes are labeled with the variables they encode. Edges to low children (dashed lines) or high children (solid lines) represent variable assignments to false or true, respectively.

4.3.2. Data Structures

Let us represent a BDD that has \(m\) nodes and encodes a Boolean formula with \(n\) variables by using the following data structures:

- The variable ordering used to synthesize the BDD is represented by an array declared as follows:

```
var_ordering: array[0..n-1] of string
```
• Each node is represented by a record declared as follows:

```pascal
type node = record
  index: 0..n
  low, high: node
  mark: Boolean
end
```

Where:
1. `index` is the index of the variables in the ordering. The terminal nodes of the BDD (i.e., 0 and 1) have index `n`.
2. `low` and `high` are the low and high node successors
3. `mark` is used to mark which nodes have been visited during a traversal of the graph. As we will see, our algorithm is called at the top level with the root node as argument and with the mark fields of the nodes being either all true or all false. It then systematically visits every node in the graph by recursively visiting the subgraphs rooted by the two children `low` and `high`. As it visits a node, it complements the value of the `mark` field, so that it can later determine whether a child has already been visited by comparing the two marks.

• The BDD is represented by an array declared as follows:

```pascal
bdd: array[0..m] of node
```

The terminal nodes of the BDD, 0 and 1, are stored at positions 0 and 1 of the `bdd` array, respectively.

For instance, Tables 5 and 6 represent the content of `bdd` and `var_ordering` for the BDD in Figure 3, respectively.

4.3.3. Algorithm to Compute Feature Probabilities

`Pr(x_i)` is computed jointly by Algorithms 6, 7, and 8. Figure 4 summarizes the computations for the BDD in Figure 3. Let us examine how our approach proceeds:

**Algorithm** 6 computes `Pr(x_i)` as

\[
Pr(x_i) = \frac{\mathbb{E}[Pr(x_i)]}{Pr(\psi)}
\]

by calling the auxiliary Algorithms 7 and 8.
Table 5: Content of the \textit{bdd} array for Figure

\begin{tabular}{|l|l|l|l|l|}
\hline
\textbf{position} & \textbf{index} & \textbf{low} & \textbf{high} & \textbf{mark} \\
\hline
0 & 4 & nil & nil & false \\
1 & 4 & nil & nil & false \\
2 & 3 & 0 & 1 & false \\
3 & 2 & 0 & 2 & false \\
4 & 1 & 3 & 1 & false \\
5 & 0 & 3 & 4 & false \\
\hline
\end{tabular}

Table 6: Content of the \textit{var\_ordering} array for the Figure

\begin{tabular}{|l|l|}
\hline
\textbf{position} & \textbf{content} \\
\hline
0 & \textquotedbl\textit{x}\textsubscript{1}\textquotedbl \\
1 & \textquotedbl\textit{x}\textsubscript{2}\textquotedbl \\
2 & \textquotedbl\textit{x}\textsubscript{3}\textquotedbl \\
3 & \textquotedbl\textit{x}\textsubscript{4}\textquotedbl \\
\hline
\end{tabular}

Algorithm 6: \texttt{get\_prob}

\begin{verbatim}
1 Input \texttt{bdd} and \texttt{var\_ordering} arrays
2 Output an array which stores Pr(\textit{x}\textsubscript{i}) in position \textit{i}
3 var \texttt{formula_sat\_prob}, \texttt{total\_prob}; \texttt{array[0..length(bdd)-1]} of float;
4 \texttt{prob}; \texttt{array[0..length(var\_ordering)/1]} of float; \texttt{i}; \texttt{int};
5 begin
6 \texttt{formula\_sat\_prob} = \texttt{get\_formula\_sat\_prob}(\texttt{bdd})
7 \texttt{get\_marginal\_prob(length(bdd)-1, total\_prob, formula\_sat\_prob, prob, bdd, var\_ordering)}
8 for \texttt{(i=0; i < length(var\_ordering); i++)} do
9 \texttt{prob}[i] = \frac{\texttt{prob}[i]}{\texttt{formula\_sat\_prob}(\texttt{x}\textsubscript{i})}
10 return \texttt{prob}
\end{verbatim}
Algorithm 7 computes $\text{Pr}(\psi)$. A nice mental picture to understand Algorithm 7 is thinking in pouring 1 liter of water from the BDD root to the terminal nodes. 1 liter goes through the root, then half a liter goes through the low branch and half a liter through the high branch. This procedure advances until the water reaches the leaves. Hence, $\text{Pr}(\psi)$ is the amount of water that node 1 has.

In Figure 3, through node $v_5$ goes 1 liter (i.e., $\text{formula.sat.prob}[5] = 1$). Half of it goes to $v_3$ and the other half to $v_4$. Whereas through $v_4$ passes $\frac{1}{2}$ liter, through $v_3$ goes the $\frac{1}{2}$ liter that comes from $v_5$ and half of the water that comes from $v_4$ (i.e., $\text{formula.sat.prob}[3] = \frac{1}{2} + \frac{1}{2} = \frac{3}{4}$).

Algorithm 7: \text{get$\_formula$sat$\_prob$}

1. \textbf{Input} bdd array
2. \textbf{Output} an array which in position 1 stores $\text{Pr}(\psi)$
3. \textbf{var} formula\_sat\_prob: array[0..\text{length(bdd)}-1] of float; i: int
4. begin
5. \hspace{1em} for \((i=0; i < \text{length(bdd)}-1; i++)\) do
6. \hspace{2em} formula\_sat\_prob[i] = 0.0 \text{ // non-root nodes prob is initialized to 0}
7. \hspace{2em} formula\_sat\_prob[0] = 1.0 \text{ // root node prob is 1}
8. \hspace{1em} i = \text{length(bdd)}-1
9. \hspace{1em} while \(i > 1\) do \text{ // for all non-terminal nodes}
10. \hspace{2em} formula\_sat\_prob[bdd[i].low] += \frac{\text{formula\_sat\_prob[i]}}{2.0}
11. \hspace{2em} formula\_sat\_prob[bdd[i].high] += \frac{\text{formula\_sat\_prob[i]}}{2.0}
12. \hspace{1em} i -= 1
13. \textbf{return} formula\_sat\_prob

Algorithm 8 computes MPr($x_1$). In particular, let us examine how it computes MPr($x_2$). In the truth Table 4, $\psi$ evaluates to true when $x_2$ is true five times:

1. In four of them $x_1$ is true. When the call \text{get\_marginal\_prob}(4, ...) is made, lines 10-23 compute the marginal probability of $x_2$ for the explicit path $v_5 \rightarrow v_4$. The probabilities due to the low and high branches of $v_i$ are stored into the prob\_low and prob\_high variables, respectively. As $\text{bdd}[4].\text{low} \neq 0$, a recursive call is made to compute the total probability due to the low descendants of $v_4$.

\footnote{according to Tables 5 and 6 the root node has label $v_5$ and it is in the position 5 of the bdd array}
(i.e., \texttt{get\_marginal\_prob(3, ...)}). As a result:

\[
\text{total \_ prob}[3] = \text{prob\_low}_{v_3} + \text{prob\_high}_{v_3} = 0 + \frac{3}{16} = \frac{3}{16}
\]

Algorithm 8: \texttt{get\_marginal\_prob}

Input: \(v: 0..\text{length}(\text{bdd})-1; \text{total\_prob}, \text{formula\_sat\_prob}: \text{array}[0..\text{length}(\text{bdd})-1] \text{ of float}; \)

Output: \(\text{prob\_low}, \text{prob\_high}: \text{float}; i: \text{int}\)

\begin{verbatim}
prob\_low = 0.0
prob\_high = 0.0

// explicit path recursive traversal
if \(\text{bdd}[v].\text{low} == 1\) then
    \(\text{prob\_low} = \text{formula\_sat\_prob}[v]\)
else if \(\text{bdd}[v].\text{low} \neq 0\) then
    if \(\text{bdd}[v].\text{mark} \neq \text{bdd}[\text{bdd}[v].\text{low}].\text{mark}\) then
        \text{get\_marginal\_prob(bdd[v].low, total\_prob, formula\_sat\_prob, prob, bdd, var\_ordering)}
    \(\text{prob\_low} = \text{total\_prob}[\text{bdd}[v].\text{low}] \cdot \text{formula\_sat\_prob}[v]\)
else if \(\text{bdd}[v].\text{high} == 1\) then
    \(\text{prob\_high} = \text{formula\_sat\_prob}[v]\)
else if \(\text{bdd}[v].\text{high} \neq 0\) then
    if \(\text{bdd}[v].\text{mark} \neq \text{bdd}[\text{bdd}[v].\text{high}].\text{mark}\) then
        \text{get\_marginal\_prob(bdd[v].high, total\_prob, formula\_sat\_prob, prob, bdd, var\_ordering)}
    \(\text{prob\_high} = \text{total\_prob}[\text{bdd}[v].\text{high}] \cdot \text{formula\_sat\_prob}[v]\)

\text{total\_prob}[v] = \text{prob\_low} + \text{prob\_high}
\text{prob}[\text{bdd}[v].\text{index}] += \text{prob\_high}

// implicit path iterative traversal
\end{verbatim}

Notice that \(\text{prob\_low}_{v_2}\) is not simply equal to \(\text{total\_prob}[3]\), because \(\text{total\_prob}[3]\) depends also on the probability that comes from the link \(v_5 \rightarrow v_3\). To get just the probability due to the link
\(v_4 \rightarrow v_3\), \texttt{prob\_low} has to be adjusted using the \texttt{formula\_sat\_prob} array as:

\[
\texttt{prob\_low} = \frac{\text{total}\_\texttt{prob}[3]}{\text{formula\_sat\_prob}[3]} = \frac{3}{16} \cdot \frac{1}{2} = \frac{3}{32} = \frac{1}{16}
\]

Since \texttt{bdd[4].high} = 1, \texttt{prob\_high} is directly computed as:

\[
\texttt{prob\_high} = \frac{\text{formula\_sat\_prob}[4]}{2} = \frac{1}{2} = \frac{1}{4}
\]

Finally:

\[
\texttt{prob[bdd[4].index]} = \texttt{prob\_high} = \frac{1}{4}
\]

2. In one of them \(x_1\) is false. The two following implicit paths that have been removed from the reduced BDD: (i) \(v_5 \rightarrow v_4 \rightarrow v_3\), and (ii) \(v_5 \rightarrow v_4 \rightarrow v_3\). Nevertheless, path \(v_5 \rightarrow v_4 \rightarrow v_3\) should be considered to compute the marginal probability of \(x_2\). Lines 24-31 account for that kind of implicit paths, adjusting the marginal probability with the variables omitted in the paths. For instance, when the algorithm is called for \(v_5\), the marginal probability of \(x_2\) is updated with half the \texttt{prob\_low} of \(v_5\).

To sum up:

\[
\text{MPPr}(x_2) = \text{MPPr}(v_5 \rightarrow v_4 \rightarrow v_3) + \text{MPPr}(v_5 \rightarrow v_4) = \frac{1}{2} \cdot \frac{1}{2} + \texttt{prob[bdd[4].index]} = \frac{3}{4} + \frac{1}{4} = \frac{5}{16}
\]

4.4. Computational cost

Let \(m\) be the number of nodes of the BDD, and \(n\) the number of variables of the Boolean formula. Algorithm 7 requires traverse all the nodes, so its computational complexity is \(O(m)\). Algorithm 8 also traverses all the BDD nodes. In addition, to account for the implicit paths removed from the
Figure 4: Probability computation for BDD 3
reduced BDD, the variables omitted on the edges that come from each node need to be traversed (which is done by lines 24-31).

Table 7 summarizes those traversals for Figure 3. For instance, when \( v_4 \) is recursively traversed, the variables \( x_3 \) and \( x_4 \) need to be iteratively traversed because the edge \( v_4 \to 1 \) omits them (i.e., the variable encoded by node \( v_4 \), \( x_2 \), jumps directly to 1 omitting the intermediate variables \( x_3 \) and \( x_4 \) in the ordering \( x_1 < x_2 < x_3 < x_4 \)). Table 7 helps noticing the savings our algorithm provides compared to the straightforward approach (Algorithms 1 and 2), which requires traversing all nodes for all variables (which in computational cost terms is equivalent to traversing all variables for every node).

Therefore, Algorithm 8 does not traverse \( mn \) elements, but \( mn' \), where \( n' \) is strictly less than \( n \). If \( n' = n \), all nodes in the BDD should go directly to 0 or 1, jumping over all the variables. Nevertheless, as BDDs are organized in hierarchical levels according to the variable ordering, this is impossible (i.e., the nodes that encode a variable with position \( k \) in the ordering only can jump over the variables with positions \( k+1 \ldots n \)).

It follows that Algorithm 6 has computational complexity \( O(mn') \), and thus, Algorithms 4 and 5 have complexity \( O(mn') \) and \( O(mn'n) \), respectively.

Table 8 summarizes the complexities for Algorithms 1, 2, 3, 4, and 5. Our approach has the best time complexity. Nevertheless, computational complexity \( O \) only provides an upper bound on the worst time required by the algorithms. As it will be empirically shown in Section 5 in practice our approach is also the fastest for all the evaluated cases. In particular:

1. Algorithms 1 and 2 run always on the worst case, i.e., they require traversing \( mn \) and \( mn^2 \) for all variability models, respectively.
2. Thanks to its lines 18 and 22, Algorithm 3 saves a number of iterations, requiring in practice \( mn'' \) steps where \( n'' \) is usually \( < n \).

3. Algorithm 5 requires \( mn' \) iterations and, on average requires less time than Algorithm 3, i.e., \( n' \ll n'' \).

<table>
<thead>
<tr>
<th>Core &amp; dead features</th>
<th>Impact &amp; exclusion sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straightforward apch.</td>
<td>Flexible apch.</td>
</tr>
<tr>
<td>(Algorithm 1)</td>
<td>(Algorithm 4)</td>
</tr>
<tr>
<td>( O(mn) )</td>
<td>( O(mn') ) where ( n' &lt; n )</td>
</tr>
<tr>
<td>Straightforward apch.</td>
<td>Boender’s apch.</td>
</tr>
<tr>
<td>(Algorithm 2)</td>
<td>(Algorithm 3)</td>
</tr>
<tr>
<td>( O(mn^2) )</td>
<td>( O(mn'/n) ) where ( n' &lt; n )</td>
</tr>
</tbody>
</table>

Table 8: Computational complexity for Algorithms 1, 2, 3, 4, and 5

To sum up, our approach not only provides new functionality (i.e., it can take into account different levels of sensitivity) but also it is more time-efficient than related work.

5. Experimental Evaluation

The goal of this section is to experimentally check if:

- Our approach is faster than others. In addition to the theoretical evaluation that computational complexity provides (see Section 4.4), it would be desirable to test if in practice our approach has better time performance on real variability models.

- The measure flexibilization we propose really helps to detect essential, dispensable and highly incompatible features that would have passed unnoticed using current rigid measures.

5.1. Time Performance

The time performance of our approach and related work have been evaluated on a benchmark composed the following models:

1. A configuration model provided by the car manufacturing company Renault DVI, which deals with the configuration of a family of cars named Renault Megane. The model is written in the Configit language and can be downloaded from:

   [http://www.itu.dk/research/cla/externals/clib/](http://www.itu.dk/research/cla/externals/clib/)
2. A configuration model for laptops, which was reverse engineered from the DELL homepage on February 2009 by Nöhrer [55, 56, 57]. The model is specified as a decision model and can be downloaded from the C2O website:

http://www.sea.jku.at/tools/c2o

3. All feature models from the SPLOT repository including more than 100 features: Xtext, Battle of Tanks, FM Test, Printers, Banking Software, Electronic Shopping, and a Model for Decision-making for Investments on Enterprise Information Systems. Those models can be downloaded from the SPLOT website:

http://www.splot-research.org/

To support the reliable comparison of the algorithms described in this paper, all of them have been implemented extending the BuDDy package for BDDs, which is freely available at:

http://sourceforge.net/projects/buddy/

Table 9 summarizes the results of the performance tests, which were conducted on an Intel® Core™ 2 i3-4010U with 1.7 GHz and 4GB RAM (although only one core was used). Our approach (Algorithms 4 and 5 parameterized with sensitivity = 0) outperforms related work (Algorithms 1, 2, and 3) in all the experiments.

<table>
<thead>
<tr>
<th>Variability model</th>
<th>Number of features</th>
<th>Dead &amp; core features</th>
<th>Impact &amp; exclusion sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alg. 1</td>
<td>Alg. 4</td>
<td>Alg. 2</td>
</tr>
<tr>
<td>Renault</td>
<td>398</td>
<td>5.07081</td>
<td>3.80750</td>
</tr>
<tr>
<td>Dell</td>
<td>123</td>
<td>0.04938</td>
<td>0.03310</td>
</tr>
<tr>
<td>Xtext</td>
<td>137</td>
<td>0.05547</td>
<td>0.00185</td>
</tr>
<tr>
<td>Battle of Tanks</td>
<td>144</td>
<td>0.11169</td>
<td>0.00200</td>
</tr>
<tr>
<td>FM Test</td>
<td>168</td>
<td>0.12117</td>
<td>0.04905</td>
</tr>
<tr>
<td>Printers</td>
<td>172</td>
<td>0.12962</td>
<td>0.00271</td>
</tr>
<tr>
<td>Banking Software</td>
<td>176</td>
<td>0.15372</td>
<td>0.00397</td>
</tr>
<tr>
<td>Electronic Shopping</td>
<td>290</td>
<td>1.18944</td>
<td>0.37749</td>
</tr>
<tr>
<td>Investments on Enterprise Information Systems</td>
<td>366</td>
<td>17.80436</td>
<td>4.90573</td>
</tr>
</tbody>
</table>

Table 9: Performance in seconds of Algorithms 1, 2, 4, 5 and 6.
It is worth noting that Boender’s shortcut to compute the impact and exclusion sets (Algorithm 3) is not always faster than the straightforward approach (Algorithm 2). Thus, in some cases the extra cost required due to the sat_one call in line 9 of Algorithm 3 does not payoff. In particular, it is slightly slower for the following SPLOT test cases: Xtext, FM Test, Printers, and Banking Software.

5.2. Benefits of Measure Sensitivity

The Renault Megane model is a benchmark widely used by the mass customization community (e.g., [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68]). To the extent of our knowledge no author has reported dispensable nor highly incompatible features for the model. Nevertheless, the model includes 6 dead features, i.e., the 1.51% of the features cannot be included in any derivative.

If sensitivity is taken into account and so feature probabilities are computed, the perception of feature reusability changes dramatically. Figure 5 shows the histogram of feature probabilities. According to such figure, 53.27% of the features are dead at sensitivity 0.05 (see the first bar). In other words, more than half of the features can be reused at most in just 5% of the valid derivatives!

![Histogram of feature probabilities for the Renault Megane example](image_url)

Figure 5: Histogram of feature probabilities for the Renault Megane example

Figure 6 shows how the number of incompatible feature grows drastically with a small sensitivity increase. According to the rigid version of feature
incompatibility, there are 33 features incompatible with at least 50% of the remaining ones (8.29% of the features). When sensitivity is set to 0.5, that number becomes 380 (95.48% of the features).

Figure 6: Histogram of feature incompatibilities for the Renault Megane example

Table 10 summarizes the aforementioned outcomes, highlighting the benefits of using our flexible approach.

<table>
<thead>
<tr>
<th></th>
<th>Rigid Measures (sensitivity=0)</th>
<th>Flexible Measures (sensitivity=0.05)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dead features</td>
<td>1.31%</td>
<td>53.27%</td>
</tr>
<tr>
<td>Core features</td>
<td>1.76%</td>
<td>10.55%</td>
</tr>
<tr>
<td>Features incompatible</td>
<td>8.29%</td>
<td>95.48%</td>
</tr>
<tr>
<td>with more than 50%</td>
<td></td>
<td>Features incompatible in 95% of</td>
</tr>
<tr>
<td>of the remaining</td>
<td></td>
<td>the valid products with more</td>
</tr>
<tr>
<td>features</td>
<td></td>
<td>than 50% of the remaining features</td>
</tr>
</tbody>
</table>

Table 10: Comparison of rigid versus flexible outcomes for the Renault Megane example
6. Conclusions

The development and maintenance of product platforms require the management of complex variability models. The role each feature plays in such models is unclear to the naked eye and so automated support is needed to identify which features are essential, dispensable, highly required by other features and highly incompatible with the remaining features. We have exposed the drawbacks of existing approaches to provide that support.

Due to the rigidness of the current measures to account for feature inter-relations, relevant information is frequently overlooked. Moreover, existing approaches to automatically compute the measures from variability models have poor time performance.

To overcome such problems, we have increased measure sensitivity by introducing the concepts of feature probability and conditional probability. It has been empirically shown that our measure flexibilization unveils critical information that current measures cannot detect.

Finally, we have provided new algorithms to support the computation of these measures. It has been shown, both theoretically and experimentally, that our algorithms not only can take into account different levels of sensitivity, but also are more time-efficient than related work even for computing the rigid measures.

Acknowledgement

This work has been supported by the Spanish Government under the CICYT project DPI-2013-44776-R, and the Comunidad de Madrid under the RoboCity2030-II excellence research network S2009DPI-1559.

References


