Received October 14, 2015, accepted November 2, 2015, date of publication November 9, 2015, date of current version November 24, 2015.

Digital Object Identifier 10.1109/ACCESS.2015.2498764

Efficient Identification of Core and Dead Features in Variability Models

HECTOR PEREZ-MORAGO, RUBEN HERADIO, DAVID FERNANDEZ-AMOROS, ROBERTO BEAN, AND CARLOS CERRADA
Department of Software Engineering and Computer Systems, Universidad Nacional de Educacion a Distancia, Madrid 28040, Spain

Corresponding author: R. Heradio (rheradio@issi.uned.es)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness through the CICYT Project under Grant DPI-2013-44776-R and in part by the Comunidad de Madrid within the RoboCity2030 Excellence Research Network under Grant S2013/MIT-2748.

ABSTRACT Mass customization enables the creation of personalized products that fulfill the features desired by specific customers. In this context, variability models are used to specify which configurable features are supported and which constraints among the features must be satisfied to guarantee the validity of the derived products. As the market demand grows and evolves, variability models become increasingly complex. In such entangled models, it is hard to identify which features are absolutely essential or dispensable because they are required to be included or excluded from all the possible products, respectively. This paper exposes the limitations of existing approaches to automatically detect those features and provides an algorithm that efficiently performs such detection.

INDEX TERMS Mass customization, product platform, variability model, configuration model, feature model, dead feature, core feature.

I. INTRODUCTION

To increase variety, improve customer satisfaction, reduce lead-times, and shorten costs, many companies have shifted from *mass production* to *mass customization* [1]. This shift of paradigm enriches the economies of scale with custom manufacturing flexibility by developing families of related products instead of single products [2], [3]. From this perspective, designing a product family is the process of capturing and modeling multiple product variants to satisfy different market niches. A generic architecture, named *product platform*, is designed to support the creation of customized products called *derivatives*.

Derivatives are usually specified as combinations of *features* demanded by the customers (e.g., get a car with *cruise control, speed limiter, directional stability control*, etc.) [4], [5]. As product platforms grow and evolve, the need for feature variability increases, and managing that variability becomes increasingly difficult. For instance, the number of derivatives for product platforms in the automotive industry may range from 10^3 for the smallest Peugeot and Nissan car models, to 10^{16} or 10^{21} for the BMW 3-Series and Mercedes C-Class, respectively [6]. 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. The features supported by a product platform and their inter-relations are often specified with variability models (also known as feature models, decision diagrams, configuration models, etc.) [7].

Features in a product platform usually have 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, or features that cannot be included in any derivative due to their incompatibility with highly required features. In particular, this paper is focused on the identification of features that must appear in all derivatives, and features that cannot be included in any derivative. In the software product line literature, those features are known as *core* and *dead*, respectively [8]–[10].

Identifying core and dead features by hand is both unfeasible and undesirable 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-shelf logic tools, such as SAT solvers [11] or Binary Decision Diagram (BDD) engines [12]. Unfortunately, current algorithms have poor time performance and thus impose long response times, hindering user interactivity [13].

To overcome this problem, this paper proposes a faster algorithm, which identifies dead and core features using BDDs. Since more complex logics than the Propositional one, which include integer arithmetic, transitive closure, etc., can be reduced to Boolean functions [14], and thus encoded as BDDs, our algorithm is general enough to support most variability model notations. + The remainder of this paper is structured as follows. Section II presents the running example that we will use to motivate and illustrate our work. Section III summarizes related work. Section IV describes our approach. Section V reports an experimental validation of our approach. Finally, Section VI outlines the conclusions of our work.

II. MOTIVATIONAL EXAMPLE

Figure 1 depicts 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 arrow depicts that the two connected features are incompatible. A solid-line arrow from a feature f to another feature f' represents that f requires f'.

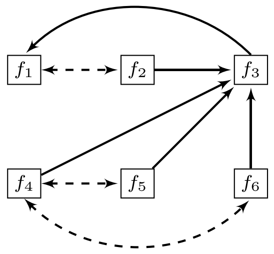

FIGURE 1. A small variability model represented as a directed graph. Each node represents a feature and arrows depict dependencies.

Each derivative is characterized by a selection of features. Feature interdependencies reduce the 2^6 potential derivatives in Figure 1 to just the 6 ones enumerated in Equation 1. For instance, $\{f_1, f_3, f_4, f_5\}$ is discarded because it violates the constraint $f_4 \dashleftarrow\dashrightarrow f_5$.

$$\text{valid derivatives} = \Big\{\{f_1\}, \{f_1, f_3\}, \{f_1, f_3, f_6\}, \{f_1, f_3, f_5\}, \{f_1, f_3, f_5, f_6\}, \{f_1, f_3, f_4\}\Big\} \tag{1}$$

As product platforms grow, variability models become bigger and harder to understand. Therefore, an automated mechanism is needed to provide information regarding the role of each feature in a variability model [15]. In particular, features that appear in all or none of the valid derivatives are essential or dispensable, respectively. For instance, 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.

III. RELATED WORK

A number of diferent notations are available to model the configurable options of a product family. Namely, Feature Diagrams (FD) [16], the Configit language,[1] the SAP Product Configurator language,[2] etc. Interestingly, most of those notations are semantically equivalent [17]. 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-shelf tools, such as SAT solvers, BDD engines, etc [15], [18].

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 $\neg f_2 \vee f_3$ 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.

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

The most straightforward approach to detect the core and dead features of a model is calling a SAT-solver to enumerate all valid derivatives, and then inspecting the derivatives to identify what features appear in all and none of them, respectively. Unfortunately, the number of derivatives represented by a variability model grows exponentially with the number of features. For instance, a model with 260 independent optional features (i.e., features without dependencies with the remaining ones) represents more combinations than the number of atoms in the observable universe [19]. So, this method, which is used for instance by the FAMA tool [20], has serious scalability limitations.

If ψ is a Boolean formula encoding a variability model, a feature f is core iff $\psi \rightarrow f$ is a tautology or, in other words, iff $\neg f \wedge \psi$ is unsatisfiable [14]. Similarly, f is dead iff $f \wedge \psi$ is unsatisfiable. Thus, all core and dead features can be detected by calling repeatedly a SAT-solver (or a BDD engine), which is the method proposed in [21] and [22].

As Lesta *et al.* [23] note, this approach can be improved for the case of dead features by reducing the number of checks as follows: whenever $f \wedge \psi$ is satisfiable, the SAT-solver not only returns "satisfiable", but also a valid assignment; all features f' that are true in that assignment cannot be dead (since, at least, they are included in the derivative returned by the solver). Therefore, checking the satisfiability of $f' \wedge \psi$ can be avoided.

Although Lesla et al.'s work does not deal with core feature detection, their reasoning can be applied to that kind of features as well: whenever $f \wedge \psi$ or $\neg f \wedge \psi$ are satisfiables, none of the features f' that are false in the satisfying assignment returned by the solver can be core.

[1]http://configit.com/
[2]https://scn.sap.com/docs/DOC-25224

IV. A BDD ALGORITHM TO DETECT CORE AND DEAD FEATURES

A. A BRIEF INTRODUCTION TO BDDs

BDDs are a way of representing Boolean functions. They are rooted, directed, acyclic graphs, which consist of decision nodes and terminal nodes [24]. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node v is labeled by a Boolean variable f and has two children called *low* and *high* (the relation of a node with its low and high children is usually depicted by dashed and solid lines, respectively). The edge from node v to a low (or high) child represents an assignment of f to 0[3] (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 rules are applied to its graph [14]:

- **R1** *Removal of duplicate terminals*. If a BDD contains more than one terminal *0*-node, then all edges which point to such a *0*-node are redirected to just one of them. We proceed in the same way with terminal nodes labelled with 1.
- **R2** *Removal of redundant tests*. If both outgoing edges of a node v_i point to the same node v_j, then v_i is eliminated, sending all its incoming edges to v_j.
- **R3** *Removal of duplicate non-terminals*. If two distinct nodes v_i and v_j in the BDD are the roots of structurally identical subBDDs, then we eliminate one of them, say v_j, and redirect all its incoming edges to the other one.

In popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram [14]. From here on, we will follow that convention as well.

In its seminal paper, Bryant [24] provided an automated procedure to build the BDD encoding of a Boolean formula according to a particular variable ordering. For example, Figure 2 is the BDD representation of Equation 2 using the variable ordering $f_1 \prec f_2 \prec f_3 \prec f_4 \prec f_5 \prec f_6$. To find if a particular configuration is valid, start at the root of the graph (v_8 in Figure 2) and traverse the edges as follows: if a feature appears in the configuration, then select its solid edge; otherwise select its dashed edge. If the path ends up in the 1-terminal node, the configuration is valid. Otherwise, the configuration is invalid. For instance, imagine that f_1 is not included in a configuration, then the dashed line coming from v_8 must be followed, which goes directly to the 0-terminal node. Hence, any configuration without f_1 is invalid, consistently with the fact that f_1 is a *core* feature (see Section II).

As another example, lets see if configuration $\{f_1, f_3\}$ is valid. Starting from the root, as f_1 belongs to the configuration, then the solid line that goes from v_8 to v_7 is selected. As f_2 is not in the configuration, the dashed line that goes from v_7 to v_6 is followed. f_3 belongs to the configuration, so the solid line from v_6 to v_5 is selected. Finally, as f_5 is not included in the configuration, the dashed line from v_5

[3] throughout this paper false/true and 0/1 are used interchangeably

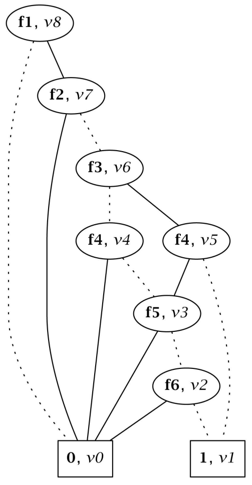

FIGURE 2. BDD for Equation 2 according to the variable ordering $f_1 \prec f_2 \prec f_3 \prec f_4 \prec f_5 \prec f_6$. Nodes are labeled with the variable they encode and with a unique identifier v_i. Edges to low children (dashed lines) or high children (solid lines) represent variable assignments to false or true, respectively.

to the 1-terminal node is followed. It is interesting to note that, at this point the BDD *does not need to know* the values of the remaining features f_5 and f_6 to determine that the configuration is valid.

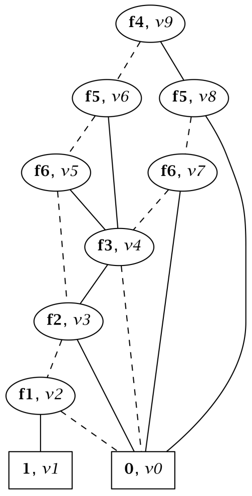

FIGURE 3. BDD for Equation 2 according to the variable ordering $f_4 \prec f_5 \prec f_6 \prec f_3 \prec f_2 \prec f_1$. This BDD is semantically equivalent to the one in Figure 2.

BDDs depend on the variable ordering used to build them. For instance, Figure 3 is the BDD representation of Equation 2 using the variable ordering $f_4 \prec f_5 \prec f_6 \prec f_3 \prec f_2 \prec f_1$. Note that this BDD requires traversing more nodes to determine if $\{f_1, f_3\}$ is a valid configuration (i.e., $v_9 \dashrightarrow v_6 \dashrightarrow v_5 \dashrightarrow v_3 \dashrightarrow v_2 \rightarrow v_1$). The BBD in Figure 3 has more nodes, and so the encoding is less efficient.

It is known that the size of a BDD is extremely sensitive to the used variable ordering [24]. Unfortunately, it is also well-known that finding an optimal ordering is an NP-complete problem [25]. Providing an heuristic to find a good variable ordering is out of the scope of this paper. Nevertheless, some heuristics specifically designed to deal with variability models are available [12], [26], [27].

B. THEORETICAL BASIS OF OUR APPROACH

A *valuation* of a formula ψ is an assignment of each variable in ψ to a truth value. For instance, a possible valuation for Equation 2 is $\{f_1 = 1, f_2 = 0, f_3 = 0, f_4 = 0, f_5 = 0, f_6 = 0\}$, which evaluates ψ to true. If ψ encodes a variability model, the valid products of the model are represented by the valuations that make ψ true [11]. For example, the former valuation represents the first derivative in Equation 1, which just includes the feature f_1.

In BDDs, valuations are represented as paths from the root to the terminal nodes. For instance, the BDD in Figure 2 encodes the aforementioned valuation as $v_8 \rightarrow v_7 \dashrightarrow v_6 \dashrightarrow v_4 \dashrightarrow v_3 \dashrightarrow v_2 \dashrightarrow 1$. A valuation is true if its corresponding path ends in the 1-terminal node.

Definition 1: Let $f \rightarrow_+ 1$ and $f \dashrightarrow_+ 1$ be two predicates. $f \rightarrow_+ 1$ is true iif the 1-terminal node is reachable from the root traversing the high edge of some node labelled with f; $f \dashrightarrow_+ 1$ is true iif the 1-terminal node is reachable through a low edge of some node labelled with f.

For example, in Figure 2, $f_3 \rightarrow_+ 1$ is true due to the path $v_8 \rightarrow v_7 \dashrightarrow v_6 \rightarrow v_5 \dashrightarrow 1$. Nevertheless, $f_1 \dashrightarrow_+ 1$ is false because v_8 is the only node labeled with f_1 and, once the low edge of v_8 is traversed, it is not possible to reach the 1-terminal node.

Lemma 1: A feature f is core iff $f \rightarrow_+ 1$ is true and $f \dashrightarrow_+ 1$ is false.

Lemma 1: A feature f is dead iff $f \rightarrow_+ 1$ is false and $f \dashrightarrow_+ 1$ is true.

Proof: By definition: a feature f (i) is core iff all valid products include f, and (ii) it is dead iff no valid derivative includes f. That is, when determining if f is core or dead, only the valid products are of interest or, in BDD terms, the paths from the root to the 1-terminal node. So, we are just concerned about the cases where f is traversed and the 1-terminal node is reached, i.e., when $f \rightarrow_+ 1$ is true or $f \dashrightarrow_+ 1$ is true:

1) If $f \rightarrow_+ 1 = true$ and $f \dashrightarrow_+ 1 = true$, the 1-terminal node may be reached traversing high and low edges of nodes labelled with f. That is, f is included in some valid products, but not in another valid ones. Therefore, f neither is core, nor is dead.
2) If $f \rightarrow_+ 1 = true$ and $f \dashrightarrow_+ 1 = false$, the 1-terminal node cannot be reached traversing low edges of nodes labelled with f (due to $f \dashrightarrow_+ 1 = false$). All the paths that end in the 1-terminal node go across a high edge of f. In other words, all valid products include f and so f is core.
3) If $f \rightarrow_+ 1 = false$ and $f \dashrightarrow_+ 1 = true$, the 1-terminal node cannot be reached traversing high edges of nodes labelled with f (due to $f \rightarrow_+ 1 = false$). All the paths that end in the 1-terminal node go across a low edge of f. As, f is disabled in all valid products, f is dead.

TABLE 1. Content of the *bdd* array for Figure 2.

position	index	low	high	mark
0	6	nil	nil	false
1	6	nil	nil	false
2	5	1	0	false
3	4	2	0	false
4	3	3	0	false
5	3	1	3	false
6	2	4	5	false
7	1	6	0	false
8	0	0	7	false

TABLE 2. Content of the *var_ordering* array for the Figure 2.

position	content
0	f_1
1	f_2
2	f_3
3	f_4
4	f_5
5	f_6

Algorithm 1 get_core_and_dead_features *(Our Approach)*

```
1  Input bdd and var_ordering arrays
2  Output two lists : one with all the core features, and
3      another one with all the dead features
4  var core_features, dead_features : list; i: int;
5      through_high_array, through_low_array :
6          array[0..n-1] of boolean;
7      it_reaches_the_1_terminal_array : array[0..$m-1$]
       of boolean
8  begin
9      core_features = { }
10     dead_features = { }
11     through_high_array = [false, false, ..., false]
12     through_low_array = [false, false, ..., false]
13     it_reaches_the_1_terminal_array = [false, false, ...,
       false]
14     does_it_reach_the_1-terminal?(length(bdd)-1,
15         through_high_array, through_low_array,
16             it_reaches_the_1_terminal_array)
17     for (i=0; i < length(var_ordering); i++) do
18         if
           through_high_array[i] $\wedge$ $\neg$through_low_array[i]
           then
19             core_features.insert(var_ordering[i])
20         else if
           $\neg$through_high_array[i] $\wedge$ through_low_array[i]
           then
21             dead_features.insert(var_ordering[i])
22     return core_features, dead_features
```

Algorithm 2 does_it_reach_the_1-terminal?

```
Input  v: index of a node in the bdd (i.e., 0..m-1);
    through_high_array, through_low_array :
        array[0..n-1] ofboolean;
    it_reaches_the_1_terminal_array :array[0..m − 1] of boolean
Output  the algorithm returns true if v can reach
        the 1-terminal. Otherwise, it returns false;
    through_high_array and through_low_array are passed
        by reference
begin
    if ¬bdd[v].mark then
        bdd[v].mark = true
        // does v reach the 1-terminal through
           high?
        if bdd[v].high == 1 then      // the 1-terminal is
        reached
            through_high_array[bdd[v].index] = true
            it_reaches_the_1_terminal_array[v] = true
            update_reduced_nodes(v, “high”,
                through_high_array, through_low_array)
        else if bdd[v].high ≠ 0 then      // keep searching
            it_reaches_the_1_terminal_array[v] =
                does_it_reach_the_1-terminal?(
                    bdd[v].high,
                        through_high_array, through_low_array,
                            it_reaches_the_1_terminal_array)
            if it_reaches_the_1_terminal_array[v] then
                through_high_array[bdd[v].index] = true
                update_reduced_nodes(v, “high”,
                    through_high_array, through_low_array)

        // does v reach the 1-terminal through
           low?
        if bdd[v].low == 1 then      // the 1-terminal is
        reached
            through_low_array[bdd[v].index] = true
            it_reaches_the_1_terminal_array[v] = true
            update_reduced_nodes(v, “low”,
                through_high_array, through_low_array)
        else if bdd[v].low ≠ 0 then      // keep searching
            it_reaches_the_1_terminal_array[v] =
                does_it_reach_the_1-terminal?(
                    bdd[v].low,
                        through_high_array, through_low_array,
                            it_reaches_the_1_terminal_array)
            if it_reaches_the_1_terminal_array[v] then
                through_low_array[bdd[v].index] = true
                update_reduced_nodes(v, “low”,
                    through_high_array, through_low_array)

    return it_reaches_the_1_terminal_array[v]
```

Algorithm 3 update_reduced_nodes

```
Input  v: 0..m-1; direction :  string ∈ {“high”, “low”};
    through_high_array, through_low_array :  array
    [0..n-1] of
        boolean
Output
through_high_array and through_low_array are passed
    by reference
var i: int
begin
    if direction == “high” then
        for (i = bdd[v].index + 1;
        i < bdd[bdd[v].high].index; i++) do
            through_high_array[i] = true
            through_low_array[i] = true
    else                    // direction == “low”
        for (i = bdd[v].index + 1;
        i < bdd[bdd[v].low].index; i++) do
            through_high_array[i] = true
            through_low_array[i] = true
```

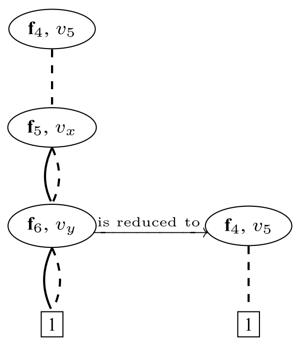

FIGURE 4. Algorithm 3 updates the *1*-reachability for features whose nodes has been removed from the BDD due to Reduction R2 (i.e., because their low and high outgoing edges point to the same node).

C. DATA STRUCTURES

Following the directions given by Bryant [24], 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.

TABLE 3. Summary of the experimental results.

Variability Model			Time-performance (in seconds)				
id.	Name	#features	Tartler. [22]		Lesta et al. [23]		Our apch.
			SAT	BDD	SAT	BDD	
m_1	Billing	88	0.02175	0.00143	0.01010	0.00070	0.00001
m_2	Coche ecologico	94	0.09131	0.00451	0.05008	0.00200	0.00002
m_3	UP estructural	97	0.08387	0.00321	0.05020	0.00170	0.00002
m_4	Xtext	137	0.07575	0.00513	0.04062	0.00270	0.00003
m_5	Battle of tanks	144	0.18828	0.00786	0.09803	0.00390	0.00003
m_6	FM Test	168	0.17512	0.01769	0.10501	0.00850	0.00009
m_7	Printers	172	0.12809	0.01177	0.06800	0.00510	0.00006
m_8	Banking software	176	0.19442	0.00921	0.09802	0.00530	0.00005
m_9	eShop	290	0.47985	0.04139	0.23400	0.02013	0.00031
m_{10}	EIS	366	0.74156	0.06534	0.44803	0.03404	0.00210
m_{11}	axTLS	108	0.07135	0.00228	0.03840	0.00121	0.00015
m_{12}	Fiasco	171	0.14856	0.00517	0.07864	0.00252	0.00024
m_{13}	uClibc	369	3.84395	0.42035	1.83082	0.18740	0.01362
m_{14}	BusyBox	881	10.04722	0.99665	4.93475	0.53460	0.04274
m_{15}	Android	88	0.79197	0.08849	0.38050	0.04031	0.00232
m_{16}	FM-500-50-1	500	27.70023	5.23822	14.42221	2.24335	0.18439
m_{17}	FM-1000-100-2	1000	70.11689	16.68317	37.82924	9.02403	0.29285
m_{18}	FM-2000-200-3	2000	325.68394	78.73022	175.9050	42.35382	2.80637
m_{19}	FM-5000-500-4	5000	1976.10194	406.55513	874.46547	226.93796	7.73044

- 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.

For instance, Tables 1 and 2 represent the content of *bdd* and *var_ordering* for the BDD in Figure 2, respectively.

D. ALGORITHMS

Algorithm 1 computes the core and dead features from an input formula ψ, which represents a given variability model. To do so, the algorithm builds the BDD that encodes ψ according to an input variable ordering (line 10). Two arrays named through_high_array and through_low_array are used to store in position $i-1$ the values of $f_i \rightarrow_+ 1$ and $f_i \dashrightarrow_+ 1$, respectively. The computation of such arrays is performed by Algorithm 2 (lines 13-14). For instance, after executing Algorithm 2 for Figure 2,

$$\text{through\_high\_array} = [\text{true, false, true, true, true, true}]$$
$$\text{through\_low\_array} = [\text{false, true, true, true, true, true}]$$

So,

$$\text{through\_high\_array}[1] = f_2 \rightarrow_+ 1 = \text{false}$$
$$\text{through\_low\_array}[1] = f_2 \dashrightarrow_+ 1 = \text{true}$$

Then, Algorithm 1 applies Lemmas 1 and 2 to identify the core and dead features according to through_high_array and through_low_array (lines 15-19). For instance, since $f_2 \rightarrow_+ 1 = \text{false}$ and $f_2 \dashrightarrow_+ 1 = \text{true}$, Algorithm 1 determines that f_2 is dead.

To compute $f \rightarrow_+ 1$ and $f \dashrightarrow_+ 1$, Algorithm 2 follows the procedure proposed by Bryant [24] for traversing a BDD and performing operations on its nodes. The algorithm is called at the top level with the root node as argument and with the mark fields of the nodes being all false. It then systematically visits every node in the graph by recursively visiting the subgraphs rooted by the two children. As it visits a node, it sets to true the value of the mark field, so that it can later determine whether a child has already been visited. For each node, it is checked if the 1-terminal node is reached. If so, Algorithm 3 is called to update arrays through_high_array and through_low_array to account for the nodes that have been removed from the BDD due to Reduction R2 (see Section IV-A). For instance, as Figure 4 shows, the edge $v_5 \dashrightarrow 1$ in Figure 2 is the result of erasing nodes v_x and v_y from the BDD because their high and low outgoing edges point to the same node. Taking that removal into account and as $f_4 \dashrightarrow_+ 1$ is true, then it follows that $f_5 \rightarrow_+ 1$, $f_5 \dashrightarrow_+ 1$, $f_6 \rightarrow_+ 1$ and $f_6 \dashrightarrow_+ 1$ are also true.

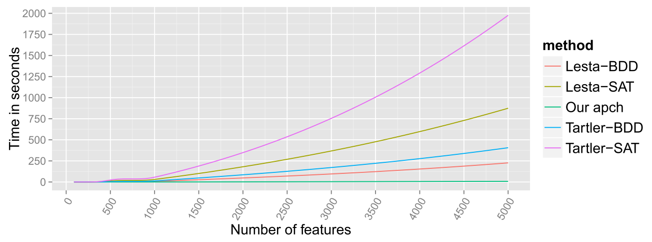

FIGURE 5. Graphical representation of the experimental results.

V. EXPERIMENTAL EVALUATION

This section reports the time-performance comparison of our approach with respect to Tartler.'s [22] and Lesta et al.'s [23] procedures.

A. EXPERIMENTAL DESIGN

Table 3 summarizes the benchmark we used to validate our approach. Variability models $m_1 - m_{10}$ come from the SPLOT repository,[4] models $m_{11} - m_{14}$ come from She et al. [28], [29], and models $m_{15} - m_{19}$ come from Bak [30], being $m_{16} - m_{19}$ randomly generated. All tests were conducted on an Intel Core™ i7-3537v 2.00 GHz with 8GB RAM. To improve the accuracy of the experimental results, we tried to minimize the operating system interference in the tests (e.g., due to interrupts to update internal OS kernel clock, automated task management and planing, memory garbage-collection, etc.) by repeating each experiment 50 times.

Our algorithm was implemented as an extension of the BDD library *BuDDy*.[5] As the implementation of alternative approaches to ours relies on SAT-technology, just comparing the performance of our algorithm to that implementation could skew the experimental validation to a SAT versus BDD contest. To overcome this problem, we also evaluated the performance of a BDD implementation of related work as well. In particular, Tartler.'s and Lesta et al.'s approaches were implemented with the SAT-solver *minisat*[6] and BuDDy. Finally, the orderings of the variables for all BDDs were computed using the heuristic proposed by Narodytska and Walsh [27].

The implementation of our algorithm and the benchmark used to validate its performance are available at:

`http://hperez30.github.io/CoreAndDeadFeatures/`

B. EXPERIMENTAL RESULTS

Table 3 and Figure 5 summarize the results of the conducted experimental evaluation. Note that, as each test was run 50 times, both the table and the figure show averaged data.

According to the results, our approach outperforms related work in every variability model, being its benefits more apparent as models are larger. It is worth noting that the BDD implementation of related work runs faster than the SAT implementation.

VI. CONCLUSIONS

Existing approaches to identify core and dead features in variability models rely on the use of logic engines as "black boxes". They follow the strategy of repeatedly calling predefined functions in the engines. We have shown that such strategy has poor performance and hinders user interactivity.

This paper has presented a new algorithm that directly interacts with the BDD that encodes a variability model. By efficiently traversing the BDD, we have been shown, both theoretically and experimentally, that our algorithm outperforms related work in terms of scalability and performance.

[4]http://www.splot-research.org/
[5]http://buddy.sourceforge.net/manual/main.html
[6]http://minisat.se/

REFERENCES

[1] T. W. Simpson, Z. Siddique, and J. R. Jiao, Eds., *Product Platform and Product Family Design: Methods and Applications*. New York, NY, USA: Springer-Verlag, 2005.
[2] R. Heradio, D. Fernandez-Amoros, L. de la Torre, and I. Abad, "Exemplar driven development of software product lines," *Expert Syst. Appl.*, vol. 39, no. 17, pp. 12885–12896, 2012.
[3] R. Heradio, D. Fernandez-Amoros, L. Torre-Cubillo, and A. P. Garcia-Plaza, "Improving the accuracy of COPLIMO to estimate the payoff of a software product line," *Expert Syst. Appl.*, vol. 39, no. 9, pp. 7919–7928, 2012.
[4] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Berlin, Germany: Springer-Verlag, 2007.
[5] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Germany: Springer-Verlag, 2013.
[6] F. K. Pil and M. Holweg, "Mitigating product variety's impact on the value chain," *Interfaces*, vol. 34, no. 5, pp. 394–403, 2004.
[7] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer-Verlag, 2005.
[8] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, Sep. 2010.
[9] R. Heradio, H. Perez-Morago, M. Alférez, D. Fernandez-Amoros, and G. H. Alférez, "Augmenting measure sensitivity to detect essential, dispensable and highly incompatible features in mass customization," *Eur. J. Oper. Res.*, vol. 248, no. 3, pp. 1066–1077, 2016.
[10] R. Heradio-Gil, D. Fernandez-Amoros, J. A. Cerrada, and C. Cerrada, "Supporting commonality-based analysis of software product lines," *IET Softw.*, vol. 5, no. 6, pp. 496–509, Dec. 2011.
[11] D. Batory, "Feature models, grammars, and propositional formulas," in *Proc. 9th Int. Conf. Softw. Product Lines*, Rennes, France, 2005, pp. 7–20.
[12] M. Mendonça, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, Dept. Comput. Sci., Univ. Waterloo, Waterloo, ON, Canada, 2009.
[13] R. Heradio, D. Fernandez-Amoros, J. A. Cerrada, and I. Abad, "A literature review on feature diagram product counting and its usage in software product line economic models," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 23, no. 8, pp. 1177–1204, Oct. 2013.
[14] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[15] R. Heradio, D. Fernandez-Amoros, H. Perez-Morago, and A. Adan, "Speeding up derivative configuration from product platforms," *Entropy*, vol. 16, no. 6, pp. 3329–3356, 2014.
[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Softw. Eng. Inst., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-90-TR-21, 1990.
[17] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, 2007.
[18] D. Fernandez-Amoros, R. Heradio, J. A. Cerrada, and C. Cerrada, "A scalable approach to exact model and commonality counting for extended feature models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 895–910, Sep. 2014.
[19] R. Flores, C. Krueger, and P. Clements, "Mega-scale product line engineering at general motors," in *Proc. 16th Int. Softw. Product Line Conf.*, 2012, pp. 259–268.
[20] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez, "FAMA framework," in *Proc. 12th Int. Softw. Product Line Conf.*, Limerick, Republic of Ireland, Sep. 2008, p. 359.
[21] C. Zengler and W. Küchlin, "Encoding the Linux kernel configuration in propositional logic," in *Proc. Eur. Conf. Artif. Intell. Workshop Configuration*, Lisbon, Portugal, 2010.
[22] R. Tartler, "Mastering variability challenges in Linux and related highly-configurable system software," Ph.D. dissertation, Dept. Comput. Sci. IV, Friedrich-Alexander-Univ. Erlangen-Nürnberg, Nuremberg, Germany, 2013.
[23] U. Lesta, I. Schaefer, and T. Winkelmann, "Detecting and explaining conflicts in attributed feature models," in *Proc. Workshop Formal Methods Anal. SPL Eng.*, London, U.K., Apr. 2015, pp. 31–43.

[24] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[25] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, Sep. 1996.
[26] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD—Foundations and Applications*. Berlin, Germany: Springer-Verlag, 1998.
[27] N. Narodytska and T. Walsh, "Constraint and variable ordering heuristics for compiling configuration problems," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, 2007, pp. 149–154.
[28] S. She, "Feature model synthesis," Ph.D. dissertation, Generative Softw. Develop. Lab, Univ. Waterloo, Waterloo, ON, Canada, 2013.
[29] T. Berger *et al.*, "A survey of variability modeling in industrial practice," in *Proc. 7th Int. Workshop Variability Modelling Softw.-Intensive Syst. (VaMoS)*, 2013, Art. ID 7.
[30] K. Bąk, "Modeling and analysis of software product line variability in Clafer," Ph.D. dissertation, Univ. Waterloo, Waterloo, ON, Canada, 2013.

HECTOR PEREZ-MORAGO received the B.S. degree in computer science and the M.S. degree in software engineering from the Universidad Nacional de Educacion a Distancia, Spain, in 2010 and 2014, respectively. He is currently completing the Ph.D. thesis with the Universidad Nacional de Educacion a Distancia. His research and interests include software engineering and computational logic.

RUBEN HERADIO received the M.S. degree in computer science from the Polytechnic University of Madrid, Spain, in 2000, and the Ph.D. degree from the Universidad Nacional de Educacion a Distancia, Spain, in 2007. Since 2001, he has been an Assistant Professor of Software Engineering with the Universidad Nacional de Educacion a Distancia. His research and teaching interests include software engineering, computational logic, and e-learning.

DAVID FERNANDEZ-AMOROS received the M.S. degrees in mathematics and philology from the Universidad Complutense de Madrid, Spain, in 1994 and 2009, respectively, and the Ph.D. degree in computer science from the Universidad Nacional de Educacion a Distancia, Spain, in 2004. Since 2001, he has been an Assistant Professor with the Universidad Nacional de Educacion a Distancia. His research and teaching interests include software engineering, natural language processing, and compiler design.

ROBERTO BEAN received the B.S. degree in computer science from the Universitat de Lleida, Spain, in 2001, and the M.S. degree in software engineering from the Universidad Nacional de Educacion a Distancia, Spain, in 2014. He is currently completing the Ph.D. thesis with the Universidad Nacional de Educacion a Distancia. His research and interests include software engineering and enterprise resource planning systems.

CARLOS CERRADA received the M.S. degree in industrial engineering and the Ph.D. degree from the Polytechnic University of Madrid, Spain, in 1983 and 1987, respectively. He was a Fulbright Scholar with the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA, from 1989 to 1990. He is currently a Full Professor with the Systems and Software Engineering Department, Universidad Nacional de Educacion a Distancia, Spain. His research interests are in software engineering and robotics.