Symbolic Model Checking of Product-Line Requirements Using SAT-Based Methods

Shoham Ben-David*, Baruch Sterin†, Joanne M. Atlee* and Sandy Beidu*
*David Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada
Email: {s3bendav,jmatlee,sbeidu}@uwaterloo.ca
†Department of EECS
University of California, Berkeley, CA, USA
Email: sterin@berkeley.edu

***Abstract*—Product line (PL) engineering promotes the development of *families* of related products, where individual products are differentiated by which optional features they include. Modelling and analyzing requirements models of PLs allows for early detection and correction of requirements errors – including unintended feature interactions, which are a serious problem in feature-rich systems. A key challenge in analyzing PL requirements is the efficient verification of the product family, given that the number of products is too large to be verified one at a time. Recently, it has been shown how the high-level design of an entire PL, that includes all possible products, can be compactly represented as a single model in the SMV language, and model checked using the NuSMV tool. The implementation in NuSMV uses BDDs, a method that has been outperformed by SAT-based algorithms.**

In this paper we develop PL model checking using two leading SAT-based symbolic model checking algorithms: IMC and IC3. We describe the algorithms, prove their correctness, and report on our implementation. Evaluating our methods on three PL models from the literature, we demonstrate an improvement of up to 3 orders of magnitude over the existing BDD-based method.

I. Introduction

Product-line engineering (PLE) is an increasingly popular approach to product development in which processes and practices are geared towards creating and managing a *family* of related products (e.g., smart phones, automobiles). Variability among products is characterized in terms of features, where a *feature* is a unit of functionality or variation. The product line maintains a collection of mandatory and optional features, and individual products are derived by selecting among and integrating features from this feature set. Companies that successfully employ PLE report dramatic improvements in productivity, quality, cost, labour needs, support for mass customization, and time to market [1].

We are particularly interested in requirements models of product lines, which we call *PL requirements*. We focus on requirements models because incorrect system requirements are a major source of software errors. Requirements analysis can reveal requirements errors early in the development process, when errors are easier and cheaper to fix [2]. In particular, in PLE, a common error is *feature interactions*, in which independently developed features behave differently when combined with other features. Determining how features *ought* to behave in combination is a requirements-engineering problem, thus early identification of undesired interactions that need to be fixed is essential. Another reason that we focus on requirements models is that they are smaller to analyze than other software artifacts are, and thus are more amenable to automated analyses.

A key challenge in analyzing PL requirements is efficient verification. Many traditional verification techniques are *product-based* [3], meaning that they can be used to assess models of individual products. In a product line of reasonable size, it is impractical to verify each derivable product individually because the number of possible products is exponential in the number of optional features. As a result, developers (especially of safety-critical products) might verify a small fraction of products and limit the choices that are offered to consumers, thereby foregoing one of the greatest assets of product-line engineering – the promise of mass customization.

As such, there is significant interest in *family-based* verification techniques [3] that analyze entire product lines. In these approaches, a product line and its family of products are modelled as a single artifact that represents all possible variable behaviour. Family-based verifiers operate on the product-line artifact, and either (1) guarantee that the desired property holds for all derivable products, or (2) identify the products that violate a desired property. Many verification techniques have been adapted to support family-based reasoning, including type checking [4], [5], constraint checking [6], [7], [8], equivalence checking [9], model checking [10], [11], [12], [13], and deductive verification [14].

For the analysis of requirements models we use symbolic model checking [15]. We note that when software is concerned, explicit state (rather than symbolic) model checking techniques are typically used. However, when verifying requirements, the picture changes. Many software requirements formalisms like statecharts [16], SCR [17], process algebras [18], [19], [20], RSML [21], Stateflow [22], and our Feature-Oriented Requirements Modelling Language [23] have a simplified execution semantics in which the software system is assumed to respond completely to one set of inputs, before the next set of inputs from the environment occurs. These

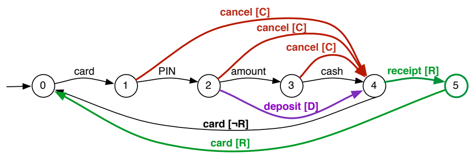

Fig. 1. Product line model of an automatic banking machine.

semantics map most naturally to the synchronous execution semantics of symbolic model checkers.

Classen et al. [10] showed how a product line can be modelled and verified using symbolic model checking [24]. The modelling language is an extended version of the SMV language [15] that supports feature modules. Each feature is modelled as a distinct module that extends a base system, and when composed together they form a single SMV model of the entire product line. If a behaviour in the product-line model is feature-specific, it is guarded by a corresponding *feature variable*, such that the behaviour is enabled or disabled depending on whether the feature is present or absent in a particular product. Consider the model of a simplified product line for an automatic banking machine (ABM) shown in Figure 1. In the base product (shown in black), the ABM authenticates the user, dispenses cash, and returns the card to the user. The product line includes three optional features – a deposit transaction (D), the issuing of a receipt (R), and a cancel operation (C), each shown in a different color – that can be combined in different ways to produce different products. The transitions that model feature-specific behavior are annotated with an expression over the feature variables. For example, the transition from state 5 to state 0 is executable only if the receipt (R) feature is present in the product; the transition from state 4 to state 0 is executable only if the receipt feature is absent from the product.

In PL model checking, the feature variables (D, R, and C in Figure 1) are Boolean variables that are assigned a value nondeterministically in the initial state and keep this value throughout the computation[1]. Each assignment to the feature variables represents a particular feature combination, whose corresponding enabled behaviours realize a single product. By analyzing all feature-variable assignments[2], the model checker analyzes the behaviours of all the possible products in the product line.

The symbolic model checker in Classen et al.'s work [10] is an adaptation of the NuSMV toolset [26][3]. Given an SMV model of a product line and a property in the CTL property language [28], the outputs of the extended model checker, when the property is violated in the model, are (1) an expression, in terms of the feature variables, representing all of the products that violate the property and (2) a counterexample trace from one of the violating products. Thus, in a single model-checking run, the user gets information regarding all products in the product line. The results of [10] report a significant speedup compared to model checking each of the products separately.

Classen et al. set nicely the basis for symbolic model checking of product line requirements. Their implementation, however, is far from optimal. They implemented their method on top of the classical symbolic model-checking algorithm [24], [15], using Binary Decision Diagrams (BDDs) [29]. This method is known to be significantly outperformed by modern techniques, that are based on satisfiability solving (SAT) [30], [31], [32], [33]. For the vast majority of cases, SAT-based symbolic model checking methods outperform BDD-based ones by an order of magnitude or more [34]. In this paper we investigate adding PL reasoning support to SAT-based algorithms.

We examine the two leading SAT-based symbolic model-checking algorithms, and show how each of them can be adapted to support PL model checking. The first is called IMC: it is Bounded Model Checking [30] combined with *interpolation* [32]. In this method, the model's transition relation is unfolded k times, and the model checker looks for a counterexample within this search space. If no counterexample exists within k execution steps, interpolation is used to try and prove that no counterexample exists in deeper states. The second method we investigated is the IC3 algorithm [31], [35], [36], which is considered the state-of-the-art symbolic model-checking algorithm. For a safety property P, the IC3 algorithm maintains a sequence of "frames" $R_0, ..., R_N$, where each frame R_j satisfies P and is an over approximation of the set of states that are reachable from the initial states in j or fewer steps. If two adjacent frames are found to be equal, it means that a fix point has been found, indicating that the property holds in all reachable states. Otherwise, a set of counterexamples is found during the computation.

We modify both of these algorithms to support PL model checking. In both methods, when a counterexample is detected, the model checker does not terminate. Instead, we slightly modify the model, removing from the search space the violating product(s) represented in the counterexample, and continue the model-checking procedure as if no counterexample has been found. The algorithm progresses in this manner, filtering out products that violate the property, until the property is found to hold on the modified model (i.e., the property holds in remaining products).

We have implemented our methods in the ABC toolset [37], on top of its existing IMC and IC3 utilities [38]. We evaluate our implementations on three product-line models. Two are taken from the literature – an Elevator and a Telephone System, both introduced by Plath and Ryan in [39]. The third is an automotive PL model, that we built based on feature requirement examples provided by General Motors.

[1]In this work, we assume that a product's feature configuration does not change dynamically as the product executes.

[2]If a *feature model* [25] is provided, which specifies the feature combinations of the *valid* products in the product line, then assignments to feature variables will be restricted to valid combinations.

[3]In [27], [11], Classen et al. present family-based explicit-state model checking, which uses a feature-aware variant of the Promela language and a new model checker, called SNIP, that is based on SPIN. In this paper we concentrate on symbolic model checking only.

Our experimental results demonstrate a dramatic improvement, ranging between one and three orders of magnitude faster than the BDD-based implementation. More importantly, our methods are capable of (easily) model checking examples that are too big to be tackled by the BDD method.

The rest of the paper is organized as follows. In Section II, we review the necessary definitions. Section III is the main section of the paper, where we describe our algorithms and prove their correctness. Section IV discusses our implementations in the ABC tool and in Section V we present our experimental results. Section VI gives an overview of related results, and in Section VII we conclude and discuss future directions.

II. Preliminaries

Let V be a finite set of Boolean variables. A *literal* is a variable $v \in V$ or its negation $\neg v$. A *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. Note that if c is a clause then $\neg c$ is a cube, and vise versa. A Boolean formula is in CNF if it is a conjunction of clauses, and in DNF form if it is a disjunction of cubes. For a CNF formula F, we denote by $\mathsf{CL}(F)$ the set of clauses of which F is composed. We use the notation $SAT?[F]$ for the satisfiability query, checking whether the Boolean formula F has a satisfying assignment.

A. Kripke Structures and Model Checking

A *model* M over V is a Kripke structure $M = (S, T, I, L)$, where S is a set of states, $T \subseteq S \times S$ is a total transition relation, $I \subseteq S$ is the set of initial states, and $L : S \to 2^V$ is the labeling function that assigns to each state the subset of variables that are true in the state. A state can also be seen as a cube of literals, one literal for every variable in V. For a state s and a variable $v \in V$, we denote by $s(v)$ the value (`true` or `false`) of v in s. We identify a Boolean formula B over the variables V with the set of states it represents: the set of all states satisfying B. If $(s, t) \in T$, we say that t is a successor of s. A *path* in M is a sequence of states $\pi = s_0, s_1, \ldots, s_n$ such that $s_0 \in I$ and for all i, $0 \leq i < n$, we have $(s_i, s_{i+1}) \in T$. We say that a state s is reachable in M if s appears on a path in M. The path π above is said to be of *length* n, and s_n on π is in *depth* n from the initial state.

We refer to the transition relation T in two different ways: (1) As a *Boolean formula*, $T(V, V_1)$ is defined over two copies of the set of variables. The copies might be called V_i and V_{i+1} (Section III-A), or V and V_1 (Section III-B). (2) As a *function*, $T : 2^S \to 2^S$ maps a set of states $G \subseteq S$ to the set of all successors of states from G. That is, $T(G) = \{t \mid \exists s \in G \cdot (s, t) \in T\}$.

An invariant property P is a set of states (represented by a Boolean formula). The *model checking* problem for a model M and property P is to determine whether all reachable states in M satisfy P. We say that a model M satisfies P (denoted $M \models P$).

B. Product Line Modeling

We follow [11] to model a product line as a Kripke structure. Let $M = (S, T, I, L)$ be a model over a set of variables V. We assume the existence of a special set of variables $V_F \subset V$, called *feature variables*. Literals of feature variables are called *feature literals*. The feature variables are assigned values in the initial state and keep their values forever. That is, for $(s, t) \in T$, we have that for all $v \in V_F$, $s(v) = t(v)$. Every assignment to the feature variables defines a *product* of the product line. A conjunction of literals of feature variables is called a *product cube*. We note that a product line model is usually accompanied by a *feature model* [25], which can be interpreted as a Boolean expression describing the allowed products in the product line. To accommodate this, we restrict the initial assignments to feature variables to include exactly those that satisfy the feature model [27].

III. PL Model Checking using SAT-based Methods

In this section we show how PL support can be added to the two leading symbolic model checking algorithms: IMC and IC3. We assume the model under verification describes a full PL as described in Section II-B above, and that the property to be verified is an *invariant* formula P. In Section IV-B we explain how complex CTL formulas are translated into invariant properties.

In both our methods we search for a counterexample; if one does not exist, it means that the property holds for *all* products. When a counterexample is found, it describes a violating product (might be a *set* of violating products in the IC3 case). We first try to $\mathsf{enlarge}$ the set of violating products as much as possible. Then, we modify the model by excluding from it all violating products found so far, and continue the model-checking procedure. We repeat this process until the property is found to hold in the modified model – at which point the rest of the products (those that have not been excluded from the model) satisfy the property.

At this stage, the set of violating products should be presented to the user. However, this set might grow very big (as the number of products grows exponentially in the number of feature variables) and should not be presented to the user as a list. We $\mathsf{simplify}$ this set to a compact Boolean expression that represents exactly the set of violating products.

In the rest of this section, we give the PL-support algorithms over IMC (Section III-A) and over IC3 (Section III-B). Since the $\mathsf{enlarge}$ and $\mathsf{simplify}$ procedures are common to both, we describe them together in Section III-C.

A. PL model checking with IMC

Let $M = (S, T, I, L)$ be a model over a set of variables V, and let P be an invariant formula. In Bounded Model Checking (BMC) [30], a counterexample is searched for among all paths of M up to a given depth k. For a given bound k, we introduce new copies $V_1, .., V_k$ of the set V (and for readability purposes, we define $V_0 = V$). A subscript i

on Boolean formulas over V indicates that the formula is expressed using the variables V_i. We denote by

$$\phi_k = \mathsf{unroll}(M, P, k)$$

the formula

$$\phi_k = I_0 \wedge T(V_0, V_1) \wedge T(V_1, V_2) \wedge ... \wedge T(V_{k-1}, V_k) \wedge \neg P_k.$$

ϕ_k represents an *unrolling* of the model M: it is a Boolean formula, representing all legal paths through M, that end in a buggy state violating P, in depth k steps from the initial state. If ϕ_k is satisfiable, the satisfying assignment produced by the SAT solver demonstrates a counterexample showing that $M \not\models P$. If ϕ_k is unsatisfiable, it means that no counterexample exists in depth k from the initial state. In the sequel, we use the terms "counterexample" and "satisfying assignment" interchangeably.

In order to verify a property P using BMC, we verify $\phi_k = \mathsf{unroll}(M, P, k)$ for $k = 0, 1, 2 \cdots$, until ϕ_k is satisfiable. Note that if P holds in the model (no counterexample exists), the BMC procedure as described above never ends. This problem is solved by applying *interpolation* [32] (thus the method is called IMC): when no satisfying assignment is found for depth k, interpolation is used to over-approximate the reachable state space. Interpolation involves several more calls to the SAT solver, searching for a fix point for the over-approximation of the reachable states. If such a fix point is found, P is proven to hold in M. If not, we need to increase k and try again.

Adding PL Support

In the method described above, the procedure terminates when either a counterexample is found or when interpolation proves that the property holds in the model. When the model describes an entire PL, we want to provide an answer for each of the products in the line. If the property holds for all the products (no satisfying assignment exists in the model), the output is the same; we need only to change the message to the user to say that the property *holds for all products*. When a satisfying assignment is found however, it provides a value to all the variables in V, and in particular, to the feature variables in V_F. The assignment to the feature variables defines a single violating product. The conjunction of the feature literals described by the assignment is a cube, pc, representing the violating product.

Our target is to find *all* violating products. To do that, we first try to enlarge the set of violating products that exist in the current depth (k). This procedure is described in Section III-C. Assume that pc is the output of the enlarge procedure, thus representing a set of violating products.

We modify the model checking problem by excluding the products pc from the model. For depth k, this is done by checking for the satisfiability of the formula:

$$\phi_k^{\text{pc}} = \phi_k \wedge \neg \text{pc}_0$$

Recall that pc_0 stands for the cube pc expressed in terms of the variables in V_0. The above formula thus disallows any product in pc to exist in the initial state of the BMC instance ϕ_k. Since feature variables keep their values (Section II-B), the conjunction with $\neg\text{pc}_0$ excludes all product satisfying pc from the model. Thus, if the above formula is satisfiable again, the satisfying assignment provided must demonstrate a product out of our pc set.

The full procedure is given in Algorithm 1. The variable

Algorithm 1: IMC^{PL}() – PL Model Checking Using BMC and Interpolation

```
1  Input: model $M = (S, T, I, L)$; property $P$; featureVars $V_F$;
2  Output: DNF formula VP;
3  cube pc, Cex;
4  int $k = 0$;
5  VP = FALSE;
6  while TRUE do
7      $\phi_k$ = unroll($M, P, k$);
8      while Cex = $SAT?[\phi_k^{VP}]$ do
9          pc = enlarge($\phi_k^{\text{VP}}$, Cex, $V_F$)
10         VP = VP $\vee$ pc;
11     end
12     (Result,$k$) = interpolation($\phi_k^{\text{VP}}$);
13     if (Result == TRUE) then
14         return simplify(VP);
15     end
16 end
```

VP (Violating Products) is a DNF formula, representing the products for which the property is found to fail so far. It is assigned FALSE in the beginning, and as the procedure progresses, violating products discovered are added to it. Cex on line 8 is the satisfying assignment, produced by the SAT procedure if ϕ_k^{VP} is satisfiable. We assume that Cex=0 if ϕ_k^{VP} is unsatisfiable, causing the while loop to exit. In the inner loop (lines 8-11), all counterexamples of length k are found. For each counterexample detected, we first try to enlarge it (line 9) to cover more products (see Section III-C1). The violating product cube pc is then added to the products already found (line 10), which guarantees the exclusion of it from the rest of the computation. When the inner loop exits, we are guaranteed that $\phi_k^{\text{VP}} = \phi_k \wedge \neg\text{VP}_0$ has no more satisfying assignments of length k.

We apply interpolation (line 12), to check whether P can be proven to hold on ϕ_k^{VP} in which case Result is TRUE. If P holds on ϕ_k^{VP}, no more violating products exist in the model, and we return a simplification of VP (line 14), describing exactly all violating products. Otherwise, the interpolation makes a few more unrolling iterations and returns a k that is larger than before. The model is unrolled again (line 7) with the larger k, and counterexamples are searched for deeper in the model.

In order to prove termination of our algorithm, we first claim that the inner while loop must terminate for every k. This is because in every iteration of the loop, all previously found violating products are removed from the computation. If a new satisfying assignment is found, it involves a product that has not been detected before. Since the number of products is finite, the loop must terminate. Note that if all products are violating, the DNF formula VP is equivalent to TRUE, thus the conjunction $\phi_k \wedge \neg\text{VP}_0$ is equivalent to FALSE and does not have a satisfying assignment.

Since the inner loop always terminates, the termination of IMC^{PL}() depends on the ability of interpolation to prove correctness of P in the modified model, which is guaranteed, for large enough k [32].

Theorem 1: When IMC^{PL}() terminates, VP describes exactly all the violating products in the model.

Proof 1: Products satisfying VP are all violating – they were added to VP either because they appeared in an explicit counterexample, or as a result of enlarge (which we assume to work correctly). When IMC^{PL}() terminates, it is because interpolation determined that no more counterexamples existed in ϕ_k^{VP}. This means that no more violating products exist in the model, thus VP satisfies exactly all of them.

B. PL Model Checking using IC3

IC3 is a SAT-based symbolic model-checking algorithm that does not involve unrolling of the model. Thus we use only two copies of the variables, V and V_1. IC3 is based on maintaining a sequence of "frames" $R_0, R_1, ..., R_N$. Each frame is a CNF formula over the variables V, representing a set of states in the model ($R_j \subseteq S$). Each frame R_j is an over-approximation of the set of states that are reachable from the initial states I in j or fewer steps. The frames R_j fulfill the conditions in Fig. 2 (adapted from [38]). The IC3 algorithm proceeds by

1) $R_0 = I$.
2) a) $R_j \subseteq R_{j+1}$.
   b) $\mathsf{CL}(R_{j+1}) \subseteq \mathsf{CL}(R_j)$, for $j > 0$.
3) $T(R_j) \subseteq R_{j+1}$.
4) $R_j \subseteq P$, for $j < N$.

Fig. 2. The conditions maintained by the frames of the IC3 algorithm

refining the frames, adding more clauses when possible, while maintaining the conditions of Fig. 2. One way of refining the frames, that we use in the sequel, is that of "pushing" clauses. A clause c in a frame R_j can be "pushed forward" to the following frame if the SAT query

$$SAT?[R_j \wedge T(V, V_1) \wedge \neg\mathsf{c}_1] \tag{1}$$

is not satisfiable. This means that starting from states in R_j and taking one step forward through T, we can never reach a state that does not satisfy the clause c. Thus, we can safely add c to R_{j+1} and this will not violate the conditions of Fig. 2.

The IC3 algorithm terminates in one of two cases:

1) For some j, $R_j = R_{j+1}$. In this case, a fix point of reachable states has been found, and thus $M \models P$.
2) An error cube $\mathsf{s}_I \subseteq I$ is found, from which a path to $\neg P$ exists. In this case $M \not\models P$.

The two termination cases above, together with the conditions in Fig. 2 and Query (1), are sufficient for understanding our product line support. For more details about the IC3 algorithm, refer to [31], [35], [36], [38].

Adding PL Support

Let $M = (S, T, I, L)$ be a model over a set of variables V, with a subset $V_F \subset V$ of feature variables. Let P be the invariant property to be verified. Our goal is to find all of the violating products – that is, all assignments to variables from V_F such that P fails to hold. As before, if P is found to hold in the model it means that it is satisfied for all products. We change the output to the user reporting that *P holds for all products*.

In the case where the property fails to hold in the model, we get a cube $\mathsf{s}_I \subseteq I$ from which a path exists to a state violating P. Like in the IMC case, we extract the product cube out of s_I, and enlarge it to get a cube pc, hopefully describing a set (larger than one) of violating products. In order to remove pc from the model we define modified frames[4]:

$$\forall 0 \leq j \leq N, \quad \hat{R_j} := R_j \wedge \neg\text{pc}. \tag{2}$$

Note that $\hat{R_j} \subseteq R_j$ for each j.

The product cube pc is put aside to be reported later to the user. The IC3^{PL} continues checking P, with the new frames $\hat{R_0}, ..., \hat{R_N}$ replacing the old ones. Note that by doing so, the algorithm actually checks P on a *modified model*, where states satisfying pc are filtered out. If another counterexample is found on the modified model, it identifies a different product cube pc$'$ that also fails to satisfy P. The product cube pc$'$ is set aside, and the frames are modified as before. The IC3^{PL} algorithm continues in this manner until the property is found to hold on the modified model.

The full procedure is given in Algorithm 2. The input to the IC3 function on line 7 is a sequence of frames (initially, the sequence of frames contains the initial states I and the full set of states S), and the output is an updated sequence of frames. IC3 returns also a Boolean Result whose value is TRUE if two frames are found to be equal; if the value of Result is FALSE, then the cube s_I is a counterexample that identifies initial states that lead to an error state. The function enlarge in line 11 is the one explained in Section III-C, which returns a cube of feature literals representing a set of violating products. We use the notation "Frames[$R_j \leftarrow (R_j \wedge \neg\text{pc})$]" (line 14) to denote the replacement of R_j by $R_j \wedge \neg\text{pc}$ in the set Frames. The function simplify in line 9 is the same as in Algorithm 1, and its functionality is described in Section III-C.

We claim that, when Algorithm 2 terminates, the set of products reported to the user is exactly the set of products in the PL that violate the property. To prove this, we first state the following.

[4]The clause $\neg$pc is in fact added to R_N only. In the IC3 algorithm, a clause added to R_j is automatically added to all R_i such that $i < j$. We explicate this in the description of our algorithm.

Algorithm 2: IC3^{PL}() – PL Model Checking Using IC3

```
1  Input: model $M = \{S, T, I, L\}$; property $P$; featureVars $V_F$;
2  Output: DNF formula VP;
3  sequence Frames = $\{I, S\}$; int result;
4  cube pc, $s_I$;
5  VP = FALSE ;
6  while TRUE do
7      (Result, Frames, $s_I$) = IC3($T$,Frames,$P$);
8      if Result == TRUE then
9          return simplify(VP);
10     end
11     pc = enlarge(Frames, $s_I, V_F$);
12     VP = VP ∨ pc;
13     for $R_j \in$ Frames do
14         Frames = Frames[$R_j \leftarrow (R_j \wedge \neg$pc)]
15     end
16 end
```

Proposition 1:. Let $R_0, ..., R_N$ be frames satisfying the conditions of Fig. 2. Then $\hat{R_0}, ..., \hat{R_N}$, defined as in Equation 2, satisfy the conditions of Fig. 2 as well, where I is replaced by $\hat{I} = I \wedge \neg$pc.

Proof 2: Conditions 1, 2 and 4 of Fig. 2 hold trivially. For condition 3, note that

$$T(\hat{R_j}) = T(\hat{R_j}) \wedge \neg\text{pc}. \qquad (3)$$

To see why, recall that $\hat{R_j} = R_j \wedge \neg$pc (Equation 2), thus states satisfying pc do not exist in $\hat{R_j}$. Since feature variables keep their values, no state satisfying pc can exist in $T(\hat{R_j})$. Thus, the conjunction of $T(\hat{R_j})$ with $\neg$pc does not change the set.

Since $\hat{R_j} \subseteq R_j$ it follows that $T(\hat{R_j}) \subseteq T(R_j)$. Given that R_j satisfies the conditions of Fig. 2 (the premise of the proposition), we have $T(R_j) \subseteq R_{j+1}$, which allows us to deduce $T(\hat{R_j}) \subseteq R_{j+1}$. By conjuncting $\neg$pc to both sides, we get $T(\hat{R_j}) \wedge \neg\text{pc} \subseteq R_{j+1} \wedge \neg$pc. By Equations 2 and 3, we conclude what we need:

$$T(\hat{R_j}) \subseteq \hat{R}_{j+1}.$$

Note that in every call to IC3, the list of frames might grow. However, new frames that are introduced will include the product clauses, because product clauses satisfy the pushing condition (Query 1). Like in proposition 1, this is because feature variables keep their values. By adding the clause $\neg$pc to the frames, we modify the model on which P is verified, in a way that the products in pc are disabled. Note also that the products in pc are the only products disabled by adding $\neg$pc to the frames. Since the new frames satisfy the conditions of Fig. 2, when IC3 returns TRUE, we are guaranteed that P holds in the modified model.

A special case that should be noted is when *all* products in the product line violate P. Suppose that our product list already holds several product cubes, and assume that the pc identified in the current iteration of the loop represents all the remaining products in the product line. Adding $\neg$pc to the frames would make each R_j a contradicting CNF formula. This would make every clause of each frame pushable to the next frame (since Query 1 would be unsatisfiable). Thus, when opening a new frame, two frames will be found equal and the algorithm would terminate correctly.

Following the discussion above we can now state the theorem:

Theorem 2: When the algorithm of Fig. 2 terminates, VP includes exactly all of the violating products.

C. Enlarge *and* Simplify *a Set of Products*

The algorithms in Sections III-A and III-B both use the functions enlarge and simplify. In this section we explain their functionalities and how they are implemented in our tool. We note that other implementation options exist that achieve the same functionalities.

1) The enlarge function: Our methods discussed in the previous subsections, detect violating products one at a time. Our target is to get more violating products at each iteration. In order to do that, we take the formula ϕ for which a counterexample was found, and try to find feature variables whose values are not needed for determining the satisfying assignment. This is the purpose of enlarge.

The functionality of enlarge is based on a capability possessed by modern SAT solvers, among them MiniSat [40] that we use in our implementation. MiniSat allows literals to be added to a SAT formula ϕ as "*assumptions*". When ϕ is found to be unsatisfiable, MiniSat returns the set of assumptions used to prove that no satisfying assignment existed. While this is not necessarily a minimal set of assumption, it is often smaller than the original set.

Let ϕ_k = unroll(M, P, k) be as in Section III-A, and Cex the satisfying assignment produced for it. In order to enlarge our set of assignments, we use a well known trick based on the assumption capability. We construct a new formula φ, built of several components. The first is ϕ'_k = unroll$(M, \neg P, k)$, that is similar to ϕ but requires P to hold at depth k (see Section III-A). The second component is a set of unit clauses, one for each literal defined by Cex, except for the feature literals. The last component is the set of feature literals, added as assumptions.

For example, let $V = \{v_1, v_2, v_3, v_4\}$ with $V_F = \{v_3, v_4\}$, and let Cex be $(v_1 = 0, v_2 = 1, v_3 = 0, v_4 = 1)$. Our formula φ will be defined as follows:

$$\varphi = \phi' \wedge \neg v_1 \wedge v_2; \quad \mathsf{assumption}(\neg v_3, v_4).$$

Note that φ is unsatisfiable: it requires that P would be TRUE in cycle k, but it forces the values of Cex (because of the unit clauses) which must cause P to be FALSE (since it is a satisfying assignment to the original ϕ).

When given to the SAT solver, φ is immediately found to be conflicting, and a conflict clause in terms of the feature literals is produced. In our example, let this conflict clause be $(\neg v_3)$.

This means that the value of v_4 does not make a difference, thus both $(\neg v_3, v_4)$ and $(\neg v_3, \neg v_4)$ are failing products.

2) The Simplify function: The list of violating products is stored in our algorithms as a DNF formula VP, satisfying exactly all the violating products. As discussed before, this formula can easily get too big to be presented to the user. For example, in a model with 10 features, if half of the products are violating, VP describes 512 products.

There exist more than one way for simplifying a DNF formula. In our implementation, we use a method by Morreale [41] known as "Irrelevant Sum of Products" (ISOP). This algorithm takes as input a truth table representing a Boolean formula f and calculates the simplest possible DNF formula that is equivalent to f.

In ABC, where we implemented our algorithms, the ISOP method is implemented, supporting formulas of up to 16 Boolean variables. Our program constructs a truth table representing the list of products, and uses the ISOP service of ABC to simplify it. The simplification using ISOP took a fraction of a second to complete. Since this step depends on the number of features in the model and not on the size of the model itself, the simplification is expected to have very little influence on the overall runtime models of any size. When the number of features in the model is larger than 16, other simplification methods should be considered, for example, using BDDs.

We note that our output will always be the simplest *DNF* formula, but not necessarily the simplest *formula*. For example, the DNF formula $(v_1 \wedge \neg v_2) \vee (v_1 \wedge \neg v_3)$ is equivalent to $v_1 \wedge (\neg v_2 \vee \neg v_3)$, which is simpler. In contrast, the output from the NuSMV implementation depends on the last BDD order in the model-checking run. While in many cases it is the simplest formula possible, we have seen cases where, due to the BDD order, the expression includes irrelevant feature variables.

IV. Implementation

We implemented our methods in the ABC toolset [37], on top of its existing IC3 and IMC functions. In the subsections below we describe the ABC tool and the conversion of safety CTL properties into invariant properties.

A. The ABC Tool

ABC is an open-source software system for the synthesis and verification of sequential logic circuits represented as And-Inverter Graphs (AIGs) [42]. ABC uses logic optimization and implements many algorithms from the literature, including synthesis, equivalence checking and various model checking algorithms, among them BMC, interpolation and IC3. The tool uses a dialect of MiniSat [40] as its SAT solving facility, and supports the ISOP [41] method, which we use to simplify the resulting set of products.

To be accepted by the ABC toolset, a model should be given in the AIGER language [42]. For our experiments, we translated models written in NuSMV to AIGER in two steps: (1) We invoked NuSMV with the option *output Boolean model* (-obm), to translate a complex NuSMV model into a simpler one that is flat (i.e., has no modules) and Boolean (i.e., has no multi-valued variables). (2) We used the utility smvtoaig from the AIGER distribution [42] to translate the Boolean model from (1) into AIGER.

B. Translating safety ACTL Properties to Invariant Properties

Our current implementations support only invariant properties, of type AGp, with p being a Boolean formula. We were able to translate all safety ACTL properties into $AG(p)$ type ones, accompanied by auxiliary state machines (that were added to the SMV model). A property is in ACTL if, when represented in Negation Normal Form (where negation is over atomic propositions only), it does not contain the E path quantifier. An ACTL property is considered a "safety" property if it does not contain the AF or AU operators. For properties that use the AU operator, but are otherwise safety ACTL properties, we converted AU into AW (weak-until). While this weakened the meaning of the properties, it allowed them to be included in our experiments.

For a safety ACTL formula φ, the translation was done in two phases. First, we converted φ into a regular expression R_φ [43]. We note that this conversion is limited to safety properties that are in the common fragment of ACTL and LTL [44]. Luckily, all safety properties in our case belong to this fragment. We then used the algorithm of [45] to build a state machine S_φ in the SMV language, together with an $AG(p)$ type property. The results of [43], [45] guarantee the correctness of the translation, but we also validated our implementation by running NuSMV on the two versions of each property, and comparing the results.

V. Experimental Results

We ran our experiments on three PL requirement models. Two are taken from the literature: the Elevator and Telephone models introduced by Plath and Ryan [39]. In order to build a PL model containing all features, we used Classen et al. method and tool [10]. The third model is based on an industrial product line model of automotive software controllers, first presented in [46]. We elaborate more on these models in the subsections below.

The experiments compare the performance of our two new methods with that of the BDD-based method of Classen et al. [10] implemented in the NuSMV tool [26]. For the NuSMV experiments, we performed a preprocessing phase: we invoked each model with the dynamic BDD ordering option (-dynamic), and saved the final variable ordering in a file. Consequent runs, recorded in the tables, were invoked with the saved variable ordering file as an initial order, and did not perform dynamic reordering.

The runs were performed on an AMD FX-6100 machine with six cores at 3.3GHz and 32GB of RAM. The output results, indicating the set of products for which the formula fails to hold, were the same for all three methods, for all runs (except for those where NuSMV could not terminate).

TABLE I

EXPERIMENTAL RESULTS FOR THE ELEVATOR MODEL. TIME IS GIVEN IN SECONDS. (*) THE PRODUCT SET IS ¬Shuttle ∧ (Antiprank ∨ Empty)

Property	Res	Products	BDD	IMC	IC3	Speedsup
Quick-close $\rightarrow AG(door = open \rightarrow AX(door = close))$	$\surd$	None	0.86	0.02	0.02	× 43
$AG(door = open \rightarrow AX(door = close))$	X	¬Quick-close	8.12	0.14	0.37	× 58
Shuttle $\rightarrow AG((floor = 2 \wedge direc = up) \rightarrow AX(direc = up))$	$\surd$	None	0.97	0.02	0.02	× 48
$AG((floor = 2 \wedge direc = up) \rightarrow AX(direc = up))$	X	¬Shuttle	999	0.33	0.55	× 3027
Shuttle $\rightarrow AG((floor = 3 \wedge direc = down) \rightarrow AX(direc = down))$	$\surd$	None	1.00	0.02	0.03	× 50
$AG((floor = 3 \wedge (direc = down)) \rightarrow AX(direc = down))$	X	¬Shuttle	2088	2.02	1.20	× 1740
$AG(But2 \wedge \neg But3 \rightarrow A[\neg(floor = 3 \wedge door = open)W(floor = 2 \wedge door = open)])$	X	ALL	230	0.21	1.14	× 1095
Overloaded $\rightarrow AG((over \wedge door = closed) \rightarrow AX\neg(door = closed))$	$\surd$	None	0.95	0.03	0.03	× 31
$AG((over \wedge (door = closed)) \rightarrow AX\neg(door = closed))$	$\surd$	¬Overloaded	9.8	0.22	0.52	× 44
Overloaded $\rightarrow AG((floor = 3 \wedge over) \rightarrow A[(floor = 3)W\neg over]$	$\surd$	None	1.02	0.06	0.04	× 25
$AG((floor = 3 \wedge over) \rightarrow A[(floor = 3)W\neg over]$	X	¬Overloaded	12.5	0.43	1.30	× 29
$AG((floor = 3 \wedge \neg But3 \wedge (direc = up)) \rightarrow (door = closed))$	X	ALL	180	0.19	1.08	× 947
$AG((floor = 3 \wedge \neg But3 \wedge (direc = down)) \rightarrow (door = closed))$	X	ALL	497	2.11	3.31	× 235
$AG((floor = 2 \wedge \neg But8 \wedge (direc = up)) \rightarrow A[(direc = up)W floor = 8])$	X	See (*)	74	0.47	1.56	× 159
$AG((floor = 8 \wedge \neg But1 \wedge (direc = down)) \rightarrow A[(direc = down)W floor = 1])$	X	See (*)	68	2.1	1.36	× 50

A. The Elevator Model

The Elevator model we use was first introduced by Berry in [47]. Plath and Ryan in [39] augmented the model with features, demonstrating how feature interactions can be detected using model checking. Classen et al. in [10] generalized the model, added features, and made it configurable, allowing the composition of part or all the features into a single PL model. We use Classen et al.'s model (taken from their site [48]), in its largest configuration possible, containing 8 floors and 9 features. The feature set in our model includes Park, which sends the elevator to a designated floor to park; Too-full, ignoring calls when elevator is too full to load more people, Executive-floor which gives priority to one of the floors; Overloaded, preventing the doors from closing when the elevator is overloaded; Open-if-idle, leaving the doors open when parked; Empty, which cancels calls inside the elevator if it is empty; Antiprank, ignoring calls inside the elevator when active; Shuttle, continuously moving, even when no calls are made, and Quick-close, closing doors one time step after opening. For a full description of the Elevator model and its features, refer to [39], [10].

After composing all features, the model had 40 state variables, making it a relatively small model. Out of about 30 properties accompanying the Elevator model, only ten were safety properties supported by our methods. The rest were mixed universal and existential CTL properties, or liveness ones. Five of the properties were conditioned on the existence of a certain feature in the product tested. For those, we introduced also an unconditioned version. Table I presents the results, for each of the properties. Column 2 of the table lists the output of the run – true ($\surd$) for all products, or false (X) for some (or all) of them. Column 3 gives the expression describing the violating products (hence, this value is 'None' when the formula holds for all products). For example, the second property in Table I is violated in all products where Quick-close is not present. Columns 4,5,6 present the time, in seconds, for the BDD, IMC, and IC3 methods respectfully. The last column calculates the speedup between the IMC method, which is the fastest, and the BDD method. We note that while the speedup gets up to 3000 times better, the run times are relatively short for all cases.

B. The Telephone Model

Our second case study is a simple version of a Telephone system, introduced by Plath and Ryan in [39]. The system models a network of 4 phones, each with a different set of functionalities. We manually translated the system and feature modules, that were written in a dialect of the SMV language (Cadence-SMV), into the NuSMV dialect. We were able to compose 7 features, using Classen et al.'s composer [48] to get a PL model with 76 state variables. Most of the features were applied to Phone 1 only, some were applied also to Phone 2. This is indicated by an extension -1 or -2 to the feature name. The list of features, taken from [39], is given below.

Call Forward Unconditional (Cfu-1): All calls to the subscriber's phone are redirected to another phone. Call Forward on Busy (Cfb-1 and Cfb-2): All calls to the subscriber's phone are redirected to another phone, if the line is busy. Call Forward on No Reply (Cfnr-1 and Cfnr-2): All calls to the subscriber's phone are redirected to another phone, if not answered in a certain amount of time. Ring Back When Free (Rbwf-1): If the user gets the busy-tone on calling another line, the feature can be activated. A call to the same line will be reissued as soon as it becomes idle. Terminating Call Screening (Tcs-1): Reject calls to the subscriber's phone from numbers on a given screening list. Full description of the Telephone model can be found in [39].

The Telephone model includes 37 properties, 23 of which are safety ACTL properties. We ran all of them, and we report the results for 10 of them, in Table II. The rest of the properties were found to fail on all products, and have run times that are comparable to those presented in Table II. While the Telephone model is almost twice as large as the Elevator, run times were not longer, and seemed to vary less than in the Elevator

TABLE II
EXPERIMENTAL RESULTS FOR THE TELEPHONE MODEL

Property	**Result**	**Products**	**BDD**	**IMC**	**IC3**	**Speedup**
$AG(ph\_1.st = talking \wedge ph\_1.dialled = 2 \rightarrow ph\_2.st = talked)$	X	ALL	212	0.18	0.43	×1177
$AG(ph\_2.tcs\_msg \rightarrow ((ph\_2.dialled = 1 \wedge ph\_1.tcs2) \vee (ph\_2.dialled = 4 \wedge ph\_4.tcs2)))$	X	Cfu-1 ∨ Cfb-1	80	0.20	0.36	×401
$AG(\neg(ph\_1.cfu\_forw = 0) \rightarrow AG(\neg(ph\_1.st\ in\{ringing, talked\})))$	X	¬Cfu-1	219	0.31	0.25	×876
$AG(ph\_2.tcs\_msg \rightarrow ((ph\_2.dialled = 1 \wedge ph\_1.tcs2) \vee(ph\_2.dialled = 4 \wedge ph\_4.tcs2)))$	X	Cfu-1 ∨ Cfb-1	80	0.20	0.36	×401
$AG((ph\_1.tcs2 \wedge ph\_1.st = ringing) \rightarrow ph\_3.st = ringingt)$	X	ALL	160	0.14	0.30	×1142
$AG(ph\_3.tcs\_msg \rightarrow ((ph\_3.dialled = 1 \wedge ph\_1.tcs3) \vee(ph\_3.dialled = 4 \wedge ph\_4.tcs3)))$	X	Cfu-1 ∨ Cfb-1	80	0.21	0.39	×387
$AG((ph\_1.st = trying) \rightarrow AX((ph\_1.st = ringingt) \vee (ph\_1.st = busyt)))$	X	Cfb-2	203	0.17	0.33	×1194
$AG((ph\_1.st = talking \wedge ph\_1.dialled = 3) \rightarrow ph\_3.st = talked)$	$\surd$	None	44	0.02	0.03	×1110
$AG(ph\_1.tcs3 \rightarrow AG\neg(ph\_3.dialled = 1 \wedge ph\_3.st\ in\{ringingt, talking\}))$	X	¬Tcs-1	97	0.21	0.25	×562
$AG(p\_4.tcs2 \rightarrow AG\neg(ph\_2.dialled = 4 \wedge ph\_2.st\ in\{ringingt, talking\}))$	X	¬Tcs-1	84	0.21	0.23	×401

example. The significant advantage of our methods over the BDD one, however, is maintained here, with all SAT-based runs completing the verification in less than half a second.

C. The Autosoft Model

Our third and largest PL Requirement model is an automotive example, called *Autosoft*. This model is derived from feature requirement examples provided by General Motors and was manually modelled in the FORML language [46], which supports precise modelling of feature requirements for software product lines. Full details of the FORML features (which are very abstract versions of the features in the original document), can be found in [49]. The features were automatically composed into a PL model, which was then semi-automatically translated, using a chain of tools being deployed by the last author, into the SMV language. Its analysis, using a model checker, is presented here for the first time.

The model consists of 10 features. BDS is the Basic driving service, which responds to the drivers's requests. Cruise Control (CC) maintains the vehicle's speed, while Headway Control (HC) maintains the vehicle's headway distance from road objects. Lane Change Alert (LCA) is responsible for issuing an alert if the driver tries to change lanes under unsafe conditions and Forward Collision Alert (FCA) issues an alert whenever there is danger of a collision with an object ahead. Headway Personalization (HP) saves the last cruise-headway setting of a driver. Speed Limit Control (SLC) overrides CC's computation of the car's acceleration, whenever cruising speed is greater than the speed limit of the road, and Lane Centring Control (LCC) periodically adjusts the car's orientation to centre the vehicle in its current lane. Lane Change Control (LXC) automatically changes the vehicle's lane to a driver selected lane, provided the conditions are safe. Finally, Driver Monitoring System (DMS) issues an alert whenever the driver is not attentive.

For our experiment, we introduced a set of safety properties, listed below.

1) If LXC is activated by the driver, then within two cycles, either LCC will be automatically activated, or a request will be issued for the driver to take control.
2) If LXC is activated, then so is LCA.
3) When LCC is active and the driver turns the steering wheel, then LCC should be temporarily deactivated.
4) Whenever a driver issues an accelerate command, the speed of the car must increase.
5) The car always tries to keep a certain minimum headway distance from the car ahead.
6) If a driver issues both an accelerate and decelerate commands, then the speed of the car should reduce.
7) When the car's ignition is off and the driver requests an ignition turn on, then the car will be turned on.

The Autosoft model in the SMV language consists of 526 state variables, and was too large to be analyzed with the BDD implementation. We left the model checker running for 33 hours for property (1), and terminated it after realizing that all of the available memory had been exhausted. For the rest of the properties, we set a cutoff time of 7200 second (2 hours). None of the BDD runs were capable of evaluating the property with a 2-hour limit.

In an attempt to cope with the size problem, we experimented with applying cone-of-influence (COI) reduction to the model. This service of NuSMV reduces a model based on the property to be verified, leaving only the parts of the model that are relevant for the verification. The COI reduction is applied separately for each property, to get a reduced model customized for the property. After applying COI, the run times for the SAT based methods were reduced (significantly, in some cases). The BDD method however, was not capable of completing any of the runs within the 2 hour limit, even for the reduced models.

Table III presents two results (in seconds) for each of the 7 properties, for the SAT based methods. The first is the run time for the original model, with no reduction (the "Size" column gives the number of state variables in the model), and the second is the time for the reduced model, after COI reduction was applied. The notation $AX[1..2](p)$ (first property of Table III) stands for $AX(p \vee AXp)$. Note that IC3 was able to solve each property in less that 4 minutes, and IMC in less than one minute.

TABLE III
EXPERIMENTAL RESULTS FOR THE AUTOSOFT MODEL. (*) THE SET OF PRODUCTS IS $(\mathsf{BDS} \wedge \mathsf{LCC} \wedge \neg\mathsf{HP} \wedge \mathsf{CC}) \vee (\mathsf{BDS} \wedge \mathsf{LCC} \wedge \mathsf{HC} \wedge \mathsf{CC})$. (**) THE SET OF PRODUCTS IS $(\mathsf{BDS} \wedge \neg\mathsf{DMS} \wedge \mathsf{HC} \wedge \mathsf{CC} \wedge \neg\mathsf{LXC}) \vee (\mathsf{BDS} \wedge \mathsf{LCC} \wedge \mathsf{HC} \wedge \mathsf{CC})$.

Property	**Size**	**Result**	**Products**	**BDD**	**IMC**	**IC3**
$AG(LXC\_execute \rightarrow AX[1..2](LCC\_active \vee LXC\_takeOverAlert))$	526	X	ALL	—-	48	180
	259			—-	47	125
$AG(LXC\_active \rightarrow AX(LCS\_engaged))$	526	X	ALL	—-	50	224
	257			—-	48	118
$AG((\mathsf{LCC} \wedge LCC\_active \wedge Steer \wedge (Steer\_angle > 0)) \rightarrow AX(LCC\_inactive))$	526	X	See (*)	—-	21	82
	258			—-	20	62
$AG(Accelerate \rightarrow AX(Car\_speed \geq Car\_speed\_pre))$	526	X	ALL	—-	6.8	24
	261			—-	6.0	15
$AG((\mathsf{HC} \wedge (Car\_headway < Car\_setHeadway)) \rightarrow AX(Car\_goalSpeed \leq Car\_goalSpeed\_pre))$	526	X	See (**)	—-	8.2	25
	260			—-	7.5	23
$AG(Accelerate \wedge Decelerate \rightarrow AX(Car\_speed \leq Car\_speed\_pre))$	526	$\surd$	None	—-	3.7	7
	261			—-	1.6	4.2
$AG(BDS\_off \wedge IgniteOn \rightarrow AX((Car\_ignition = on) \wedge BDS\_on)$	526	$\surd$	None	—-	2	13
	204			—-	1.6	9.3

Threats to Validity

NuSMV provides many configuration 'flags' in which a model-checking run can be invoked. Thus, our configuration might not have been the optimal one. Finding the best configuration, however, is itself extremely time consuming. Also, while our new methods significantly outperform the old one, it is possible that part of this difference has to do with the differences in the underlying tool infrastructures.

VI. RELATED WORK

Family-based symbolic model checking has been proposed in several works. Plath and Ryan [39] were the first to suggest the use of SMV to model and verify feature *interactions*. They proposed an extension to the SMV input language that allowed modifying an existing system with additional features. Interactions between features can then be verified by model checking a property φ before and after adding a new feature. The work of Classen et al. [10] was mentioned before and it is the closest to ours. While our method performs much better, it currently supports safety ACTL properties only, whereas Classen et al. support the full CTL language. We note, however, that it is common practice in the model-checking world to compromise expressiveness for better performance.

Researchers have also proposed family-based explicit-state model checking. Lauenroth et al. [13] showed how product lines could be modeled as an extension to I/O automata. They adapted the original CTL model-checking algorithm of Clarke et al. [28] to verify a product-line model. A variability model, which specifies the legal products of the product line, ensures that the model checker explores only legal feature combinations. Gruler et al. [12] showed how features could be modelled in CCS [18] and integrated into a product line. The product line is represented as a labelled transition system whose transitions are annotated with feature combinations; the model-checking algorithm returns a set of products that satisfy a given formula. Classen et al. [50], [27] implemented what they call a (family-based) semi-symbolic model-checking algorithm that searches the explicit state space of products, but represents sets of products in a symbolic data structure. We consider this work as being in the camp of explicit-state model checkers. In general, explicit-state and symbolic model-checking approaches are complementary and cannot be directly compared, since the models have different execution semantics (interleaving vs. synchronized execution). One chooses between explicit-state and symbolic model-checking based on the characteristics of the problem to be verified.

Another related area is that of *variability-aware* model checking [51], [52] in which modelling languages and model-checking algorithms accommodate variability (e.g., optional behaviour), but the variability is not packaged into features. As a result, the output of the model checker is not in terms of features and products that violate a property.

Cimatti et al. [53] consider the synthesis of parameters for infinite state models. They use IC3 with an SMT (rather than SAT) solver, in the target of finding parameters that are feasible for a given system. While the context of their work is very different from ours, their IC3 algorithm to find the set of relevant parameters is close in nature to our algorithm for finding violating products.

VII. CONCLUSION

We demonstrated how PL support can be added to SAT based symbolic model checking methods, thus enabling the verification of PL requirement models much larger than before. Our experiments show that the IMC method performs consistently better than the IC3 one, a phenomena that is different from what is reported in the literature, and would be interesting to investigate.

In our current implementations, only a single property can be verified in one run. This seems inherent in the algorithms, as the model is modified during the verification (to filter out violating products), and thus cannot be used to model-check other properties. One way to deal with this is to modify the *property* rather the model. Experimenting with this idea is left for future work.

REFERENCES

[1] Software Engineering Institute, "Software product lines: Overview [online web page]," (2013, Oct. 1), http://www.sei.cmu.edu/productlines/.

[2] B. W. Boehm, *Software Engineering Economics*. Prentice Hall PTR, 1981.

[3] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis strategies for software product lines," 2012, technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany.

[4] L. Aversano, M. Di Penta, and I. Baxter, "Handling preprocessor-conditioned declarations," in *SCAM*, 2002, pp. 83–92.

[5] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *In TOSEM*, vol. 21, no. 3, pp. 14:1–14:39, July 2012.

[6] S. Apel, W. Scholz, C. Lengauer, and C. Kastner, "Detecting dependences and interactions in feature-oriented design," in *ISSRE*, 2010, pp. 161–170.

[7] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *GPCE*, 2006, pp. 211–220.

[8] F. Heidenreich, "Towards systematic ensuring well-formedness of software product lines," in *FOSD*, 2009, pp. 69–74.

[9] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay, "Simulation-based abstractions for software product-line model checking," in *ICSE*, 2012, pp. 672–682.

[10] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE*. ACM, 2011, pp. 321–330.

[11] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE (1)*, 2010, pp. 335–344.

[12] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and model checking software product lines," in *FMOODS*, 2008, pp. 113–131.

[13] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE*, 2009, pp. 269–280.

[14] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, "Family-based deductive verification of software product lines," in *GPCE*, 2012, pp. 11–20.

[15] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *LICS*, 1987, pp. 54–64.

[17] C. L. Heitmeyer and R. D. Jeffords, "The SCR tabular notation: A formal foundation," Naval Research Lab, Tech. Rep. NLR/MR/5546-03-8678, 2003, nLR/MR/5546-03-8678.

[18] R. Milner, *Communication and Concurrency*. New York: Prentice Hall, 1989.

[19] C. A. R. Hoare, *Communicating Sequential Processes*. UK: Prentice Hall, 1985.

[20] ISO8807, "LOTOS - a formal description technique based on the temporal ordering of observational behaviour," ISO, Tech. Rep., 1988.

[21] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," vol. 20, no. 9, pp. 684–707, September 1994.

[22] G. Hamon, "A denotational semantics for stateflow," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05, 2005, pp. 164–172.

[23] P. Shaker, J. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *IEEE International Requirements Engineering Conference (RE)*, Sept 2012, pp. 151–160.

[24] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop on Logics of Programs*, ser. LNCS 131. Springer-Verlag, 1981, pp. 52–71.

[25] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[26] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model checker," *In STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[27] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1069–1089, 2013.

[28] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *In TOPLAS*, vol. 8, no. 2, pp. 244–263, 1986.

[29] R. Bryant, "Graph-based algorithms for boolean function manipulation," in *IEEE Transactions on Computers*, vol. C-35(8), 1986, pp. 677–691.

[30] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[31] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

[32] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV*, 2003, pp. 1–13.

[33] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, 2000, pp. 108–125.

[34] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of sat-based model checking techniques in an industrial environment," in *CHARME*, 2005, pp. 254–268.

[35] A. R. Bradley, "IC3 and beyond: Incremental, inductive verification," in *CAV*, 2012, p. 4.

[36] ——, "Understanding IC3," in *SAT*, 2012, pp. 1–14.

[37] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[38] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[39] M. Plath and M. Ryan, "Feature integration using a feature construct," *In Science of Computer Programming*, vol. 41, no. 1, pp. 53–84, 2001.

[40] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.

[41] E. Morreale, "Recursive operators for prime implicant and irredundant normal form determination," *In IEEE Trans. Comput.*, vol. 19, no. 6, pp. 504–509, 1970.

[42] A. Biere, "The AIGER And-Inverter Graph (AIG) Format," http://fmv.jku.at/aiger/.

[43] I. Beer, S. Ben-David, and A. Landver, "On-the-fly model checking of RCTL formulas," in *CAV*, ser. LNCS 1427. Springer-Verlag, 1998, pp. 184–194.

[44] M. Maidl, "The common fragment of CTL and LTL," in *FOCS*, 2000, pp. 643–652.

[45] S. Ben-David, D. Fisman, and S. Ruah, "Automata construction for regular expressions in model checking," June 2004, iBM research report H-0229.

[46] P. Shaker, J. M. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *RE*, 2012, pp. 151–160.

[47] M. Berry, "Proving properties of the lift system," 1996, master?s Thesis, School of Computer Science, University of Birmingham.

[48] A. Classen, "Feature Transition System," https://projects.info.unamur.be/fts/implementations/nusmv-extension/.

[49] P. Shaker, "A feature-oriented modelling language and a feature-interaction taxonomy for product-line requirements," Ph.D. dissertation, Waterloo, ON, Canada, 2013.

[50] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking software product lines with SNIP," *In STTT*, vol. 14, no. 5, pp. 589–612, 2012.

[51] P. Asirelli, M. H. T. Beek, A. Fantechi, and S. Gnesi, "A logical framework to deal with variability," in *IFM*, 2010, pp. 43–58.

[52] A. Fantechi and S. Gnesi, "Formal modeling for product families engineering," in *SPLC*, 2008, pp. 193–202.

[53] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 165–168.