Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Efficient Software Product-line Model Checking using Induction and a SAT Solver

Fei HE (✉)[1,2,3], Yuan GAO [1,2,3], Liangze YIN [4]

1 Tsinghua National Laboratory for Information Science and Technology (TNList)
2 Key Laboratory for Information System Security, Ministry of Education
3 School of Software, Tsinghua University, Beijing, China, 100084
4 Department of Computer Science and Technology, National University of Defense Technology, Changsha, China, 410073

Abstract Software product line (SPL) engineering is increasingly being adopted in safety-critical systems. It is highly desirable to rigorously show that these systems are designed correctly. However, formal analysis for SPLs is more difficult than for single systems because an SPL may contain a large number of individual systems. In this paper, we propose an efficient model-checking technique for SPLs using induction and a SAT (Boolean satisfiability problem) solver. We show how an induction-based verification method can be adapted to the SPLs, with the help of a SAT solver. To combat the state space explosion problem, a novel technique that exploits the distinguishing characteristics of SPLs, called feature cube enlargement, is proposed to reduce the verification efforts. The incremental SAT mechanism is applied to further improve the efficiency. The correctness of our technique is proved. Experimental results show dramatic improvement of our technique over the existing binary decision diagram (BDD)-based techniques.

Keywords Software product line, model checking, satisfiability

1 Introduction

Software product line engineering (SPLE) is a paradigm for the development of software product lines (SPLs) [1, 2]. An SPL [2] is a set of similar software systems (called *products*) that rely on a common code base. In general, an SPL promotes strategic, well-managed software reuse. Using SPLs helps in creating a variety of related systems efficiently with a short lead time.

E-mail: hefei@tsinghua.edu.cn

The concept of a *feature* is at the heart of SPLE. A feature is a packet of functionalities optional to the system. For example, the software packages loaded in mobile phones that belong to the same brand form an SPL. The base functionalities of these mobile phones (such as the call subsystem and the short message subsystem) are realized as the code base, whereas other functionalities (such as support for GPS, a high-resolution screen, and camera) are realized as independent features. The software loaded in a mobile phone is a *product*. A product can be considered as a combination of the code base and a set of features. An SPL with n features yields up to 2^n products.

SPLE is increasingly being adopted in safety-critical systems [3–9]. It is highly desirable to rigorously show that these systems are designed correctly. Model checking is one of the most successful approaches in formal verification [10]. For model checking in practice, one has to deal with the state space explosion problem. Note that an SPL can yield an exponential number of products in the number of features. To avoid verifying these products one by one, a single model that simulates the behaviors of all products, called the *metaproduct model*, is often constructed. The model checking is then performed on this metaproduct model [3]. Apparently, the state space explosion problem is worse for analysis in the metaproduct model than in a single product.

Model checking of SPLs has recently been studied by many researchers [11–18]. Most of these studies used explicit algorithms that enumerate system states *one by one*. For realistic designs, the size of the metaproduct model may be very large [19] and the explicit-state analysis can easily exceed the resource limits. Classen et al. proposed a binary decision diagram (BDD)-based model checking algorithm for SPLs [12, 20]. With the BDD techniques, the state space is explored more efficiently. However, for larger systems the BDDs generated during model checking can still become too

large to be handled by currently available computers [21].

SAT-based model checking [22, 23] is another symbolic verification technique, which outperforms the BDD-based techniques in many industrial cases [21]. There have been some attempts at applying SAT procedures to the verification of SPLs. The tool ProVeLines [18] employs SAT procedures. However, these solvers are only used for checking the satisfiability of feature expressions (i.e. propositional formulas over the features). The verification algorithms [11, 18, 24] implemented in this tool are essentially based on explicit-state enumeration. Another tool, SPLVerifier [25, 26], uses off-the-shelf model checkers (including a SAT-based one, CBMC [27]) to verify product-line programs, where SPL programs are treated as ordinary programs and verified by classical model checkers. However, the characteristics of SPLs are not explored in their model checking algorithm.

Given the large number of products contained in an SPL, applying the off-the-shelf verification techniques directly may lead to severe state space explosion. A dedicated model-checking algorithm that exploits the distinguishing characteristics of SPLs to reduce the verification efforts is thus required. In this paper, we propose an efficient model-checking technique using induction and a SAT solver. Recall that the SAT-based model checking technique relies on the SAT solver to verify the properties. A SAT solver returns one solution (a counterexample) during one call. With this solution, one can only conclude that one product violates the property. This is inefficient because an SPL with m failing products needs m calls of the SAT solver, whereas m may be exponential in the number of features. We propose the *feature cube enlargement* technique that exploits the knowledge about features and extends one product to a set of products. The times of calling the SAT solver (and thus the verification efforts) are greatly reduced. Then together with the temporal induction [28] technique, an unbounded model-checking algorithm for SPLs is proposed.

We also note that the sequence of SAT instances produced by model checking for increasing search length are made up of problems that are highly correlated, the information learned from former problems can help in solving the latter problems. The incremental SAT technique [29, 30] can thus be applied, which also leads a significant reduction to the verification time.

We implemented a prototype of our techniques on top of NuSMV. It takes as inputs an fSMV model and a featured safety property. Our algorithm can not only find the set of products violating the property, but also prove that the remaining products all satisfy the property. We compare our approach with the existing BDD-based symbolic model checker presented in [12]. Experimental results demonstrate the dramatic improvement of our approach.

Our main contributions are summarized as follows.

- We propose a dedicated and efficient model-checking technique for SPLs using induction and a SAT solver. The distinguishing characteristics of SPLs are exploited to reduce the verification efforts. The correctness of this technique is established.
- A technique that exploits the distinguishing characteristics of SPLs, called feature cube enlargement, is proposed to reduce the verification efforts. We also show that the incremental SAT mechanism can be integrated with our technique to further improve the efficiency.
- We implement a prototype of our technique. Experimental results demonstrate the promising performance of our approach.

The remainder of this paper is organized as follows. In Section 2, the formal preliminaries and background on SAT-based model checking are introduced. Section 3 presents our transition system for SPLs. Section 4 describes our SAT-based unbounded model checking algorithms for SPLs. Section 5 reports the experimental results on a parameterized example. Section 6 provides a discussion of the related work. Finally, Section 7 concludes the paper.

2 Background

2.1 Formal Preliminaries

Let $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$ be the *Boolean domain* with the *truth values* false and true, and $\mathbf{x}$ a finite set of *Boolean variables*. Define $\mathbf{x}' = \{x' : x \in \mathbf{x}\}$. A *literal* is a Boolean variable $x_i \in \mathbf{x}$, or its negation $\neg x_i$. A *clause* is a disjunction of a set of literals. If a clause contains one literal, it is called a *unit clause*. The negation of a clause is a *cube* (a conjunction of literals). A *conjunctive normal form* (CNF) formula is a conjunction of a set of clauses. In the remainder of this paper, both a clause and a cube are regarded as a *set* of literals, a CNF formula is regarded as a *set* of clauses, where disjunction and conjunction of these sets are implicit from the context. All set operations are applicable to clauses, cubes, and CNFs.

A *valuation* s over $\mathbf{x}$ is a mapping from $\mathbf{x}$ to $\mathbb{B}$. A valuation is *partial* if its domain is a subset of $\mathbf{x}$. We write $Val_{\mathbf{x}}$ for the set of valuations over $\mathbf{x}$. The *projection* $s|_{\mathbf{y}}$ of a valuation s on $\mathbf{y} \subseteq \mathbf{x}$ is a partial valuation such that $s|_{\mathbf{y}}(y) = s(y)$ for $y \in \mathbf{y}$. A (partial) valuation is often described as a cube. For example, the valuation $\{x_0 = 1, x_1 = 0\}$ can be represented as a cube $x_0 \wedge \neg x_1$. In the remainder of this paper, we do not distinguish between a valuation and a cube; both are referred to as s.

A *predicate* $\phi(\mathbf{x})$ (or ϕ when the variables $\mathbf{x}$ are clear from the context) is a Boolean function over $\mathbf{x}$. A clause, a cube, and a CNF are all a predicate. For a predicate ϕ and a (partial) valuation s, s *satisfies* ϕ (written $s \models \phi$) if ϕ evaluates to true by assigning $s(x)$ to $x \in \mathbf{x}$. Similarly, a pair of (partial) valuations $(s, t) \in Val_{\mathbf{x}} \times Val_{\mathbf{x}}$ *satisfies* a predicate $\psi(\mathbf{x}, \mathbf{x}')$ (written $(s, t) \models \psi$) if ψ evaluates to true by assigning $s(x)$ and $t(x)$ to $x \in \mathbf{x}$ and $x' \in \mathbf{x}'$, respectively. We write $[\![\phi]\!]$ for the set of satisfying valuations for the predicate ϕ, namely, $[\![\phi]\!] = \{s \in Val_{\mathbf{x}} : s \models \phi\}$. A predicate ϕ_1 *implies* another predicate ϕ_2, written $\phi_1 \Rightarrow \phi_2$, if every satisfying assignment

of ϕ_1 satisfies ϕ_2. A predicate ϕ is a *tautology* (written $\models \phi$) if $s \models \phi$ for every valuation $s \in Val_{\mathbf{x}}$.

Definition 1. *A* transition system *is a 4-tuple* $M = (\mathbf{v}, \mathbf{x}, I, T)$*, where:*

- $\mathbf{v}$ *is a finite set of* frozen *variables;*
- $\mathbf{x}$ *is a finite set of* state *variables;*
- $I(\mathbf{v}, \mathbf{x})$ *is an* initial predicate *over* $\mathbf{v}$ *and* $\mathbf{x}$*; and*
- $T(\mathbf{v}, \mathbf{x}, \mathbf{x}')$ *is a* transition predicate *over* $\mathbf{v}$, $\mathbf{x}$ *and* $\mathbf{x}'$*, where* $\mathbf{x}'$ *is the next-state copy of* $\mathbf{x}$.

In the above definition, we differentiate the frozen variables from other variables. A frozen variable is one whose value does not change on any transition of the system. As a result, there is no next-state copy of the frozen variables in the transition predicate.

An $\mathbf{x}$-*state* (or a $\mathbf{v}$-*state*) of M is a valuation over $\mathbf{x}$ (or $\mathbf{v}$). Let $p \in Val_{\mathbf{v}}$ be a $\mathbf{v}$-state. An $\mathbf{x}$-state s is *p-initial* if $s \models I(p, \mathbf{x})$. A pair (s, t) of $\mathbf{x}$-states is a *p-transition* if $(s, t) \models T(p, \mathbf{x}, \mathbf{x}')$. A *p-trace* of M is a sequence $\sigma = [s_0, s_1, \ldots, s_n]$ of $\mathbf{x}$-states such that $I(p, s_0)$ holds and $T(p, s_i, s_{i+1})$ holds for $i = 0, \ldots, n-1$. We write $Tr(p, M)$ for the set of p-traces in M, and $Tr(M)$ for the set of all traces in M. Obviously,

$$Tr(M) = \bigcup_{p \in Val_{\mathbf{v}}} Tr(p, M).$$

An $\mathbf{x}$-state s is *p-reachable* in M if there exists a p-trace σ ending in s. For any predicate π over $\mathbf{x}$, a sequence $\sigma = [s_0, s_1, \ldots, s_n]$ of x-states *satisfies* π (written $\sigma \models \pi$) if $s_i \models \pi$ for $i = 0, \ldots, n$. We say M *satisfies* π with respect to p (written $(p, M) \models \pi$) if $\sigma \models \pi$ for any p-trace σ in M. We say M *satisfies* π (written $M \models \pi$), if $\sigma \models \pi$ for any trace σ in M.

Throughout the paper, we use p and q to represent $\mathbf{v}$-states, and s and t to represent $\mathbf{x}$-states. We use ϕ to represent a predicate over $\mathbf{v} \cup \mathbf{x}$, λ and μ to represent predicates over $\mathbf{v}$, and π a predicate over $\mathbf{x}$.

2.2 SAT-based Bounded Model Checking

Given a transition system M and an invariant property (a predicate) π, the essential idea of *bounded model checking* [22] is to search for counterexamples to π of bounded length in $Tr(M)$. Let $\mathbf{x}^i = \{x^i : x \in \mathbf{x}\}$ be the ith copy of $\mathbf{x}$, for $0 \le i \le k$. The existence of a counterexample to π of length k can be formulated [28] as

$$bmc_k = I(\mathbf{v}, \mathbf{x}^0) \wedge \bigwedge_{i=0}^{k-1} \Big(T(\mathbf{v}, \mathbf{x}^i, \mathbf{x}^{i+1}) \wedge \pi(\mathbf{x}^i)\Big) \wedge \neg\pi(\mathbf{x}^k) \quad (1)$$

Note that in above formulation, we assume all shorter bmc_j ($j < k$) have been proved already. bmc_k is usually solved by a conventional SAT procedure. A satisfying solution to bmc_k gives a counterexample of length k. The unsatisfiability of bmc_k makes the algorithm increase the search length by 1 and check the satisfiability of bmc_{k+1}.

Bounded model checking is essentially incomplete. To verify a given property, one has to prove the unsatisfiability of bmc_k for $k = 0$ up to the *completeness threshold* (CT) [31]. The CT is usually very hard to compute [21], bounded model checking is thus considered as a bug-finding method.

2.3 SAT-based Temporal Induction

As an improvement to the bounded model checking, *SAT-based unbounded model checking* [32–34] finds counterexamples of failed properties, as well as proving the correctness of satisfied properties, with no need to compute the CT.

One popular technique is based on the temporal induction [28, 33]. An inductive proof for verifying a property π consists of two parts: the *base case* and the *induction step*. Proving the base case on length k is equivalent to checking the unsatisfiability of bmc_k. The proof for the induction step on length k is formulated [28] as

$$induct_k = \left(\bigwedge_{i=0}^{k} \Big(T(\mathbf{v}, \mathbf{x}^i, \mathbf{x}^{i+1}) \wedge \pi(\mathbf{x}^i)\Big)\right) \wedge \neg\pi(\mathbf{x}^{k+1}) \wedge \left(\bigwedge_{0 \le i < j \le k} (\mathbf{x}^i \neq \mathbf{x}^j)\right) \quad (2)$$

A satisfiable solution to $induct_k$ gives a state-unique trace of length $k+1$, consisting of k consecutive π-satisfying states and a following π-failing state. Note that a model may contain loops and thus infinite traces. The state uniqueness requires all states on a trace are unique. This restriction makes this method complete [28, 33] in the sense that there is always a length (by looking at the finite number of states in the model) for which the induction step is provable.

The unsatisfiability of bmc_k indicates that there exists no trace of length k violating the property, assuming the property holds the first $k-1$ states. The unsatisfiability of $induct_k$ indicates that following k consecutive π-satisfying states, there exists no next state violating the property. If both bmc_k and $induct_k$ are unsatisfiable, by induction on the length k, the property holds for all traces of any length [28].

The base case of this algorithm can be understood as a bug-hunting procedure. If bmc_k is satisfiable for any k, the algorithm terminates and reports the satisfiable solution as a counterexample.

3 Transition Systems for SPLs

In this section, we introduce *transition system for the software product line* (SPL-TS), a formalism for modeling the behaviors of an SPL. The SPL-TS conforms to the standard definition of transition systems and is thus suitable for SAT-based model checking. Note that the SPL-TS is used as the semantics model of an SPL in our approach. For the modeling language, we adopt fSMV [35].

```
MODULE main
VAR
   state : {ready, busy};
ASSIGN
   init(state) := ready;
   next(state) := case
                    state = busy : ready;
                    1: {ready, busy};
                  esac;
```

(a) Base system

```
FEATURE request
INTRODUCE
   VAR request : boolean;
CHANGE
   IF request = 1 & state = ready
   THEN IMPOSE next(state) := busy;
```

(b) Feature

Fig. 1 An fSMV example

3.1 Transitions Systems

An SPL consists of two parts: a base system and a set of features. The base system describes the basic behaviors of an SPL. A feature encapsulates a set of changes to the base system, which can be the introduction of new variables or modification of existing transitions. The base system can be composed with these features, leading to a superimposition.

In the following, we use a running example shown in Fig. 1 to illustrate our modeling formalism. This example is described in fSMV (an extension of the SMV language). For further details of the SMV and fSMV languages, the interested readers are referred to [35, 36].

The base system of the running example is shown in Fig. 1(a), which has two states: ready and busy. Initially, the system is on the ready state. If the current state is busy, the system transits to the ready state; otherwise, it either stays on the ready state, or transits to the busy state. Formally, the base system can be modeled as a transition system $M_b = (\mathbf{v}_b, \mathbf{x}_b, I_b, T_b)$, where

$$\begin{aligned}
\mathbf{v}_b &= \emptyset,\\
\mathbf{x}_b &= \{state\},\\
I_b &= (state = ready), \text{ and}\\
T_b &= (state = busy) \wedge (state' = ready)\\
&\quad \vee (state = ready) \wedge (state' = ready)\\
&\quad \vee (state = ready) \wedge (state' = busy).
\end{aligned}$$

Note that there is no frozen variable in the base system.

The **request** feature of the running example is shown in Fig. 1(b), which adds a new variable *request* to the base system, and changes the system behavior such that if the system is currently on the ready state, and a request arrives, the system transits to the busy state.

Definition 2. *A* feature *is a transformation function ξ that takes as input a transition system M_1, returns a transformed transition system M_2.*

Denote by ξ_{req} the transformation function of the **request** feature. Let $M_{req} = (\mathbf{v}_{req}, \mathbf{x}_{req}, I_{req}, t_{req})$ be the transformed transition system after applying the request feature to the base system, then

$$\begin{aligned}
\mathbf{v}_{req} &= \emptyset,\\
\mathbf{x}_{req} &= \{state, request\},\\
I_{req} &= (state = ready),
\end{aligned}$$

and

$$\begin{aligned}
T_{req} = &(state = busy) \wedge (state' = ready)\\
&\vee (state = ready) \wedge (request = 1) \wedge (state' = busy)\\
&\vee (state = ready) \wedge (request = 0) \wedge (state' = ready)\\
&\vee (state = ready) \wedge (request = 0) \wedge (state' = busy).
\end{aligned}$$

Note that the set of frozen variables is still empty.

Definition 3. *Let ξ_i be a feature, the* feature variable *with respect to ξ_i is a Boolean variable v_i, such that $v_i = 1$ if and only if the feature ξ_i is present.*

For the running example, let v be the *feature* variable indicating whether the **request** feature is present. Let $M_\zeta = (\mathbf{v}_\zeta, \mathbf{x}_\zeta, I_\zeta, T_\zeta)$, where

$$\begin{aligned}
\mathbf{v}_\zeta &= \{v\},\\
\mathbf{x}_\zeta &= \{state, request\},\\
I_\zeta &= (\neg v \wedge I_b) \vee (v \wedge I_{req}), \text{ and}\\
T_\zeta &= (\neg v \wedge T_b) \vee (v \wedge T_{req}),
\end{aligned}$$

where I_b, T_b, I_{req}, and T_{req} are defined as before. Note the difference between a feature variable and a state variable. The value of a state variable (either *state* or *request*) can be changed along a transition of the system. In contrast, the value of the feature variable v cannot be changed in the whole transition relation of the system. Thus, we define v as a *frozen* variable in M_ζ.

It can easily be concluded that the behaviors of M_ζ are the same as M_b if $v = 0$, and the same as M_{req} if $v = 1$. Thus, M_ζ is a model containing behaviors of both M_b and M_{req}.

Definition 4. *Given a base system M_b and a finite list of m features $\langle \xi_0, \xi_1, \ldots, \xi_{m-1} \rangle$, let v_i be the feature variable with respect to ξ_i, the* SPL-TS *is*

$$M = (\mathbf{v}, \mathbf{x}, I(\mathbf{v}, \mathbf{x}), T(\mathbf{v}, \mathbf{x}, \mathbf{x}')),$$

where:

- $\mathbf{v} = \{v_0, v_1, \ldots, v_{m-1}\}$ *is the finite set of feature variables;*
- $\mathbf{x}$ *is the finite set of state variables;*
- $I(\mathbf{v}, \mathbf{x})$ *is the initial predicate; and*
- $T(\mathbf{v}, \mathbf{x}, \mathbf{x}')$ *is the transition predicate.*

Note that the definition of an SPL-TS is in accordance with that of a transition system. Therefore, the definitions of a $\mathbf{v}$-state, an $\mathbf{x}$-state, and a p-trace follow those in a transition system.

Definition 5. *Let $M = (\mathbf{v}, \mathbf{x}, I, T)$ be an SPL-TS, and let $p \in Val_{\mathbf{v}}$ be a $\mathbf{v}$-state, the* projection *of M to p is a transition system $M|_p = (\emptyset, \mathbf{x}, I_p, T_p)$, where I_p and T_p are predicates obtained by assigning value $p(v)$ to $v \in \mathbf{v}$ in I and T, respectively. We call p a* product*, and $M|_p$ a* product model.

The number of products in an SPL depends on the size of $\mathbf{v}$. If $|\mathbf{v}| = m$, there are maximally 2^m products in the SPL. Note that a product model is a transition system with no frozen variables. Consider the example in Fig. 1, applying the valuation $\{v = 0\}$ and $\{v = 1\}$ to M_ζ gives the transition system M_b and M_{req}, respectively.

According to the definition of frozen variables, any trace in the product model $M|_p$ is a p-trace in M. Apparently, the traces of an SPL-TS model are the union of all traces in all product models, i.e.

$$Tr(M) = \bigcup_{p \in Val_{\mathbf{v}}} Tr(M|_p).$$

3.2 Featured Safety Properties

In this paper, we focus on model checking of featured safety properties [12] on SPL-TSs.

Definition 6. *A* featured safety property *is of the form $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$, where $\lambda(\mathbf{v})$ is a predicate over $\mathbf{v}$ and $\pi(\mathbf{x})$ is a predicate over $\mathbf{x}$.*

In a featured safety property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$, $\lambda(\mathbf{v})$ characterizes the set of products needed to be checked, and $\pi(\mathbf{x})$ characterizes an invariant that should hold on all reachable $\mathbf{x}$-states.

To check the behaviors of valid products only, let ϕ_{FD} be the propositional formula representing the set of valid products in a feature model [37–39]; we simply conjoin ϕ_{FD} to λ, and ask to verify $\lambda \wedge \phi_{FD} : \pi$ on the model.

Definition 7. *The semantics of a featured safety property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$ with respect to an SPL-TS model M is*

$$M \models \lambda(\mathbf{v}) : \pi(\mathbf{x}) \iff \forall p \in [\![\lambda(\mathbf{v})]\!],\ M|_p \models \pi(\mathbf{x})$$

Given a product p, if $M|_p \models \pi$, we say p is *π-satisfying*; otherwise, we say p is *π-dissatisfying*.

Definition 8. *Let M be an SPL-TS and let $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$ be a featured safety property, then the model checking problem is to identify a predicate $\mu(v)$, such that:*

1. $\forall p \in [\![\mu]\!].\ M|_p \models \pi$; *and*
2. $\forall p \in [\![\lambda]\!] \setminus [\![\mu]\!].\ M|_p \not\models \pi$.

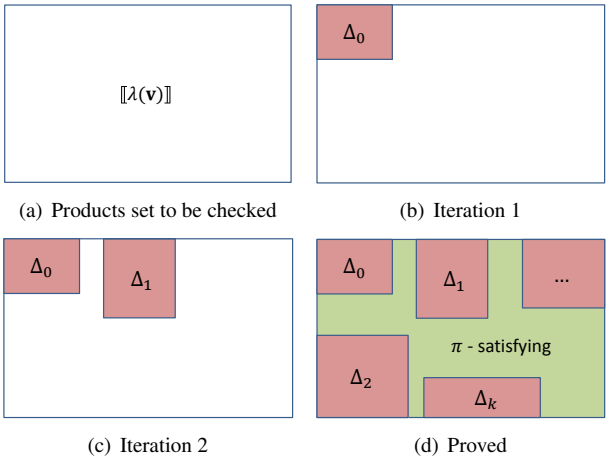
Note that with the technique in [40], any liveness property or fairness property can be handled as a safety property. Our techniques can be extended to the verification of more general properties in the same way.

4 Model Checking SPL-TSs

Note that an SPL yields a number of products, and the model checking of SPL-TSs (Definition 8) asks for a verification result for every product; the traditional model-checking algorithms are therefore not applicable. In this section, we introduce an SPL model-checking technique using induction and a SAT solver.

The basic idea behind our algorithms is illustrated in Fig. 2, where the blank area represents *unchecked* products, the red area represents *dissatisfying* products, and the yellow area represents *satisfying* products. The whole area is initialized to $[\![\lambda(\mathbf{v})]\!]$. We iteratively find in the blank area the dissatisfying products, until no further dissatisfying products can be found.

The technique for finding dissatisfying products is inspired by the bounded model checking. At the first iteration (Fig. 2(b)), we try to find the set of all products $\triangle_0$ that can be falsified by a counterexample of length 0 (i.e. the initial states). The products set $\triangle_0$ is then eliminated from the set $[\![\lambda(\mathbf{v})]\!]$. Then we try to prove whether all remaining products satisfy the property. If not, the algorithm proceeds to the next iteration. At the second iteration (Fig. 2(c)), we find in the remaining blank area the set of all products $\triangle_1$ that can be falsified by a counterexample of length 1. Again, the products set $\triangle_1$ is eliminated from $[\![\lambda(\mathbf{v})]\!]$. This process repeats until all products in the remaining blank area satisfy the property. Note that the set $[\![\lambda(\mathbf{v})]\!]$ is finite; therefore, this process always terminates.



(a) Products set to be checked (b) Iteration 1

(c) Iteration 2 (d) Proved

Fig. 2 Iterative bug-finding and proving process

Further, to realize an efficient model-checking algorithm for SPL-TSs, one needs to address the following issues:

1. how to exploit the SAT procedures to realize an efficient and symbolic algorithm (Section 4. 1);
2. how to exploit the knowledge about *features* to speed up the model checking of SPL-TSs (Sections 4.1 and 4.2);

Input: An SPL-TS $M = (\mathbf{v}, \mathbf{x}, I, T)$ and a featured safety property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$.
Output: A predicate $\mu(\mathbf{v})$.

```
1  k ← 0 ;                          /* search length */
2  μ ← λ ;                          /* set of products */
3  while true do
4  |  while (p, σ) ⊨ bmc_k ∧ μ do
5  |  |  μ ← μ ∧ ¬p ;               /* blocking clause */
6  |  end
7  |  if induct_k ∧ μ is UNSAT then
8  |  |  return μ;
9  |  end
10 |  k ← k + 1 ;
11 end
```

Algorithm 1: *Verify*(M, ϕ)

3. how to exploit the commonalities of an SPL to avoid redundant computations (Section 4.4).

4.1 Basic Algorithm

Algorithm 1 shows our basic algorithm for model-checking SPL-TSs. It takes as inputs an SPL-TS $M = (\mathbf{v}, \mathbf{x}, I, T)$ and a featured safety property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$, outputs a predicate that characterizes the set of π-satisfying products in $[\![\lambda]\!]$.

We use a predicate $\mu(\mathbf{v})$ to represent the set of products whose satisfiability to π remains *unknown*. The set $[\![\mu]\!]$ characterizes the blank area in Fig. 2. The predicate μ is initialized to be λ (at line 2 of Algorithm 1). When the algorithm proceeds, only products in $[\![\mu]\!]$ need to be checked for satisfiability to π. We call $[\![\mu]\!]$ the *search space of products*. If a product dissatisfies the property, it is removed from $[\![\mu]\!]$. When the algorithm proceeds, μ is iteratively strengthened, and the search space of products is reduced accordingly. The algorithm terminates when all products in $[\![\mu]\!]$ are π-satisfying, or $[\![\mu]\!]$ becomes empty. In the algorithm, we keep μ as a CNF, i.e. a set of clauses.

Let k be the bounded length, the existence of a bounded counterexample to π for any product in $[\![\mu]\!]$ is formulated as

$$bug\_finding_k = bmc_k \wedge \mu \tag{3}$$

where bmc_k is as given in (1). Note that μ in the above formula is used to limit the search of counterexamples in the products set $[\![\mu]\!]$.

If $bug\_finding_k$ is satisfiable, the SAT solver returns (at line 4 of Algorithm 1) a satisfying valuation of this formula. Denote this valuation as (p, σ), where p is a valuation over $\mathbf{v}$ (i.e. a product), and σ is a valuation over $\mathbf{x}^0 \cup \cdots \cup \mathbf{x}^k$ (i.e. a p-trace). As $(p, \sigma) \models bmc_k \wedge \mu$, the product p must dissatisfy π, and σ gives a counterexample. Thus, p needs to be excluded from $[\![\mu]\!]$. This can be done by adding the negation of p (a clause) to μ (at line 5 of Algorithm 1). The clause $\neg p$ is called a *blocking clause* [32], which blocks the product p from reconsideration in later computations.

Consider the **while** loop at line 4 of Algorithm 1. At each iteration, one blocking clause is added to μ (equivalently, one dissatisfying product is excluded from $[\![\mu]\!]$). This process repeats until $bug\_finding_k$ becomes unsatisfiable, which means there is no new product p' such that $M|_{p'}$ can be falsified by a counterexample of length k. The following lemma summarizes the **while** loop at line 4 of Algorithm 1.

Lemma 1. *Let $M = (\mathbf{v}, \mathbf{x}, I, T)$ be an SPL-TS and let $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$ be a featured safety property. At the kth iteration of the* **while** *loop at line 3, right after line 6 of Algorithm 1:*

1. *$\forall p \in ([\![\lambda]\!] \setminus [\![\mu]\!]).M|_p \not\models \pi$;*
2. *all products that can be falsified by a trace of length k (denoted as $\triangle_k$) have been excluded from $[\![\mu]\!]$.*

Proof. Let $\mu_0 = \lambda \wedge \neg p_0 \wedge \neg p_1 \wedge \cdots \wedge \neg p_l$ be the value of μ right after line 6 of Algorithm 1, where $\neg p_0, \neg p_1, \ldots, \neg p_l$ are blocking clauses. Note that each blocking clause corresponds to a π-dissatisfying product. For any $p \in [\![\lambda \wedge \neg \mu_0]\!]$, we have $p \in \{p_0, p_1, \ldots, p_l\}$, thus $M|_p \not\models \pi$. The second proposition follows from the fact that $bmc_k \wedge \mu_0$ is unsatisfiable. □

If $bug\_finding_k$ is unsatisfiable, the algorithm proceeds to prove whether all remaining products satisfy the property. This is equivalent to checking (at line 7 of Algorithm 1) the unsatisfiability of the following formula:

$$prove_k = induct_k \wedge \mu \tag{4}$$

where $induct_k$ is as given in (2).

If $prove_k$ is satisfiable, let (p, σ) be the satisfiable valuation returned by the SAT solver, where p is a product in $[\![\mu]\!]$ and σ is a p-trace. As $(p, \sigma) \models induct_k$, we cannot prove by induction of length k that $M|_p \models \pi$. The algorithm then proceeds to increase the bound k and repeats the loop at line 3. If $prove_k$ is unsatisfiable, then for any product $q \in [\![\mu]\!]$, $induct_k \wedge q$ is unsatisfiable, thus $M|_q \models \pi$. Therefore, we can safely conclude the algorithm with μ.

For proving the correctness of our algorithm, recall that the induction-based unbounded model checking is sound and complete [28].

Lemma 2 (Eén and Sörensson [28])**.** *Let $M = (\mathbf{v}, \mathbf{x}, I, T)$ be an SPL-TS and let $\pi(\mathbf{x})$ be an invariant property over $\mathbf{x}$. For any product model $M|_p$, there exists some k (called the* induction length *of $M|_p$) such that either $induct_k \wedge p$ is unsatisfiable or $bmc_j \wedge p$ is satisfiable for some $j \leq k$.*

Theorem 1. *Given an SPL M and a property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$, Verify$(M, \phi)$ always terminate.*

Proof. By Lemma 2, let k_{max} be the maximal induction length among all products. Note that $induct_k \wedge p$ is unsatisfiable implies $induct_{k_{max}} \wedge p$ is unsatisfiable, where $k \leq k_{max}$. Thus, at iteration k_{max}, either $[\![\mu]\!] = \emptyset$ or $induct_{k_{max}} \wedge p$ is unsatisfiable for all remaining products $p \in [\![\mu]\!]$. In either case, *Verify*(M, ϕ) terminates. □

Theorem 2. *Let $M = (\mathbf{v}, \mathbf{x}, I, T)$ be an SPL-TS and let $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$ be a featured safety property. If $\mu(\mathbf{v})$ is returned by Verify(M, ϕ) (Algorithm 1), then:*

1. $\forall p \in [\![\mu]\!]. M|_p \models \pi$; *and*
2. $\forall p \in [\![\lambda]\!] \setminus [\![\mu]\!]. M|_p \not\models \pi$.

Proof. We prove the first proposition by contradiction. Assume there exists a product $p \in [\![\mu]\!]$, such that $M|_p \not\models \pi$. Let k^* be the induction length of $M|_p$. By Lemma 2, there must exist some $j \leq k^*$, such that $bmc_j \wedge p$ is satisfiable. Then $\neg p$ must have been added to μ at line 5 of Algorithm 1, which contradicts $p \in [\![\mu]\!]$.

Note that μ can only be revised at line 5 of Algorithm 1. The second proposition follows directly from Lemma 1. □

If we replace the condition in line 7 of Algorithm 1 with $k >= B$, where B is a predefined threshold for the search length, we obtain a bounded model-checking algorithm for SPLs. We call this algorithm *VerifyBMC*(M, ϕ). Obviously *VerifyBMC*(M, ϕ) is sound but incomplete.

Theorem 3. *Let $M = (\mathbf{v}, \mathbf{x}, I, T)$ be an SPL-TS and let $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$ be a featured safety property. If $\mu(\mathbf{v})$ is returned by VerifyBMC(M, ϕ), then:*

1. $\forall p \in [\![\lambda]\!] \setminus [\![\mu]\!]. M|_p \not\models \pi$;
2. $\forall p \in [\![\mu]\!]$, *if* $M|_p \models \pi$ *is* UNKNOWN.

4.2 Feature Cube Enlargement

Algorithm 1 realizes a SAT-based model-checking algorithm for SPL-TSs. Looking at line 4 of Algorithm 1, each invocation of the SAT solver generates one new dissatisfying product. The dissatisfying products are actually enumerated in $[\![\mu]\!]$ one by one. Consider an SPL with m features, maximally it can yield 2^m products. In the worst case, the SAT solver at line 4 of Algorithm 1 needs be called 2^m times. This is apparently inefficient because each SAT solution has a high cost in terms of computational resources. We aim to propose a technique to find dissatisfying products *set by set*.

Let (p, σ) be the satisfying solution returned at line 4 of Algorithm 1, where p is a full valuation over v (a product) and σ is a p-trace. Our idea is to reduce p to $\bar{p}$ (a partial valuation), such that $(\bar{p}, \sigma) \models bmc_k \wedge \mu$ still holds. This reduction is possible because the formula $bmc_k \wedge \mu$ may have more than one satisfying solution. For example, let $\mathbf{v} = \{v_0, v_1, v_2\}$ and let $\bar{p} = v_0 \wedge v_1$ be a partial valuation, if we prove $(\bar{p}, \sigma) \models bmc_k \wedge \mu$, we can safely conclude that the set of following products are all π-dissatisfying:

$$v_0 \wedge v_1 \wedge v_2, \qquad v_0 \wedge v_1 \wedge \neg v_2$$

In this way, we enlarge the set of dissatisfying products.

Definition 9. *Given a valuation p over $\mathbf{v}$, a valuation σ over $\mathbf{y}$, and a predicate φ over $\mathbf{v} \cup \mathbf{y}$, such that $(p, \sigma) \models \varphi$, the* feature cube enlargement problem *is to find a reduced cube $\bar{p}$, such that:*

1. $\bar{p} \subseteq p$; *and*
2. $(\bar{p}, \sigma) \models \varphi$.

Here *EnlargeFCube*(φ, p, σ) (Algorithm 2) solves the feature cube enlargement problem. It computes a shortened cube $\bar{p} \subseteq p$ as specified in Definition 9. We assume that φ is in the CNF form, i.e. a set of clauses. For $(\bar{p}, \sigma) \models \varphi$, we need that for every clause c in φ, $(\bar{p}, \sigma) \models c$ holds, i.e. either $\bar{p} \models c$ or $\sigma \models c$.

Input: A predicate φ (in CNF form) over $\mathbf{v} \cup \mathbf{y}$, a valuation p over $\mathbf{v}$, and a valuation σ over $\mathbf{y}$, such that $(p, \sigma) \models \varphi$.
Output: A shortened cube $\bar{p}$.

```
1  p̄ ← ∅ ;                                  /* p̄ is a cube */
2  ϱ ← ∅ ;                                  /* ϱ is a CNF */
3  for c ∈ φ do                             /* c is a clause */
4  |  if σ ⊭ c then ϱ ← ϱ ∪ {c|_v} ;
5  end
6  for c̄ ∈ ϱ do
7  |  if c̄ is a unit clause then
8  |  |  ϱ ← ϱ \ c̄;
9  |  |  p̄ ← p̄ ∪ c̄;
10 |  end
11 end
12 for c̄ ∈ ϱ do
13 |  if p̄ ⊭ c̄ then
14 |  |  Let l be a literal in both c̄ and p, but not in p̄;
15 |  |  p̄ ← p̄ ∪ {l};
16 |  end
17 end
18 return p̄;
```

Algorithm 2: *EnlargeFCube*(φ, p, σ)

The main body of Algorithm 2 consists of three **for** loops. We explain them in turn in the following. An example is presented to explain our algorithm, in which $\mathbf{v} = \{v_0, v_1, v_2\}$, $\mathbf{y} = \{y_0, y_1, y_2\}$, $p = \neg v_0 \wedge \neg v_1 \wedge v_2$, $\sigma = y_0 \wedge \neg y_1 \wedge y_2$, and φ is a CNF:

$$\begin{aligned}
&\{v_0 \vee \neg v_1 \vee \neg v_2 \vee y_0 \vee y_2,\\
&\neg v_0 \vee v_1 \vee y_1 \vee \neg y_2,\\
&\quad v_0 \vee v_2 \vee \neg y_0 \vee y_1,\\
&\neg v_0 \vee \neg y_0,\\
&\neg v_1 \vee \neg v_2 \vee y_2\}
\end{aligned}$$

The first **for** loop (at line 3 of Algorithm 2) aims to preclude the satisfied clauses by σ from further consideration. Consider the example, the first and fifth clauses in φ are satisfied by σ. Trivially, they are also satisfied by $(\bar{p}, \sigma)$ for any form of $\bar{p}$. For clauses dissatisfied by σ, they must be satisfied by $\bar{p}$ (to make them be satisfied by $(\bar{p}, \sigma)$). Only projections of these clauses to $\mathbf{v}$ need to be considered. For the example, right after line 5 of Algorithm 2,

$$\varrho = \{\neg v_0 \vee v_1, v_0 \vee v_2, \neg v_0\}.$$

Lemma 3 summarizes the **for** loop at line 3 of Algorithm 2.

Lemma 3. *Let ϱ_1 be the value of ϱ right after line 5 of Algorithm 2, we have:*

1. $\forall \bar{c} \in \varrho_1$, $\bar{c}$ *is a clause over* $\mathbf{v}$*; and*
2. $\forall \bar{p} \subseteq p, (\bar{p} \models \varrho_1) \Rightarrow ((\bar{p}, \sigma) \models \varphi)$.

Proof. The first proposition holds apparently . We prove the second proposition as follows. Let $\bar{p}$ be any reduced cube of p, such that $\bar{p} \models \varrho_1$. For any clause c in φ, it falls into one of following two cases:

1. $\sigma \models c$, then trivially $(\bar{p}, \sigma) \models c$;
2. $\sigma \not\models c$, then $c|_{\mathbf{v}} \in \varrho_1$ must hold; as $\bar{p} \models \varrho_1$, thus $\bar{p} \models c|_{\mathbf{v}}$, thus $\bar{p} \models c$, therefore $(\bar{p}, \sigma) \models c$.

No matter which case, we have proved that $(\bar{p}, \sigma) \models c$; thus, the proposition holds. □

The second **for** loop (at line 6 of Algorithm 2) handles the unit clauses in ϱ. To satisfy a unit clause, its only literal must be present in $\bar{p}$. For the example, right after line 11 of Algorithm 2,

$$\varrho = \{\neg v_0 \vee v_1, v_0 \vee v_2\},$$
$$\bar{p} = \{\neg v_0\}.$$

The following lemma summarizes the **for** loop at line 6 of Algorithm 2.

Lemma 4. *Let ϱ_1 be the value of ϱ right after line 5, $\bar{p}_2$ and ϱ_2 the values of $\bar{p}$ and ϱ right after line 11 of Algorithm 2, respectively, for any cube $\bar{p} \subseteq p$, we have*

$$(\bar{p} \supseteq \bar{p}_2) \wedge (\bar{p} \models \varrho_2) \Rightarrow (\bar{p} \models \varrho_1).$$

Proof. Let $\bar{p}$ be any cube satisfying $\bar{p} \supseteq \bar{p}_2$ and $\bar{p} \models \varrho_2$. For any clause c in ϱ_1, if c is a unit clause, its literal must be in $\bar{p}_2$ (by line 9 of Algorithm 2), which means $\bar{p}_2 \models c$. As $\bar{p} \supseteq \bar{p}_2$, $\bar{p} \models c$. If c is not a unit clause, it must be kept in ϱ_2. From $\bar{p} \models \varrho_2$, we have $\bar{p} \models c$. No matter which case, $\bar{p} \models c$; thus, the proposition holds. □

The third **for** loop (at line 12 of Algorithm 2) tries to extend $\bar{p}$ to satisfy all remaining clauses in ϱ. Consider the example, if the clause taken from ϱ (at line 12 of Algorithm 2) is $\neg v_0 \vee v_1$, which is already satisfied by the current $\bar{p}$, the algorithm proceeds to the next iteration of the loop. Now the only clause can be taken from ϱ is $v_0 \vee v_2$, which is not satisfied by the current $\bar{p}$. In this case we need to choose a literal (that is, v_2) in $v_0 \vee v_2$ and add it to $\bar{p}$, such that the new $\bar{p}$ satisfies $v_0 \vee v_2$. Recall that we want $\bar{p} \subseteq p$, the chosen literal must also be present in p. The following lemma summarizes the **for** loop at line 12 of Algorithm 2.

Lemma 5. *Let $\bar{p}_2$ and ϱ_2 be the values of $\bar{p}$ and ϱ right after line 11, $\bar{p}_3$ the value of $\bar{p}$ right after line 17 of Algorithm 2, we have:*

1. *if $\bar{c} \in \varrho_2$ and $\bar{p} \not\models \bar{c}$, there exists at least one literal that satisfies the condition in line 14 of Algorithm 2;*
2. $\bar{p}_2 \subseteq \bar{p}_3 \subseteq p$, *and* $\bar{p}_3 \models \varrho_2$.

Proof. Let $\bar{c}$ be any clause in ϱ_2 that $\bar{p} \not\models \bar{c}$, all literals in $\bar{c}$ must not be in $\bar{p}$ (otherwise, $\bar{p} \models \bar{c}$). Let c be the clause in φ such that $\bar{c} = c|_{\mathbf{v}}$. Note that $(p, \sigma) \models \varphi$ and $\sigma \not\models c$; thus, $p \models c$, and so $p \models \bar{c}$. Therefore, at least one literal is shared by p and $\bar{c}$. The first proposition holds.

For the second proposition, $\bar{p}_2 \subseteq \bar{p}_3 \subseteq p$ holds apparently. Let $\bar{c}$ be any clause in ϱ_2, there must be a literal in $\bar{c}$ that is also in $\bar{p}_3$, which means $\bar{p}_3 \models \bar{c}$. Thus, $\bar{p}_3 \models \varrho_2$. □

In case that there exists more than one literal satisfying the condition in line 14 of Algorithm 2, we simply take the first satisfying literal in $\bar{c}$.

With all the above lemmas, we prove the correctness of Algorithm 2 in the following.

Theorem 4. *Let φ be a predicate over $\mathbf{v} \cup \mathbf{y}$, let p be a valuation over $\mathbf{v}$, and let σ be a valuation over $\mathbf{y}$, such that $(p, \sigma) \models \varphi$. If $\bar{p}$ is returned by EnlargeFCube(φ, p, σ) (Algorithm 2), then:*

1. $\bar{p} \subseteq p$*; and*
2. $(\bar{p}, \sigma) \models \varphi$.

Proof. By Lemma 5, $\bar{p}_2 \subseteq \bar{p} \subseteq p$ and $\bar{p} \models \varrho_2$. Then by Lemma 4, $\bar{p} \models \varrho_1$, and by Lemma 3, $(\bar{p}, \sigma) \models \varphi$. □

For the running example, the algorithm returns $\bar{p} = \neg v_0 \wedge v_2$, which is a partial valuation over $\mathbf{v}$ representing two products: $\neg v_0 \wedge v_1 \wedge v_2, \neg v_0 \wedge \neg v_1 \wedge v_2$.

4.3 Improved Algorithm

The basic verification algorithm (Algorithm 1) for SPL-TSs can be improved by the feature cube enlargement technique (Algorithm 2), as shown in Algorithm 3. The line that is different in Algorithm 3 to Algorithm 1 is emphasized by a surrounding box.

At line 5 of Algorithm 3, *EnlargeFCube*$(bmc_k \wedge \mu, p, \sigma)$ is invoked to shorten p to $\bar{p}$. Note that $[\![\bar{p}]\!]$ represents a set of products. By Theorem 4, $(\bar{p}, \sigma) \models bmc_k \wedge \mu$; Thus, all products in $[\![\bar{p}]\!]$ are π-dissatisfying. A blocking clause $\neg\bar{p}$ is then added to preclude all these dissatisfying products from reconsideration. In this way, we realize an algorithm that operates the dissatisfying products set by set.

This is advantageous in two ways: first the blocking clause for the reduced cube prunes the SAT search space drastically; second, the number of iterations required for the **while** loop at line 4 of Algorithm 3 decreases considerably.

The correctness of *Verify*$^+(M, \phi)$ can be proved in the same way as we prove the correctness of *Verify*(M, ϕ), with the additional support of Theorem 4.

If *Verify*$^+(M, \phi)$ (Algorithm 3) returns a predicate that is equivalent to λ, then for every product p in $[\![\lambda]\!]$, $M|_p \models \pi$, i.e. $M \models \phi$. Otherwise, there exists at least one product $p \in [\![\lambda]\!] \setminus [\![\mu]\!]$, such that $M|_p \not\models \pi$; thus, $M \not\models \phi$.

Input: An SPL-TS $M = (\mathbf{v}, \mathbf{x}, I, T)$ and a featured safety property $\phi = \lambda(\mathbf{v}) : \pi(\mathbf{x})$.
Output: A predicate $\mu(\mathbf{v})$.

```
1  k ← 0 ;
2  μ ← λ ;
3  while true do
4  |  while (p, σ) ⊨ bmc_k ∧ μ do
5  |  |  p̄ ← EnlargeFCube(bmc_k ∧ μ, p, σ) ;
6  |  |  μ ← μ ∧ ¬p̄ ;
7  |  end
8  |  if induct_k ∧ μ is UNSAT then
9  |  |  return μ;
10 |  end
11 |  k ← k + 1 ;
12 end
```

Algorithm 3: *Verify*$^+(M, \phi)$

4.4 Incremental SAT Solving

Many modern SAT solvers (such as MiniSAT [29] and PicoSAT [30]) support solving a series of related SAT problems by an incremental SAT interface. With the incremental SAT technique, the learned clauses in former SAT problems can be reused in solving the latter problems. The reused learned clauses can potentially greatly prune the search space and thus accelerate the search procedure. Our approach can also benefit from this technique.

For example, consider the **while** loop at line 4 of Algorithms 3, the SAT problems to be solved at the ith and $(i+1)$th iterations are

$$bmc_k \wedge \mu_i, \tag{5}$$

$$bmc_k \wedge \mu_i \wedge \neg \bar{p}_i \tag{6}$$

After solving (5), the SAT solver need not be restarted, the SAT problem (6) can be solved by adding one more clause (i.e. $\neg \bar{p}_i$) to the existing SAT instance. All conflict clauses learned in SAT solving of (5) are thus retained.

Further, if the algorithm proceeds to the next iteration of the out **while** loop (at line 3), the SAT problem at line 4 is

$$bmc_{k+1} \wedge \mu_i \wedge \neg \bar{p}_i \wedge \varphi, \tag{7}$$

where φ is a set of new blocking clauses added after solving (6). Note that bmc_k and bmc_{k+1} are structurally similar [23]. With the technique in [28], the SAT problem (7) can also be solved using the incremental interface. For the SAT problems encountered at line 8 of Algorithm 3, similar analysis follows.

In this way, all SAT instances in Algorithm 3 can be solved in an incremental way. With the incremental SAT technique, potentially huge computations can be saved across SAT problems in our algorithm.

5 Experiments

We implemented a prototype of our technique. To evaluate the performance of our approach, we compare our implementation (denoted as fUMC) with the state-of-the-art symbolic model checker for SPLs[1] (denoted as fBDD) developed by Classen et al. [12,20]. Both techniques were implemented on top of NuSMV 2.5.0 [41].

Both tools take as inputs a NuSMV model that serves as the base system, a list of features specified in fSMV, and one featured safety property. The composition tool in [12] is applied to compose the base system and features to create a new NuSMV model (the SPL-TS model)[2]. The fBDD and fUMC are then performed on the same SPL-TS model to verify the featured safety properties. All experiments were run on a machine with 3.06 GHz CPU and 4 GB RAM. The implementation of our prototype is available online[3].

Our approach is evaluated on seven examples[4].

- The *elevator* system [12, 35] consists of a cabin and a platform. The model is scalable by changing its number of floors. There are 9 features in this system and a total of 512 possible products.
- The *email* system [26,42] consists of N email clients. By changing N, this model is also scalable. There are eight features in this system. Owing to feature interactions (for example, the "Decrypt" and "Encrypt" features interact with the "Keys" feature); there are a total of 40 valid products.
- The *mine pump* system [13,43] consists of a water pump and a sensor that detects the water level. There are five features and six valid products.
- The *car wiper* system [13,43] models a rain sensor and a wiper. There are three features and eight valid products.
- The *vending machine* [43, 44] models a machine that serves drinks for people. There are three features and six valid products.
- The *pacemaker* system [45] models a pacemaker that monitors and regulates human heart beats. There are two features and four valid products.

The BDD-based model checking is highly dependent of the variable order in the BDD package [41]. Finding an optimal variable order is, in general, very time-consuming [10]. Fortunately, the precomputed variable orders for the elevator models with five and eight floors are available at [12]. We thus test fBDD with both *dynamic* (denoted as fBDD$_{dyn}$) and *static* (denoted as fBDD$_{sta}$) ordering strategies for these two models. For all other models, we test only the dynamic ordering strategy.

[1] https://projects.info.unamur.be/fts/implementations/nusmv-extension/
[2] Frozen variables are supported in NuSMV.
[3] See https://bitbucket.org/hefei/fumc
[4] Models and specifications of all these examples are available

Table 1 Results on the elevator example: time in seconds, memory in kilobytes

#Flr	Property		$fBDD_{dyn}$		$fBDD_{sta}$		fUMC			$fBDD_{dyn}$/fUMC		$fBDD_{sta}$/fUMC	
	ID	SAT	Time	Mem	Time	Mem	Time	Mem	#Steps	Time	Mem	Time	Mem
5	p1	32/480	95.23	30088	15.74	31930	0.04	5114	5	2164.4	5.9	357.8	6.2
	p2	272/240	32.32	28084	12.14	31526	0.08	5346	8	409.1	5.3	153.6	5.9
	p3	48/464	33.76	30386	5.48	31782	0.06	5210	8	602.8	5.8	97.8	6.1
	p4	48/464	16.03	22912	3.88	31652	0.06	5158	8	254.5	4.4	61.5	6.1
	p5	0/512	11.07	20700	1.21	28502	0.13	5956	13	82.6	3.5	9.1	4.8
	p6	240/272	11.97	19738	2.67	31752	0.05	5226	8	244.3	3.8	54.4	6.1
	p7	256/256	12.13	20542	1.65	31398	0.05	5078	7	224.7	4.0	30.5	6.2
8	p1	32/480	-	-	2024.51	227560	0.06	5852	5	-	-	36151.9	38.9
	p2	272/240	-	-	967.25	169774	0.11	6272	8	-	-	8793.2	27.1
	p3	48/464	2022.7	76776	192.56	64168	0.09	5986	8	22474.5	12.8	2139.6	10.7
	p4	48/464	-	-	146.8	64536	0.09	6028	8	-	-	1727.1	10.7
	p5	0/512	67.37	31156	11.98	32150	0.37	8630	19	181.1	3.6	32.2	3.7
	p6	240/272	124.71	31822	15.58	39370	0.09	6400	11	1341	5.0	167.6	6.2
	p7	256/256	62.61	31634	12.57	31984	0.12	6302	10	513.2	5.0	103.0	5.1
13	p1	32/480	-	-	N/A	N/A	0.11	7652	5	-	-	N/A	N/A
	p2	272/240	-	-	N/A	N/A	0.19	8376	8	-	-	N/A	N/A
	p3	48/464	-	-	N/A	N/A	0.14	7778	8	-	-	N/A	N/A
	p4	48/464	-	-	N/A	N/A	0.15	7782	8	-	-	N/A	N/A
	p5	0/512	-	-	N/A	N/A	1.49	17146	29	-	-	N/A	N/A
	p6	240/272	433.23	40604	N/A	N/A	0.29	10052	16	1478.6	4.0	N/A	N/A
	p7	256/256	2280.85	61498	N/A	N/A	0.38	9866	15	6082.3	6.2	N/A	N/A
20	p1	32/480	-	-	N/A	N/A	0.2	11144	5	-	-	N/A	N/A
	p2	272/240	-	-	N/A	N/A	0.37	12464	8	-	-	N/A	N/A
	p3	48/464	-	-	N/A	N/A	0.27	11316	8	-	-	N/A	N/A
	p4	48/464	-	-	N/A	N/A	0.26	11322	8	-	-	N/A	N/A
	p5	0/512	-	-	N/A	N/A	5.58	40376	43	-	-	N/A	N/A
	p6	240/272	-	-	N/A	N/A	0.97	18842	23	-	-	N/A	N/A
	p7	256/256	-	-	N/A	N/A	1.28	18382	22	-	-	N/A	N/A
30	p1	32/480	-	-	N/A	N/A	0.41	18218	5	-	-	N/A	N/A
	p2	272/240	-	-	N/A	N/A	0.67	20012	8	-	-	N/A	N/A
	p3	48/464	-	-	N/A	N/A	0.48	18240	8	-	-	N/A	N/A
	p4	48/464	-	-	N/A	N/A	0.53	18244	8	-	-	N/A	N/A
	p5	0/512	-	-	N/A	N/A	21.62	101666	63	-	-	N/A	N/A
	p6	240/272	-	-	N/A	N/A	3.33	39734	33	-	-	N/A	N/A
	p7	256/256	-	-	N/A	N/A	4.29	39000	32	-	-	N/A	N/A

The first experiment was conducted on the *elevator* system. Experimental results are listed in Table 1. In the experiments, we set the number of floors to 5, 8, 13, 20, and 30. The sizes of the SPL-TS models range from 2^{30} states for 5 floors to 2^{84} states for 30 floors. We specified seven safety properties on these models and verified them on all products. Logical formulas of these properties are listed in Table 2.

In Table 1, the first column (*#Flr*) lists the number of floors in the model, the second column (*ID*) gives the property ID, and the third column (*SAT*) shows the satisfiability result of this property on the model. The *SAT* results are given as a pair of integers that represents the number of products satisfying and violating the property, respectively. Note that the sum of these two integers gives the total number of valid products (it is 512 for this example). For each verification technique, we report its run time (in seconds) and maximum memory usages (in kilobytes). For fUMC, we also report the number of steps (*#Steps*) for giving a conclusive result. The last four columns compare fBDD (with either *dynamic* or *static* ordering) to fUMC. The symbol "-" indicates timeout (4000 s) and the symbol "N/A" indicates the variable order file is not available.

The results show dramatic improvement of our approach over fBDD. In all models and all properties, our approach succeeds in giving results in less than 22 s, whereas $fBDD_{dyn}$ failed to give results in 22 out of 35 cases. Note that the time for computing the static variable order is not included for $fBDD_{sta}$, the total time of $fBDD_{sta}$ may be much longer than that listed in Table 1. When the model size scales, both BDD-based approaches blow up quickly. Even for cases solvable for all approaches, fUMC outperforms both $fBDD_{dyn}$ and $fBDD_{sta}$ significantly. Depending on cases, the speedup is between 9.1 and 36000, and the reduction of memory usage is between 3.5 and 38.9. There are some cases in which fUMC can give results within less than 1 s, whereas $fBDD_{dyn}$

at `https://bitbucket.org/hefei/fumc`

Table 2 Properties for the elevator system

ID	Properties
p1	$((idle \wedge floor = 1) \rightarrow door = closed)$
p2	$((idle \wedge floor = 3) \rightarrow door = closed)$
p3	$((floor = 3 \wedge \neg liftBut3.pressed \wedge \neg landingBut3.pressed \wedge direction = up) \rightarrow door = closed)$
p4	$((floor = 3 \wedge \neg liftBut3.pressed \wedge \neg landingBut3.pressed) \rightarrow door = closed)$
p5	$((floor = 1 \wedge \neg idle \wedge door = closed) \rightarrow direction = up)$
p6	$((door = open \wedge TopLiftButIsReset) \rightarrow direction = down)$
p7	$((IsTopFloor \wedge TopLiftButIsReset) \rightarrow door = open)$

Table 3 Features of the email system

Features	Descriptions
Keys	grant a private key to the client
Encrypt	encrypt emails before they are sent
Decrypt	decrypt the encrypted incoming emails
Sign	write a signature on each outgoing email
Verify	verify signatures of incoming emails
AutoResp	response automatically on receiving an email
Forward	forward incoming emails to another client
AddrBook	each client have an addressbook

and fBDD$_{sta}$ need thousands of seconds.

The second experiment was conducted on the *email* system. Features of this system are described in Table 3. Among these features, there are interactions:

1. the "Decrypt" feature is present if and only if the "Encrypt" feature is present;
2. the "Verify" feature is present if and only if the "Signature" feature is present;
3. the "Key" feature is present if either the "Encrypt" or the "Signature" feature is present.

Denote by "f" + "feature name" the name of the corresponding feature variable. We use $\lambda(\mathbf{v})$ in the featured safety property $\lambda(\mathbf{v}) : \pi(\mathbf{x})$ to specify the feature interactions. For the *email* system, $\lambda(\mathbf{v})$ is

$$(fVerify \leftrightarrow fSign) \wedge (fDecrypt \leftrightarrow fEncrypt) \wedge (fSign \vee fEncrypt \rightarrow fKeys)$$

The results on the *email* system are listed in Table 4, where the first column lists the number of clients in the system. Similar phenomena as in the first experiment can be observed from this table. For all cases, our approach finishes in less than 16 s, whereas fBDD$_{dyn}$ can only produce results for the smallest model (with two clients). Even for the smallest model, the speedup of our approach is between 177 and 3400, and the reduction of memory usage is between 3.5 and 5.0.

Table 5 More results: time in seconds, memory in kilobytes

Example	Property	fBDD$_{dyn}$		fUMC	
		Time	Mem	Time	Mem
PaceMaker	2/2	0.003	3354	0.005	3750
Wiper	16/0	0.007	6796	0.007	7630
Vending	6/12	0.065	10616	0.078	13070
MinePump	64/32	0.018	10236	0.012	11514

The third experiment was conducted on other examples. The results are listed in Table 5. These examples are not parameterized and thus are not scalable. Both approaches can finish their verifications in less than 0.1 s.

The fourth experiment illustrates the impact of our optimization techniques on the effectiveness of fUMC. Let fUMC be the standard version. Denote by fUMC-c the version without cube enlargement and by fUMC-i the version without incremental SAT solving. This experiment is performed on the same models and properties as in the first experiment. All cases are solvable for each version of fUMC. On each model, we sum up the run time and memory usage in verifying all properties. The results are plotted in Fig. 3, where (a) shows the run time and (b) shows the memory usages. From the figures, we observe that both optimization techniques improve our algorithm on this example. On average, the cube enlargement technique reduces run time by about 66% and the incremental SAT solving reduces run time by about 70%. The memory usages of all versions are similar. Note that for the largest model, fUMC-i consumes the least memory. This may be because incremental SAT solving needs to clone several clauses and thus consumes more memory.

The experimental results demonstrate the very promising performance of fUMC, especially when the problem size is large. For these reasons, we make the following analysis. (1) fUMC is SAT-based. It does not suffer from the potential space explosion of BDDs. With modern SAT solvers, it can handle the model-checking problems with thousands of variables. (2) fUMC is carefully designed to exploit the knowledge about features and commonality and variability of SPLs. Both *feature cube enlargement* and *incremental SAT* improve fUMC greatly.

6 Related Work

6.1 Model Checking of SPLs

To specify the metamodel of an SPL, Classen et al. proposed the so-called featured transition system (FTS) [11, 12]. However, FTS is a formalism between the high-level modeling languages and the low-level computation models. It cannot be used directly for verification. In contrast, our SPL-TS formalism conforms to the standard definition of transition systems, and is more suitable for SAT-based model check-

Table 4 Results on the email example: time in seconds, memory in kilobytes

N	Prop	fBDD$_{dyn}$		fUMC		fBDD$_{dyn}$/fUMC	
		Time	Mem	Time	Mem	Time	Mem
2	20/20	84.5	28198	0.16	7090	528	4.0
	40/0	208.6	28848	0.12	6692	1683	4.3
	12/28	105.7	29536	0.10	6322	1067	4.7
	16/24	533.1	34508	0.16	6964	3396	5.0
	20/20	45.5	27866	0.26	7952	177	3.5
3	20/20	-	-	0.33	9310	-	-
	40/0	-	-	0.26	8628	-	-
	12/28	-	-	0.20	8138	-	-
	16/24	-	-	0.29	9264	-	-
	20/20	-	-	0.53	10988	-	-
5	20/20	-	-	0.79	17134	-	-
	40/0	-	-	0.72	15594	-	-
	12/28	-	-	0.48	14110	-	-
	16/24	-	-	0.79	17136	-	-
	20/20	-	-	1.38	20652	-	-
8	20/20	-	-	11.59	135536	-	-
	40/0	-	-	9.84	121248	-	-
	12/28	-	-	8.19	106992	-	-
	16/24	-	-	11.37	135554	-	-
	20/20	-	-	15.70	165130	-	-

ing. Gruler et al. [13] proposed a product line extension of CCS, called PL-CCS, which can also be used to depict the metaproduct model. The multi-valued modal μ-calculus is adapted as the property specification language. However, we are unaware of any implementations this technique.

Many advances have been made on explicit-state model checking of SPLs. Classen et al. [11] designed algorithms for verifying FTSs [11] against linear temporal logic properties. Their algorithms were implemented in a tool called SNIP [46]. The FTS was extended to support multi-features and numeric features in [17], and a simulation relation for FTS was proposed in [24]. A product line of verifiers that realizes a family of verification algorithms for SPLs was proposed in ProVeLines [18]. Lauenroth et al. [14] proposed the use of variable I/O automata to specify and verify the domain artifacts. Asirelli et al. [47] proposed the use Of the action-based branching-time temporal logic to express common dependencies of variability models, and realized their verification algorithms in VMC [15]. All these techniques are based on explicit-state model-checking algorithms, which are inefficient for large cases. In contrast, our algorithm is fully symbolic.

The first symbolic model-checking algorithm for SPLs is proposed in [12], which relies on BDDs to explore the state space. In their latest publication [20], Classen et al. proposed a new compositional formal semantics to the fSMV language and proved the expressiveness equivalence between fSMV and FTS. However, the verification algorithm remains the same as in [12]. As shown in our experiments, the BDDs generated during model checking may quickly blow up when the problem scales. In contrast, our algorithm can verify much larger systems by using SAT procedures.

6.2 SAT-based model checking

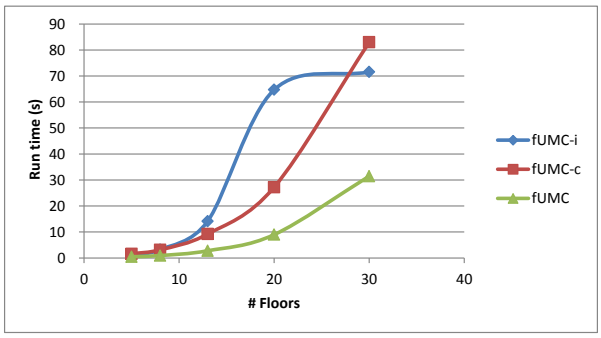
SAT-based model checking has long been an active area of research in formal verification [21]. The first SAT-based bounded model checking is first proposed in [22]. The well-known techniques for SAT-based unbounded model checking includes the interpolation-based [34] and induction-based methods [28, 33].

A similar problem of *cube enlargement* technique was studied in [32, 48, 49] for ordinary systems. In these approaches, the cube enlargement is applied to the image computation, to compute as large as possible an image of sets of states. However, these techniques do not distinguish the feature variables and state variables and thus cannot be applied to our problem. Note that we are only interested in reducing the feature cube.

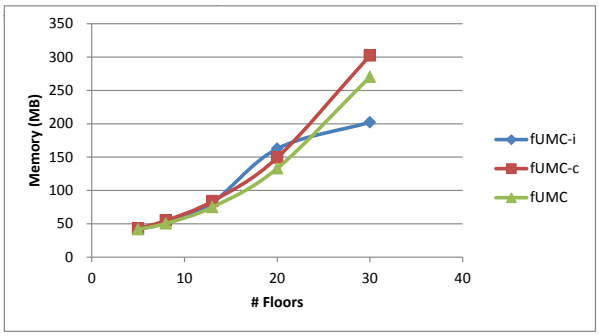
There have been some attempts at applying SAT procedures to the verification of SPLs. As we have discussed in the introduction, the tool ProVeLines [18] uses SAT solvers to check the satisfiability of feature expressions, and the tool SPLVerifier [25] employs CBMC to verify product-line programs. Even though, there lacks dedicated SAT-based model-checking algorithms for SPLs.

6.3 SAT-based analysis of feature models

The SAT techniques have been widely adopted in automated analysis of feature models [37–39, 50]. Batory proposed to formulate the semantics of feature models in Boolean formulas [38]. The feature models can be translated into Boolean



(a) Run time



(b) Memory

Fig. 3 Experiment on fUMC with different settings

formulas; then the SAT solver is employed to analyze the feature models automatically. However, all these techniques work on the feature models only. A feature model describes the relationships and dependencies of features in an SPL, which lies in the domain engineering. In contrast, we consider the verification of behavior models of SPLs. If one wants to check a safety property on valid products only, the analysis technique for feature models can be combined into our approach to decide $\lambda(\mathbf{v})$ in a featured safety property. The application of SAT techniques to this area provides further evidence of the scalability of robustness of SAT techniques.

7 Conclusions

We have considered model checking of SPLs. A symbolic model-checking algorithm for SPLs using induction and a SAT solver have been proposed. The feature cube enlargement technique has been developed to reduce the times of calling the SAT solver, and the incremental SAT-solving technique has been applied to reduce redundant computations across SAT instances. The correctness of our algorithms has been proved. Experimental results on a number of examples show dramatic improvement of our approach over existing BDD-based techniques.

ACKNOWLEDGMENTS

This work was supported in part by the Chinese National 973 Plan (2010CB328003), the NSF of China (61672310, 61272001, 60903030, 91218302), and the Chinese National Key Technology R&D Program (SQ2012BAJY4052).

References

1. Clements P, Northrop L. Software product lines: practices and patterns. Addison-Wesley, 2002
2. Pohl K, Böckle G, Van Der Linden F. Software product line engineering. Springer, 2005
3. Thüm T, Apel S, Kästner C, Schaefer I, Saake G. A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (CSUR), 2014, 47(1): 6
4. Galster M, Weyns D, Tofan D, Michalik B, Avgeriou P. Variability in software systems - a systematic literature review. IEEE Transactions on Software Engineering, 2014, 40(3)
5. Dordowsky F, Hipp W. Adopting software product line principles to manage software variants in a complex avionics system. In: Proceedings of the 13th International Software Product Line Conference. 2009, 265–274
6. Hutchesson S, McDermid J. Development of high-integrity software product lines using model transformation. In: Computer Safety, Reliability, and Security, 389–401. Springer, 2010
7. Polzer A, Kowalewski S, Botterweck G. Applying software product line techniques in model-based embedded systems engineering. In: ICSE'09 Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'09). 2009, 2–10
8. Van Ommering R. Building product populations with software components. In: Proceedings of the 24th International Conference on Software Engineering. 2002, 255–265
9. Braga R T V, Junior O T, Branco K R C, Neris L D O, Lee J. Adapting a software product line engineering process for certifying safety critical embedded systems. In: Computer Safety, Reliability, and Security, 352–363. Springer, 2012
10. Clarke E M, Grumberg O, Peled D. Model checking. MIT press, 1999
11. Classen A, Heymans P, Schobbens P Y, Legay A, Raskin J F. Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. 2010, 335–344
12. Classen A, Heymans P, Schobbens P Y, Legay A. Symbolic model checking of software product lines. In: Proceedings of the 33rd International Conference on Software Engineering. 2011, 321–330
13. Gruler A, Leucker M, Scheidemann K. Modeling and model checking software product lines. In: Formal Methods for Open Object-Based Distributed Systems, 113–131. Springer, 2008

14. Lauenroth K, Pohl K, Toehning S. Model checking of domain artifacts in product line engineering. In: Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on. 2009, 269–280
15. Beek t M H, Mazzanti F, Sulova A. VMC: A tool for product variability analysis. In: FM 2012: Formal Methods, 450–454. Springer, 2012
16. Sabouri H, Khosravi R. Efficient verification of evolving software product lines. In: Fundamentals of Software Engineering, 351–358. Springer, 2012
17. Cordy M, Schobbens P Y, Heymans P, Legay A. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: Proceedings of the 2013 International Conference on Software Engineering. 2013, 472–481
18. Cordy M, Classen A, Heymans P, Schobbens P Y, Legay A. ProVeLines: a product line of verifiers for software product lines. In: Proceedings of the 17th International Software Product Line Conference co-located workshops. 2013, 141–146
19. Tartler R, Lohmann D, Dietrich C, Egger C, Sincero J. Configuration coverage in the analysis of large-scale system software. In: Proceedings of the 6th Workshop on Programming Languages and Operating Systems. 2011, 2
20. Classen A, Cordy M, Heymans P, Legay A, Schobbens P Y. Formal semantics, modular specification, and symbolic verification of product-line behaviour. Science of Computer Programming, 2014, 80: 416–439
21. Prasad M R, Biere A, Gupta A. A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer, 2005, 7(2): 156–173
22. Biere A, Cimatti A, Clarke E M, Zhu Y. Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. 1999, 193–207
23. Shtrichman O. Tuning SAT checkers for bounded model checking. In: Proceedings of the 12th International Conference on Computer Aided Verification, CAV '00. 2000, 480–494
24. Cordy M, Classen A, Perrouin G, Schobbens P Y, Heymans P, Legay A. Simulation-based abstractions for software product-line model checking. In: Proceedings of the 2012 International Conference on Software Engineering. 2012, 672–682
25. Apel S, Speidel H, Wendler P, Rhein v A, Beyer D. Detection of feature interactions using feature-aware verification. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. 2011, 372–375
26. Apel S, Rhein A v, Wendler P, Größlinger A, Beyer D. Strategies for product-line verification: Case studies and experiments. In: Proceedings of the 2013 International Conference on Software Engineering. 2013, 482–491
27. Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 168–176. Springer, 2004
28. Eén N, Sörensson N. Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science, 2003, 89(4): 543–560
29. Eén N, Sörensson N. An extensible SAT-solver. In: Theory and applications of satisfiability testing. 2004, 502–518
30. Biere A. PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation, 2008, 4: 75–97
31. Clarke E, Kroening D, Ouaknine J, Strichman O. Completeness and complexity of bounded model checking. In: Verification, Model Checking, and Abstract Interpretation. 2004, 85–96
32. McMillan K L. Applying SAT methods in unbounded symbolic model checking. In: Computer Aided Verification. 2002, 250–264
33. Sheeran M, Singh S, Stålmarck G. Checking safety properties using induction and a SAT-solver. In: Formal Methods in Computer-Aided Design. 2000, 127–144
34. McMillan K L. Interpolation and sat-based model checking. In: Computer Aided Verification. 2003, 1–13
35. Plath M, Ryan M. Feature integration using a feature construct. Science of Computer Programming, 2001, 41(1): 53–84
36. McMillan K L. Symbolic model checking. Springer, 1993
37. Mannion M. Using first-order logic for product line model validation. In: Software Product Lines, 176–187. Springer, 2002
38. Batory D. Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Software Product Lines Conference (SPLC 2005), Rennes, France, September 26-29, 2005. 2005, 7–20
39. Mendonca M, Wasowski A, Czarnecki K. SAT-based analysis of feature models is easy. In: Proceedings of the 13th International Software Product Line Conference. 2009, 231–240
40. Biere A, Artho C, Schuppan V. Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science, 2002, 66(2): 160–177
41. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A. NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification. 2002, 359–364
42. Hall R J. Fundamental nonmodularity in electronic mail. Automated Software Engineering, 2005, 12(1): 41–79
43. Classen A. Modelling with FTS: a collection of illustrative examples. PReCISE Research Center, University of Namur, Namur, Belgium, Tech. Rep. P-CS-TR SPLMC-00000001, 2010
44. Fantechi A, Gnesi S. Formal modeling for product families engineering. In: Software Product Line Conference, 2008. SPLC'08. 12th International. 2008, 193–202
45. Ellenbogen K A, Wood M A. Cardiac pacing and ICDs. John Wiley & Sons, 2008
46. Classen A, Cordy M, Heymans P, Legay A, Schobbens P Y. Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer, 2012, 14(5): 589–612
47. Asirelli P, Beek t M H, Fantechi A, Gnesi S. A compositional framework to derive product line behavioural descriptions. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, 146–161. Springer, 2012
48. Gupta A, Yang Z, Ashar P, Gupta A. SAT-based image computation with application in reachability analysis. In: Formal Methods in

Computer-Aided Design. 2000, 391–408

49. Ganai M K, Gupta A, Ashar P. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design. 2004, 510–517
50. Schobbens P, Heymans P, Trigaux J C. Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Conference on Requirements Engineering. 2006, 139–148

Fei He received a B.S. degree from National University of Defense Technology in 2002 and a Ph.D. from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include satisfiability, model checking, compositional reasoning, and their applications to embedded systems.

Yuan Gao received a B.S. degree from Nanjing University in 2012 and a Master's degree from Tsinghua University in 2015. He is currently a software engineer in Thunisoft Information Technology Co., Ltd, Beijing, China. His research interests include satisfiability and model checking.

Liangze Yin received a B.S. degree from National University of Defense Technology in 2008 and a Ph.D. from Tsinghua University in 2014. He is currently an Assistant Professor in the School of Computer at National University of Defense Technology, Hunan, China. His research interests include satisfiability, model checking, program verification, concurrent program verification, and their applications.