European Journal of Operational Research 000 (2015) 1–12

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Decision Support

Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization

Ruben Heradio [a,*], Hector Perez-Morago [a], Mauricio Alférez [b], David Fernandez-Amoros [a], Germán H. Alférez [c]

[a] Department of Software Engineering and Computer Systems, Universidad Nacional de Educacion a Distancia, Madrid, Spain
[b] INRIA, DiverSE Team, Rennes, France
[c] Facultad de Ingenieria y Tecnología, Universidad de Montemorelos, Mexico

ARTICLE INFO

Article history:
Received 18 November 2014
Accepted 5 August 2015
Available online xxx

Keywords:
Mass customization
Product platform
Variability modeling
Binary decision diagram

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.

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 (Liou, Yen, & Tzeng, 2010; Ngniatedema, Fono, & Mbondo, 2015; Simpson, Siddique, & Jiao, 2005; Takagoshi & Matsubayashi, 2013). For instance, van der Linden, Schmid, and Rommes (2007) report 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.) (Apel, Batory, Kästner, & Saake, 2013; van der Linden et al., 2007).

Product platforms usually offer a high number of features whose combination can produce a large quantity of derivatives (Sternatz, 2014). For instance, the BMW 7-Series platform supports 10^{17} derivatives (ElMaraghy et al., 2013). 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 (Bachmann & Clements, 2005). *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 (Pohl, Bockle, & Linden, 2005; Zhang & Tseng, 2007).

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 (Du, Jiao, & Chen, 2014; Yang et al., 2015), to diagnose and re-factor variability models (Zhang, 2014) (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

* Corresponding author. Tel.: +34 913 988 242.
E-mail addresses: rheradio@gmail.com, rheradio@issi.uned.es (R. Heradio), hperez@issi.uned.es (H. Perez-Morago), mauricio.alferez@inria.fr (M. Alférez), david@lsi.uned.es (D. Fernandez-Amoros), harveyalferez@um.edu.mx (G.H. Alférez).

http://dx.doi.org/10.1016/j.ejor.2015.08.005
0377-2217/

managers are rewarded with greater revenue when product platforms are extended (Jacobs, 2013). Nevertheless, this may increase costs and reduce profits (ElMaraghy et al., 2013; Patel & Jayaram, 2014; Salvador, Chandrasekaran, & Sohail, 2014; Takagoshi & Matsubayashi, 2013; Yenipazarli & Vakharia, 2015). In order to provide guidance on how maximize the profits and minimize the costs associated with product platforms, this paper describes how to analyze variability models 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 importance. 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 indirectly 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. Nevertheless, existing approaches to carry out such identification have the following limitations:

1. *Measures are rigid*. Measures are needed to account for feature incompatibility and relative importance. Although some measures have been proposed (Benavides, Segura, & Ruiz-Cortes, 2010; Boender, 2011; Cosmo & Boender, 2010; van Deursen & Klint, 2002; Zhang, Zhao, & Mei, 2004), 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 platform (Benavides et al., 2010). 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 percent 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 formulas and use off-the-self logic tools, such as SAT solvers (Batory, 2005) or Binary Decision Diagrams (BDDs) (Mendonça, 2009), to get the measures. Unfortunately, current algorithms have poor time performance (Fernandez-Amoros, Heradio, Cerrada, & Cerrada, 2014; Heradio, Fernandez-Amoros, Cerrada, & Abad, 2013; Mendonça, 2009).

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 (Benavides et al., 2010), *impact* and *exclusion sets* (Boender, 2011), feature *necessity* and *incompatibility* (Boender, 2011). 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 percent 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 (Huth & Ryan, 2004; Jackson, 2012), 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* (Kang, Cohen, Hess, Novak, & Peterson, 1990), and *decision models* (Reuse-driven software processes, 1993).

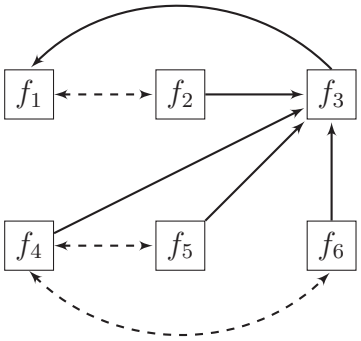

Fig. 1. A small variability model represented as a directed graph. Each node represents a feature and arrows depict inter-feature constraints.

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

Fig. 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.

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 Eq. (1) due to the feature interdependencies. For instance, $\{f_1, f_3, f_4, f_5\}$ is discarded because it violates the constraint $f_4 \dashleftarrow\!\dashrightarrow f_5$,

$$\text{valid derivatives} = \{\{f_1\}, \{f_1, f_3\}, \{f_1, f_3, f_6\}, \{f_1, f_3, f_5\}, \{f_1, f_3, f_5, f_6\}, \{f_1, f_3, f_4\}\}. \quad (1)$$

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:

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

Table 1
Summary of feature co-occurrences in Fig. 1.

Feature	Impact set	Necessity	Exclusion set	Incompatibility
f_1	$\{f_1, f_3, f_4, f_5, f_6\}$	$\frac{5}{6}$	$\{f_2\}$	$\frac{1}{6}$
f_2	$\emptyset$	0	$\{f_1, f_2, f_3, f_4, f_5, f_6\}$	1
f_3	$\{f_3, f_4, f_5, f_6\}$	$\frac{4}{6}$	$\{f_2\}$	$\frac{1}{6}$
f_4	$\{f_4\}$	$\frac{1}{6}$	$\{f_2, f_5, f_6\}$	$\frac{3}{6}$
f_5	$\{f_5\}$	$\frac{1}{6}$	$\{f_2, f_4\}$	$\frac{2}{6}$
f_6	$\{f_6\}$	$\frac{1}{6}$	$\{f_2, f_4\}$	$\frac{2}{6}$

1. *"Look at the feature distribution in all valid derivatives"* (Benavides, Segura, Trinidad, & Ruiz-Cortes, 2007; van Deursen & Klint, 2002; Fernandez-Amoros, Heradio, & Somolinos, 2009; Mendonça, Wasowski, & Czarnecki, 2009; Pena, Hinchey, Ruiz-Cortes, & Trinidad, 2006; Segura, 2008; Trinidad, Benavides, Duran, Ruiz-Cortes, & Toro, 2008; Trinidad & Cortes, 2009; Yan et al., 2009; Zhang, Mei, & Zhao, 2006; Zhang, Yan, Zhao, Jin, & Mei, 2008; Zhang et al., 2004). In particular, features that appear in all or none the valid derivatives are absolutely essential or dispensable, respectively. In the Software Product Line literature (Benavides et al., 2010), those features are known as *core* and *dead*, respectively.
   According to Eq. (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"* (Abate, Cosmo, Boender, & Zacchiroli, 2009; Boender, 2011; Boender, 2011; Boender, Barthe, Pardo, & Schneider, 2011; Cosmo & Boender, 2010). 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 (2011) 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.,
   $$\text{Impact Set}(f) = \{f' \cdot f' \Rightarrow f\}.$$
   (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.,
   $$\text{Exclusion Set}(f) = \{f' \cdot f \Rightarrow \neg f'\}.$$
   (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 $\#\mathcal{F}$, i.e.,
   $$\text{Necessity}(f) = \frac{\#\text{Impact Set}}{\#\mathcal{F}},$$
   $$\text{Incompatibility}(f) = \frac{\#\text{Exclusion Set}}{\#\mathcal{F}}.$$
   For instance, Table 1 summarizes feature co-occurrences in Fig. 1. According to Eq. (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 MOST or ALMOST NONE products is overlooked. For instance, according to Eq. (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'. Fig. 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.

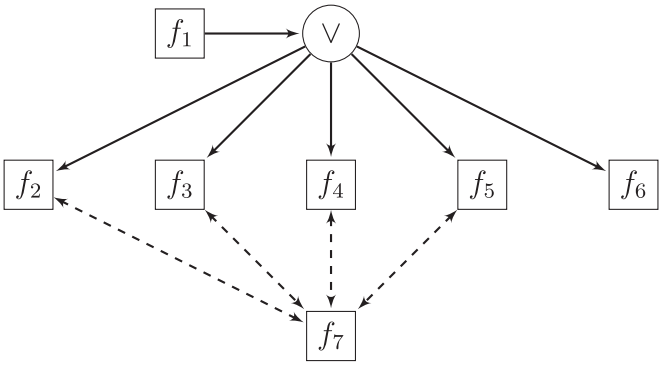

Fig. 2. Highly incompatible feature undetected by the current definition of exclusion set and incompatibility.

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) (Kang et al., 1990), Decision Diagrams (guidebook, 1993), the Configit language,[1] the SAP Product Configurator language,[2] the Oracle Configurator language,[3] etc. Interestingly, most of those notations are semantically equivalent (Czarnecki, Grünbacher, Rabiser, Schmid, & Wasowski, 2012; Schobbens, Heymans, Trigaux, & Bontemps, 2007). 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. (Zhang, Xu, Yu, & Jiao, 2012)

For instance, Eq. (2) shows the Boolean encoding of Fig. 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 $\neg f_2 \vee f_3$ means that f_2 requires f_3. The third row encodes three incompatibilities, for example $\neg f_1 \vee \neg f_2$ means that f_1 is incompatible with f_2,

$$\begin{aligned}\psi = {} & (f_1 \vee f_2 \vee f_3 \vee f_4 \vee f_5 \vee f_6) \wedge \\ & (\neg f_2 \vee f_3) \wedge (\neg f_3 \vee f_1) \wedge (\neg f_4 \vee f_3) \wedge (\neg f_5 \vee f_3) \wedge \\ & (\neg f_6 \vee f_3) \wedge (\neg f_1 \vee \neg f_2) \wedge (\neg f_4 \vee \neg f_5) \wedge (\neg f_4 \vee \neg f_6). \end{aligned} \quad (2)$$

The usual way to compute the core and dead features of a formula ψ with n features is described by Algorithm 1 (Benavides et al., 2007; van Deursen & Klint, 2002; Fernandez-Amoros et al., 2009; Heradio, Fernandez-Amoros, Cerrada, & Cerrada, 2011; Heradio et al., 2013; Mendonça et al., 2009; Pena et al., 2006; Segura, 2008; Trinidad et al., 2008; Trinidad & Cortes, 2009; Yan et al., 2009; Zhang et al., 2006; Zhang et al., 2008; Zhang et al., 2004). The idea is to repeatedly call the `sat_count` function, one time for each feature f, to get the number of satisfying assignments of $\psi \wedge f$. The most common approaches to compute `sat_count` are:

[1] http://configit.com/.
[2] https://scn.sap.com/docs/DOC-25224.
[3] http://www.oracle.com/us/products/applications/ebusiness/scm/051314.html.

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

Algorithm 1: get_core_and_dead_features *(straightforward approach).*

```
1  Input Boolean formula ψ; 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(ψ)
8      forall the f ∈ F do
9          count' = sat_count(ψ ∧ 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
```

Algorithm 2: get_impact_and_exclusion_sets *(straightforward approach).*

```
1  Input Boolean formula ψ; 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 the f ∈ F do
8          count = sat_count(ψ ∧ 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 the f' ∈ F\{f} do
16             count' = sat_count(ψ ∧ f ∧ 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. 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 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 (Cook, 1971), it is widely believed that the #SAT problem is even harder (Biere, Heule, & van Maaren, 2009).
2. Using a BDD engine. BDDs are an optimized way of representing Boolean functions by mean of rooted, directed, acyclic graphs (Berghammer & Bolus, 2012; Bryant, 1986). The Achilles heel of BDDs is its size, which depends on the variable ordering the BDD uses to encode ψ. 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 (Bollig & Wegener, 1996). For a BDD with m nodes, computing `sat_count` has complexity $O(m)$ (Bryant, 1986). So, the time complexity of Algorithm 1 for a model with n features is $O(mn)$.

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':

1. If sat_count($\psi \wedge f$) and sat_count($\psi \wedge f \wedge f'$) coincide then f belongs to the f' impact set.
2. If sat_count($\psi \wedge f \wedge f'$) is zero then f' belongs to the f exclusion set.

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

Boender (2011), Boender (2011), Boender et al. (2011) propose the alternative Algorithm 3 , which includes the following shortcut: for each feature f a satisfying assignment a of $\psi \wedge f$ is computed using the `sat_one` function (line 9), which has time complexity O(m) (Bryant, 1986). 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`($\psi \wedge f \wedge f'$) 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`($\psi \wedge f \wedge f'$). 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 organized 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:

$$\Pr(f) = \frac{\text{Number of valid derivatives that include } f}{\text{Total number of valid derivatives}}.$$

Table 2 summarizes feature probabilities for Fig. 1. For instance, looking at Eq. (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 α iif Eqs. (3) or (4) holds, respectively

$$\Pr(f) \geq (1-\alpha) \tag{3}$$

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

Algorithm 3: get_impact_and_exclusion_sets *(Boender's approach).*

```
1  Input Boolean formula ψ; 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;
   true_features, true_features: list;
4  begin
5      impact_sets = Hash.new
6      exclusion_sets = Hash.new
7      forall the f ∈ F do
8          count = sat_count(ψ ∧ f)
9          assignment = sat_one(ψ ∧ f) // get a satisfying
               feature assignment
10         true_features =
           get_features_with_true_value(assignment)
11         false_features =
           get_features_with_false_value(assignment)
12         if count == 0 then
13             impact_sets[f] = {}
14             exclusion_sets[f] = {f}
15         else
16             impact_sets[f] = {f}
17             exclusion_sets[f] = {}
18         forall the
           f' ∈ F\({f} ∪ exclusion_sets[f] ∪ true_features) do
19             count' = sat_count(ψ ∧ f ∧ f')
20             if count' == 0 then
21                 exclusion_sets[f].insert(f')
22                 exclusion_sets[f'].insert(f)
23         forall the f' ∈ F\({f} ∪ false_features) do
24             count' = sat_count(ψ ∧ f ∧ f')
25             if count == count' then
26                 impact_sets[f'].insert(f)
27     return impact_sets, exclusion_sets
```

Table 2
Feature probabilities for Fig. 1.

Feature	f_1	f_2	f_3	f_4	f_5	f_6
Pr(f)	1	0	$\frac{5}{6}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$

$$\Pr(f) \leq \alpha. \tag{4}$$

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 Eq. (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 α 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
Feature conditional probabilities for Fig. 1.

		Probability of					
		f_1	f_2	f_3	f_4	f_5	f_6
Conditioned to	f_1	1	0	$\frac{5}{6}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$
	f_2	0	0	0	0	0	0
	f_3	1	0	1	$\frac{1}{5}$	$\frac{2}{5}$	$\frac{2}{5}$
	f_4	1	0	1	1	0	0
	f_5	1	0	1	0	1	$\frac{1}{2}$
	f_6	1	0	1	0	$\frac{1}{2}$	1

Table 3 summarizes the conditional probabilities for all features in Fig. 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 ≥ 0.8 in column f_3).

Definition 5. The *exclusion set* of a feature f under sensitivity α 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.,

$$\text{Exclusion Set}_\alpha(f) = \{f' \cdot \Pr(f'|f) \leq \alpha\}.$$

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 ≤ 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 and dead features, and to compute impact and 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 ψ according to an input variable ordering. Remember from Section 3 that ψ is the Boolean representation of the input variability model. The variable ordering is specified using

Algorithm 4: get_core_and_dead_features *(flexible approach).*

```
1  Input Boolean formula ψ; var_ordering: array[0..n-1] of string;
   sensitivity ∈ [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(ψ, var_ordering)
8      feature_probabilities = get_prob(bdd, var_ordering)
9      i = 0
10     while i < length(var_ordering) do
11         if feature_probabilities[i] ≥ (1 − sensitivity) then
12             core_features.insert(var_ordering[i])
13         else if feature_probabilities[i] ≤ 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 ψ; var_ordering: array[0..n-1] of string;
   sensitivity ∈ [0, 1]
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; i, j: int;
4  begin
5      impact_sets = Hash.new
6      exclusion_sets = Hash.new
7      i = 0
8      while i < length(var_ordering) do
9          impact_sets[var_ordering[i]] = {}
10         exclusion_sets[var_ordering[i]] = {}
11         bdd = get_bdd(ψ ∧ var_ordering[i], var_ordering)
12         feature_probabilities = get_prob(bdd, var_ordering)
13         j = 0
14         while j < length(var_ordering) do
15             if feature_probabilities[j] ≥ (1 − sensitivity) then
16                 impact_sets[var_ordering[j]].insert(var_ordering[i])
17             if feature_probabilities[j] ≤ sensitivity then
18                 exclusion_sets[var_ordering[i]].insert(var_ordering[j])
19             j += 1
20         i += 1
21     return impact_sets, exclusion_sets
```

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 $\psi \wedge f$ (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).

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 (Bryant, 1986). 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 $\psi \equiv (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ as a running example for this subsection. Table 4 is the truth table for ψ. Fig. 3 is its BDD[4] representation using the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$.[5]

The remainder of this subsection is structured as follows. First, some definitions required to understand our algorithm are given. Next, the data structures the algorithm uses are described from a theoretical perspective. Then, the algorithm is presented. Finally, the algorithm computational cost is discussed.

Algorithm 6: get_prob.

```
1  Input bdd and var_ordering arrays
2  Output an array which stores Pr(x_i) in position i
3  var formula_sat_prob, total_prob: array[0..length(bdd)-1] of float;
4      prob: array[0..length(var_ordering)-1] of float; i: int;
5  begin
6      formula_sat_prob = get_formula_sat_prob(bdd)
7      get_marginal_prob(length(bdd)-1, total_prob,
       formula_sat_prob, prob, bdd, var_ordering)
8      for (i=0; i < length(var_ordering); i++) do
9          prob[i] = prob[i] / formula_sat_prob[1]
10     return prob
```

Table 4
Truth table for $\psi \equiv (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

x_1	x_2	x_3	x_4	ψ
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

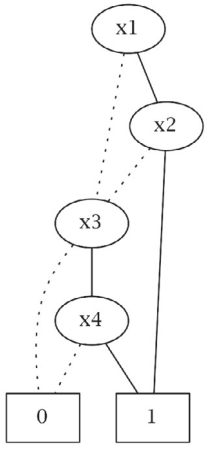

Fig. 3. BDD for ψ according to the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$. Nodes are labeled with the variables they encode. Edges to low children (dashed lines) or high children (solid lines) represent variable assignments to false or true, respectively.

[4] In popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram (Huth & Ryan, 2004). In this paper, we will follow that convention as well.

[5] Note 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).

4.3.1. Definitions

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

$$S_\psi = \{(x_1, \ldots, x_n) \cdot \psi(x_1, \ldots, x_n) = \text{true}\}. \quad (5)$$

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

$$S_{\psi|_{x_i=\text{true}}} = \{(x_1, \ldots, x_{i-1}, \text{true}, x_{i+1}, \ldots, x_n) \cdot \psi(x_1, \ldots, x_{i-1}, \text{true}, x_{i+1}, \ldots, x_n) = \text{true}\}. \quad (6)$$

For instance, according to Table 4, $\#S_\psi = 7$ since there are 7 rows where ψ evaluates to true,[6] and $\#S_{\psi|x_4} = 5$ because $x_4 = 1$ in 5 of the 7 rows where $\psi = 1$.

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

$$\Pr(\psi) = \frac{\#S_\psi}{2^n}. \quad (7)$$

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

$$\text{MPr}(\psi|_{x_i=\text{true}}) = \frac{\#S_{\psi|_{x_i=\text{true}}}}{2^n}. \quad (8)$$

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

$$\Pr(\psi|_{x_i=\text{true}}) = \frac{\#S_{\psi|_{x_i=\text{true}}}}{\#S_\psi}. \quad (9)$$

For instance, looking at Table 4, it is easy to see that $\Pr(\psi) = \frac{7}{2^4}$, $\text{MPr}(\psi|_{x_4=\text{true}}) = \frac{5}{2^4}$, and $\Pr(\psi|_{x_4=\text{true}}) = \frac{5}{7}$.

For convenience, in the remainder of the paper we denote $\Pr(\psi|_{x_i=\text{true}})$ and $\text{MPr}(\psi|_{x_i=\text{true}})$ as $\Pr(x_i)$ and $\text{MPr}(x_i)$, respectively, which is consistent with the notation introduced in Definition 1 (i.e., if a variability model is encoded by ψ then: $\#S_{\psi|_{f_i=\text{true}}}$ represents the number of derivatives than include f_i, $\#S_\psi$ is the total number of derivatives, and consequently, $\Pr(\psi|_{f_i=\text{true}}) = \Pr(f_i)$).

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:

```
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:

```
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.

Table 5
Content of the *bdd* array for Fig. 3.

Position	Index	Low	High	Mark
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

Table 6
Content of the *var_ordering* array for the Fig. 3.

Position	Content
0	"x_1"
1	"x_2"
2	"x_3"
3	"x_4"

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

4.3.3. Algorithm to compute feature probabilities

$\Pr(x_i)$ is computed jointly by Algorithms 6–8 . Fig. 4 summarizes the computations for the BDD in Fig. 3. Let us examine how our approach proceeds:

Algorithm 7: get_formula_sat_prob.

```
1  Input bdd array
2  Output an array which in position 1 stores Pr(ψ)
3  var formula_sat_prob: array[0..length(bdd)-1] of float; i: int
4  begin
5      for (i=0; i < length(bdd)-1; i++) do
6          formula_sat_prob[i] = 0.0 // non-root nodes prob is
               initialized to 0
7      formula_sat_prob[i] = 1.0 // root node prob is 1
8      i=length(bdd)-1
9      while i > 1 do // for all non-terminal nodes
10         formula_sat_prob[bdd[i].low] += formula_sat_prob[i] / 2.0
11         formula_sat_prob[bdd[i].high] += formula_sat_prob[i] / 2.0
12         i -= 1
13     return formula_sat_prob
```

[6] Throughout this paper 0/1 and false/true are used interchangeably.

Algorithm 8: get_marginal_prob.

```
1  Input v: 0..length(bdd)-1; total_prob, formula_sat_prob:
   array[0..length(bdd)-1] of float;
2      prob: array[0..length(var_ordering)-1] of float; bdd and
   var_ordering arrays
3  Output prob is passed by reference and, at the end of the
   algorithm execution,
4          it stores MPr(x_i) in position i
5  var prob_low, prob_high: float; i: int
6  begin
7      prob_low = 0.0
8      prob_high = 0.0
9      bdd[v].mark = not bdd[v].mark
       // explicit path recusive traversal
10     if bdd[v].low == 1 then
11         prob_low = formula_sat_prob[v] / 2.0
12     else if bdd[v].low ≠ 0 then
13         if bdd[v].mark ≠ bdd[bdd[v].low].mark then
14             get_marginal_prob(bdd[v].low, total_prob,
               formula_sat_prob, prob, bdd, var_ordering)
15         prob_low = total_prob[bdd[v].low] · (formula_sat_prob[v] / 2.0) / formula_sat_prob[bdd[v].low]
16     if bdd[v].high == 1 then
17         prob_high = formula_sat_prob[v] / 2.0
18     else if bdd[v].high ≠ 0 then
19         if bdd[v].mark ≠ bdd[bdd[v].high].mark then
20             get_marginal_prob(bdd[v].high, total_prob,
               formula_sat_prob, prob, bdd, var_ordering)
21         prob_high = total_prob[bdd[v].high] · (formula_sat_prob[v] / 2.0) / formula_sat_prob[bdd[v].high]
22     total_prob[v] = prob_low + prob_high
23     prob[bdd[v].index] += prob_high
       // implicit path iterative traversal
24     i = bdd[v].index + 1
25     while i<bdd[bdd[v].low].index do
26         prob[i] += prob_low / 2.0
27         i +=1
28     i = bdd[v].index + 1
29     while i<bdd[bdd[v].high].index do
30         prob[i] += prob_high / 2.0
31         i +=1
```

Algorithm 6 Computes $\Pr(x_i)$ as $\Pr(x_i) = \frac{\text{MPr}(x_i)}{\Pr(\psi)}$ by calling the auxiliary Algorithms 7 and 8.

Algorithm 7 Computes $\Pr(\psi)$. 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, $\text{MPr}(x_i)$ is the amount of water that node 1 has.

In Fig. 3, through node v_5 goes 1 liter (i.e., formula_sat_prob[5][7] = 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., formula_sat_prob[3] $= \frac{1}{2} + \frac{1}{2} = \frac{3}{4}$).

[7] According to Tables 5 and 6, the root node has label v_5 and it is in the position 5 of the *bdd* array.

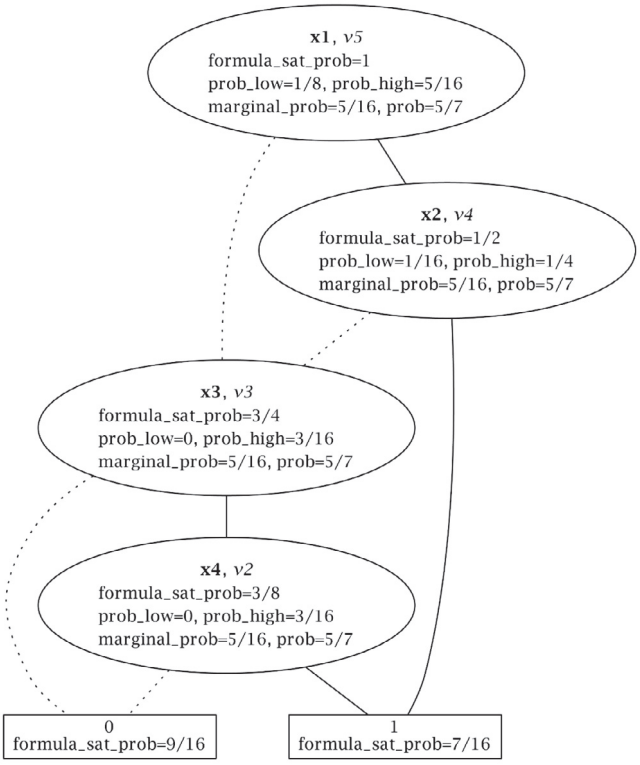

Fig. 4. Probability computation for BDD 3.

Algorithm 8 Computes $\text{MPr}(x_i)$. In particular, let us examine how it computes $\text{MPr}(x_2)$. In the truth Table 4, ψ evaluates to true when x_2 is true five times:

1. In four of them x_1 is true. When the call 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 bdd[4].low $\neq 0$, a recursive call is made to compute the total probability due to the low descendants of v_4 (i.e., get_marginal_prob(3, ...)). As a result:

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

Notice that prob_low_{x_2} is not simply equal to total_prob[3], because total_prob[3] depends also on the probability that comes from the link $v_5 \dashrightarrow v_3$. To get just the probability due to the link $v_4 \dashrightarrow v_3$, prob_low has to be adjusted using the formula_sat_prob array as:

$$\text{prob\_low} = \frac{\text{total\_prob}[3] \cdot \frac{\text{formula\_sat\_prob}[4]}{2}}{\text{formula\_sat\_prob}[3]} = \frac{\frac{3}{16} \cdot \frac{\frac{1}{2}}{2}}{\frac{3}{4}} = \frac{1}{16}.$$

Since bdd[4].high = 1, prob_high is directly computed as:

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

Finally:

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

2. In one of them x_1 is false. The two following implicit paths that have been removed from the reduced BDD: (i) $v_5 \dashrightarrow v_4 \dashrightarrow v_3$, and (ii) $v_5 \dashrightarrow v_4 \rightarrow v_3$. Nevertheless, path $v_5 \dashrightarrow v_4 \rightarrow v_3$ 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

Table 7
Variables iteratively traversed for BBD in Fig. 3.

Node	Arcs	Omitted vars that are traversed
v_5	$v_5 \dashrightarrow v_3$	x_2
	$v_5 \rightarrow v_4$	None
v_4	$v_4 \dashrightarrow v_3$	None
	$v_4 \rightarrow 1$	x_3, x_4
v_3	$v_3 \dashrightarrow 0$	x_4
	$v_3 \rightarrow v_2$	None
v_5	$v_5 \dashrightarrow 0$	None
	$v_5 \rightarrow 1$	None

instance, when the algorithm is called for v_5, the marginal probability of x_2 is updated with half the `prob_low` of v_5.

To sum up:

$$\begin{aligned} \mathrm{MPr}(x_2) &= \mathrm{MPr}(v_5 \dashrightarrow v_4 \rightarrow v_3) + \mathrm{MPr}(v_5 \rightarrow v_4) \\ &= \frac{\mathrm{prob\_low}_{v_5}}{2} + \mathrm{prob[bdd[4].index]} \\ &= \frac{\frac{1}{8}}{2} + \frac{1}{4} = \frac{5}{16}. \end{aligned}$$

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 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 Fig. 3. For instance, when v_4 is recursively traversed, the variables x_3 and x_4 need to be iteratively traversed because the edge $v_4 \rightarrow 1$ 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 $x_1 \prec x_2 \prec x_3 \prec x_4$). 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 1–5. 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 mnn'' steps where n'' is usually $< n$.
3. Algorithm 5 requires mnn' iterations and, on average requires less time than Algorithm 3, i.e., $n' \ll n''$.

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/.
2. A configuration model for laptops, which was reverse engineered from the DELL homepage on February 2009 by Nöhrer (Nöhrer, Biere, & Egyed, 2012; Nöhrer & Egyed, 2011; Nöhrer & Egyed, 2013). 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 gigahertz and 4 gigabyte RAM (although only one core was used). Our approach (Algorithms 4 and 5 parameterized with sensitivity = 0) outperforms related work (Algorithms 1–3) in all the experiments.

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., Amilhastre, Fargier, & Marquis, 2002; Bessiere, Fargier, Lecoutre, & Schulte, 2013; Cambazard & O'Sullivan, 2008; Gange, 2012; Hansen & Tiedemann, 2007; Hebrard, Hnich, O'Sullivan, & Walsh, 2005; Jensen, 2004; Kroer, 2012; Narodytska & Walsh, 2007; O'Sullivan, O'Callaghan, & Freuder, 2005; Queva, 2011). 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 percent 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.

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

Table 8
Computational complexity for Algorithms 1–5.

Core & dead features		Impact & exclusion sets		
Straightforward apch. (Algorithm 1)	Flexible apch. (Algorithm 4)	Straightforward apch. (Algorithm 2)	Boender's apch. (Algorithm 3)	Flexible apch. (Algorithm 5)
$O(mn)$	$O(mn')$ where $n' < n$	$O(mn^2)$		$O(mn'n)$ where $n' < n$

Table 9
Performance in seconds of Algorithms 1–5.

Variability model	Number of features	Dead & core features		Impact & exclusion sets		
		Alg. 1	Alg. 4	Alg. 2	Alg. 3	Alg. 5
Renault	398	5.07081	3.80750	823.38893	764.03980	587.12394
Dell	123	0.04938	0.03310	1.38710	1.31957	0.88028
Xtext	137	0.05547	0.00185	1.41291	1.67129	0.20609
Battle of tanks	144	0.11169	0.00200	2.81473	2.59596	0.21417
FM test	168	0.12117	0.04005	6.52654	6.67389	1.72019
Printers	172	0.12962	0.00271	2.53492	2.90606	0.21415
Banking software	176	0.15372	0.00397	3.69472	5.16730	0.32365
Electronic shopping	290	1.18944	0.37749	276.89974	224.35847	98.18766
Investments on enterprise information systems	366	17.86456	4.50573	6,002.73245	5,120.10355	952.13433

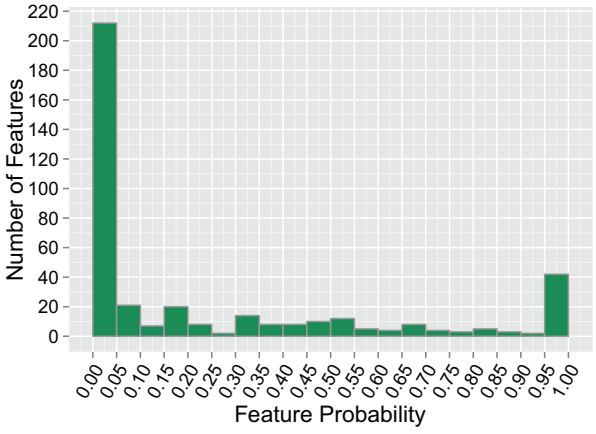

Fig. 5. Histogram of feature probabilities for the Renault Megane example.

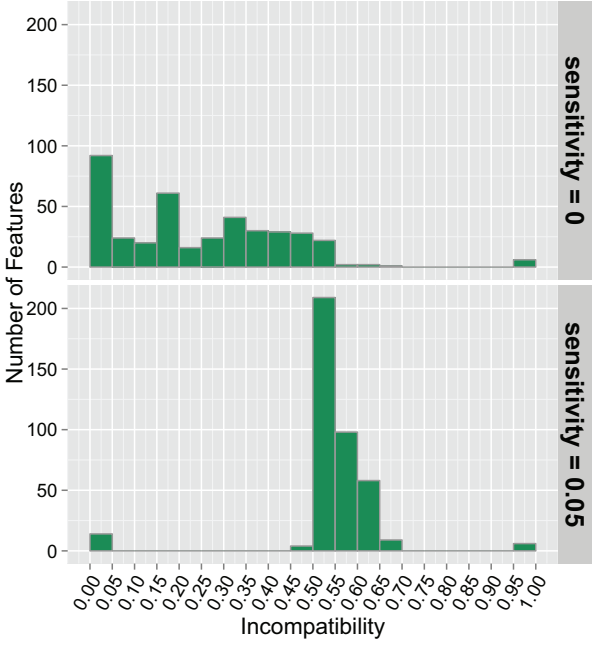

Fig. 6. Histogram of feature incompatibilities for the Renault Megane example.

Fig. 5 shows the histogram of feature probabilities. According to such figure, 53.27 percent 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 percent of the valid derivatives!

Fig. 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 percent of the remaining ones (8.29 percent of the features). When sensitivity is set to 0.5, that number becomes 380 (95.48 percent of the features).

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

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

Rigid measures (sensitivity=0) (percent)		Flexible measures (sensitivity=0.05) (percent)	
Dead features	1.51	53.27	Features with reuse probability ≤ 0.05
Core features	1.76	10.55	Features with reuse probability ≥ 0.95
Features incompatible with more than 50 percent of the remaining features	8.29	95.48	Features incompatible in 95 percent of the valid products with more than 50 percent of the remaining features

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

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 interrelations, 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.

Acknowledgments

The authors thank Roberto López Herrejón and Alexander Egyed at the Johannes Kepler University of Linz for their advice.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

Abate, P., Cosmo, R. D., Boender, J., & Zacchiroli, S. (2009). Strong dependencies between software components. In *3rd International symposium on empirical software engineering and measurement* (pp. 89–99). Florida, USA: Buena Vista.

Amilhastre, J., Fargier, H., & Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPS-application to configuration. *Artificial Intelligence, 135*(1–2), 199–234.

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). *Feature-oriented software product lines*. Springer.

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

Batory, D. (2005). Feature models, grammars, and propositional formulas. *9th International conference on software product lines* (pp. 7–20). Berlin, Heidelberg: Springer-Verlag.

Benavides, D., Segura, S., & Ruiz-Cortes, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems, 35*(6), 615–636.

Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortes, A. (2007). Tooling a framework for the automated analysis of feature models. *1st International workshop on variability modelling of software-intensive systems* (pp. 12–134). Ireland: Limerick.

Berghammer, R., & Bolus, S. (2012). On the use of binary decision diagrams for solving problems on simple games. *European Journal of Operational Research, 222*(3), 529–541.

Bessiere, C., Fargier, H., Lecoutre, C., & Schulte, C. (2013). Global inverse consistency for interactive constraint satisfaction. In *Principles and practice of constraint programming*. In *Lecture Notes in Computer Science: Vol. 8124* (pp. 159–174).

Biere, A., Heule, M. J., & van Maaren, H. (2009). Toby, Walsh, handbook of satisfiability. In *Frontiers in Artificial Intelligence and Applications: Vol. 185*. IOS Press.

Boender, J. (2011). Formal verification of a theory of packages. *Electronic Communications of the EASST, 48*, 1–9.

Boender, J. (March 2011). *A formal study of free software distributions*. Universite Paris Diderot (Ph.D. thesis).Ecole doctorale de Sciences Mathematiques de Paris Centre

Boender, J., Barthe, G., Pardo, A., & Schneider, G. (2011). Efficient computation of dominance in component systems (short paper). In *Software engineering and formal methods*. In *Lecture Notes in Computer Science: Vol. 7041* (pp. 399–406). Berlin, Heidelberg: Springer.

Bollig, B., & Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-Complete. *Transactions on Computers, 45*, 993–1002.

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

Cambazard, H., & O'Sullivan, B. (2008). Reformulating positive table constraints using functional dependencies. *14th International conference on principles and practice of constraint programming* (pp. 418–432). Sydney, Australia: Springer.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on theory of computing* (pp. 151–158). New York, NY, USA: ACM.

Cosmo, R. D., & Boender, J. (2010). Using strong conflicts to detect quality issues in component-based complex systems. *3rd India software engineering conference* (pp. 163–172). Mysore, India: ACM.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., & Wasowski, A. (2012). Cool features and tough decisions: A comparison of variability modeling approaches. *6th International workshop on variability modeling of software-intensive systems* (pp. 173–182). New York, NY, USA: ACM.

van Deursen, A., & Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology, 10*(1), 1–17.

Du, G., Jiao, R. J., & Chen, M. (2014). Joint optimization of product family configuration and scaling design by Stackelberg game. *European Journal of Operational Research, 232*(2), 330–341.

ElMaraghy, H., Schuh, G., ElMaraghy, W., Piller, F., Schönsleben, P., Tseng, M., et al. (2013). Product variety management. *CIRP Annals – Manufacturing Technology, 62*(2), 629–652.

Fernandez-Amoros, D., Heradio, R., & Somolinos, J. C. (2009). Inferring information from feature diagrams to product line economic models. In *13th International software product line conference* (pp. 41–50). San Francisco, CA, USA.

Fernandez-Amoros, D., Heradio, R., Cerrada, J., & Cerrada, C. (2014). A scalable approach to exact model and commonality counting for extended feature models. *IEEE Transactions on Software Engineering, 40*(9), 895–910.

Gange, G. K. (2012). *Combinatorial reasoning for sets, graphs and document composition*. Department of Computing and Information Systems. The University of Melbourne (Ph.D. thesis).

Reuse-driven software processes (1993). *Tech. rep. spc-92019-cmc, version 02.00.03*. Tech. rep., Software Productivity Consortium Services Corporation.

Hansen, E. R., & Tiedemann, P. (2007). Compressing configuration data for memory limited devices. *22nd national conference on artificial intelligence*. Vancouver, British Columbia, Canada: AAAI Press.

Hebrard, E., Hnich, B., O'Sullivan, B., & Walsh, T. (2005). Finding diverse and similar solutions in constraint programming. *20th National conference on artificial intelligence and the 17th innovative applications of artificial intelligence confHrence*. Pittsburgh, Pennsylvania, USA: AAAI Press/The MIT Press.

Heradio, R., Fernandez-Amoros, D., Cerrada, J., & Cerrada, C. (2011). Supporting commonality-based analysis of software product lines. *IET Software, 5*(6), 496–509.

Heradio, R., Fernandez-Amoros, D., Cerrada, J. A., & Abad, I. (2013). A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering, 23*(8), 1177–1204.

Huth, M. Ryan, M. (a). Logic in Computer Science: Modelling and Reasoning about Systems, 2004.

Jackson, D. (2012). *Software abstractions: Logic, language, and analysis* (2nd ed.). The MIT Press.

Jacobs, M. A. (2013). Complexity: Toward an empirical measure. *Technovation, 33*(4-5), 111–118.

Jensen, R. M. (2004). CLab: a C++ library for fast backtrack-free interactive product configuration. *10th International conference on principles and practice of constraint programming*. Toronto, Canada: Springer.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, S. (1990). Feature-oriented domain analysis (foda) feasibility study. *Tech. rep. cmu/sei-90-tr-21*. Software Engineering Institute.

Kroer, C. (2012). *Sat and SMT-based interactive configuration for container vessel stowage planning*. IT University of Copenhagen (Master's thesis).

van der Linden, F. J., Schmid, K., & Rommes, E. (2007). Software product lines in action. *The best industrial practice in product line engineering*. Berlin, Heidelberg: Springer-Verlag.

Liou, J. J., Yen, L., & Tzeng, G. H. (2010). Using decision rules to achieve mass customization of airline services. *European Journal of Operational Research, 205*(3), 680–686.

Mendonça, M. (2009). *Efficient reasoning techniques for large scale feature models*. University of Waterloo Ph.D. thesis.

Mendonça, M., Wasowski, A., & Czarnecki, K. (2009). Sat-based analysis of feature models is easy. In *13th international software product line conference* (pp. 231–240). San Francisco, CA, USA.

Narodytska, N., & Walsh, T. (2007). Constraint and variable ordering heuristics for compiling configuration problems. *Proceedings of the 20th international joint conference on artifical intelligence* (pp. 149–154). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ngniatedema, T., Fono, L. A., & Mbondo, G. D. (2015). A delayed product customization cost model with supplier delivery performance. *European Journal of Operational Research, 243*(1), 109–119.

Nöhrer, A., Biere, A., & Egyed, A. (2012). Managing sat inconsistencies with humus. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems, VaMoS '12* (pp. 83–91). New York, NY, USA: ACM.

Nöhrer, A., & Egyed, A. (2011). Optimizing user guidance during decision–making. *15th international software product line conference*. Munich, Germany: IEEE Computer Society.

Nöhrer, A., & Egyed, A. (2013). C2o configurator: A tool for guided decision-making. *Automated Software Engineering, 20*(2), 265–296.

O'Sullivan, B., O'Callaghan, B., & Freuder, E. C. (2005). Corrective explanation for interactive constraint satisfaction. *19th International joint conference on artificial intelligence* (pp. 1531–1532). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Patel, P. C., & Jayaram, J. (2014). The antecedents and consequences of product variety in new ventures: An empirical study. *Journal of Operations Management, 32*(1–2), 34–50.

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005

Pena, J., Hinchey, M., Ruiz-Cortes, A., & Trinidad, P. (2006). Building the core architecture of a multiagent system product line: With an example from a future NASA mission. In *7th international workshop on agent oriented software engineering*.Hakodate, Japan

Pohl, K., Bockle, G., & Linden, F. (2005). Software Product Line Engineering: Foundations. *Principles and techniques*. Berlin, Heidelberg: Springer-Verlag.

Reuse-driven software processes (November 1993). *Technical report spc-92019-cmc, version 02.00.03, Tech. rep*. Software Productivity Consortium Services Corporation.

Queva, M. (2011). *A framework for constraint-programming based configuration*. Technical University of Denmark (Ph.D. thesis)..

Salvador, F., Chandrasekaran, A., & Sohail, T. (2014). Product configuration, ambidexterity and firm performance in the context of industrial equipment manufacturing. *Journal of Operations Management, 32*(4), 138–153.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., & Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks, 51*(2), 456–479. Feature Interaction.

Segura, S. (2008). Automated analysis of feature models using atomic sets. *12th international software product line conference* (pp. 201–207). Ireland: Limerick.

Simpson, T. W., Siddique, Z., & Jiao, J. R. (2005). *Product platform and product family design: Methods and applications*. Springer.

Sternatz, J. (2014). Enhanced multi-hoffmann heuristic for efficiently solving real-world assembly line balancing problems in automotive industry. *European Journal of Operational Research, 235*(3), 740–754.

Takagoshi, N., & Matsubayashi, N. (2013). Customization competition between branded firms: Continuous extension of product line from core product. *European Journal of Operational Research, 225*(2), 337–352.

Trinidad, P., Benavides, D., Duran, A., Ruiz-Cortes, A., & Toro, M. (2008). Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software, 81*(6), 883–896.

Trinidad, P., & Cortes, A. R. (2009). Abductive reasoning and automated analysis of feature models: How are they connected?. In *3rd international workshop on variability modelling of software-intensive systems*. Sevilla, Spain.

Yan, H., Zhang, W., Zhao, H., Mei, H., Edwards, S., & Kulczycki, G. (2009). An optimization strategy to feature models verification by eliminating verification-irrelevant features and constraints. In *Formal foundations of reuse and domain engineering*. In *Lecture Notes in Computer Science: Vol. 5791* (pp. pp.65–75). Berlin, Heidelberg: Springer.

Yang, D., Jiao, J. R., Ji, Y., Du, G., Helo, P., & Valente, A. (2015). Joint optimization for coordinated configuration of product families and supply chains by a leader-follower Stackelberg game. *European Journal of Operational Research, 246*(1), 263–280.

Yenipazarli, A., & Vakharia, A. (2015). Pricing, market coverage and capacity: Can green and brown products co-exist? *European Journal of Operational Research, 242*(1), 304–315.

Zhang, L. L. (2014). Product configuration: A review of the state-of-the-art and future research. *International Journal of Production Research, 52*, 6381–6398. doi:10.1080/00207543.2014.942012.

Zhang, L. L., Xu, Q., Yu, Y., & Jiao, R. J. (2012). Domain-based production configuration with constraint satisfaction. *International Journal of Production Research, 50*(24), 7149–7166. doi:10.1080/00207543.2011.640714.

Zhang, M., & Tseng, M. M. (2007). A product and process modeling based approach to study cost implications of product variety in mass customization. *IEEE Transactions on Engineering Management, 54*(1), 130–144.

Zhang, W., Mei, H., & Zhao, H. (2006). Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering, 11*(3), 205–220.

Zhang, W., Yan, H., Zhao, H., Jin, Z., & Mei, H. (2008). A BDD-based approach to verifying clone-enabled feature models constraints and customization. In *High confidence software reuse in large systems*. In *Lecture Notes in Computer Science: Vol. 5030* (pp. pp.186–199). Berlin, Heidelberg: Springer.

Zhang, W., & Zhao, H. (2004). A propositional logic-based method for verification of feature models. In H. Mei, J. Davies, W. Schulte, & M. Barnett (Eds.), *Formal methods and software engineering*. In *Lecture Notes in Computer Science: Vol. 3308* (pp. pp.115–130). Berlin, Heidelberg: Springer.

Please cite this article as: R. Heradio et al., Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization, European Journal of Operational Research (2015), http://dx.doi.org/10.1016/j.ejor.2015.08.005