Autom Softw Eng (2011) 18: 39–76
DOI 10.1007/s10515-010-0075-7

Compositional model checking of software product lines using variation point obligations

Jing Liu · Samik Basu · Robyn R. Lutz

Received: 1 March 2010 / Accepted: 12 October 2010 / Published online: 1 December 2010
© Springer Science+Business Media, LLC 2010

Abstract This paper introduces a technique for incremental and compositional model checking that allows efficient reuse of model-checking results associated with the features in a product line. As the use of product lines has increased, so has the need to verify the models used to construct the products in the product line. However, this effort is currently hampered by the difficulty of composing model-checking results for the features in a way that allows reuse for subsequent products. The contributions of this paper are to remove restrictions on how the features can be sequentially composed, to describe how to generate obligations such that all sequentially composed systems can be verified, and to show how to compositionally model check the product in the product line by reusing the variation-point obligations. The paper develops the technique and its implementation in the context of a medical-device product line.

Keywords Software product lines · Compositional model checking · Variation point · Feature

J. Liu (✉)
Rockwell Collins, Inc., Cedar Rapids, IA, USA
e-mail: Jing.Janet.Liu@gmail.com

S. Basu · R.R. Lutz
Department of Computer Science, Iowa State University, Ames, IA, USA

S. Basu
e-mail: sbasu@cs.iastate.edu

R.R. Lutz
Jet Propulsion Laboratory/Caltech, Pasadena, CA, USA
e-mail: rlutz@cs.iastate.edu

1 Introduction

Software product lines are widely used due to their advantageous reuse of shared elements, but this reuse across different products poses challenges for model checking of product lines. Especially for high-integrity product lines such as pacemakers, medical imaging systems, and avionics control systems, we would like to use model checking to verify that key properties hold in each new product. However, model-based verification of software product lines is currently hampered by the complexity of composing model checking results of the various features in a way that allows reuse when model checking new products.

In a software product line, the products all share a common set of mandatory features but are differentiated one from the other by their variable (optional and alternative) features (Weiss and Lai 1999). Each feature carries an increment of functionality for the system (Batory et al. 2006). Typically, a set of variations are selected and composed on top of the common base features to create each distinct, new product (Jacobson et al. 1997; Webber and Gomaa 2004). The locations in the features where other features can be added to construct the various products are called *variation points*.

Model checking (Queille and Sifakis 1982; Clarke et al. 1986) takes a model of a given system's design, and checks if it satisfies certain properties of the system, interpreted in terms of logic formulas. It is a powerful technique for enhancing the quality of software systems, e.g., by identifying flaws that would not have been caught otherwise (Havelund et al. 2001; Kaivola 2005). Especially for software product lines, since parts of the systems are reused in multiple products, it is important to detect flaws in those reused parts. And, since the common reused parts can present varied behaviors in different products due to different compositions with other parts, model checking can help detect subtle errors in the various compositions. Compositional model checking (Abadi and Lamport 1995; Berezin et al. 1998; Clarke et al. 1989) infers verification results of the whole system from checking each component in isolation. It can reduce the complexity of model checking product lines by amortizing the effort spent on checking the common parts.

Statement of problem Formal reasoning about each product in isolation fails to exploit the fact that all the products in a product line share common features. Similarly, many products in a product line typically share some of the variable features. Repeated verification of the same sets of features wastes resources and discourages industrial adoption of model checking for product lines. Compositional model checking saves model checking effort by allowing reuse of model checking results of common set of features.

Existing approaches to compositional model checking of features impose restrictions on how the features can be composed (e.g., the sequence in which features are added, the type of connection points at their interfaces, or the number of connections allowed). Blundell et al. (2004) show how compositional model checking can be performed when interface states (here, the variation points) are terminal states with no outgoing transitions, and the circular dependency between the features is free of loops in the underlying composed state-space. Wang (2005) extends this to allow cycles

that do not require re-exploration of non-interface states in the composed state space. Thang (2005) presents necessary conditions for preserving a desired property when extending a base, including when loops exist between the base and the extensions, but does not discuss verification results when the conditions are violated. Section 8 describes related work in more detail, and shows, with a worked-out example, how the technique introduced here relaxes restrictions on applying compositional verification in a sequential fashion.

In this section, we summarize the related works on applying compositional verification of products in product line and provide an overview of distinguishing aspects of our technique. We also present in Sect. 8.1 a detailed comparison between our technique and closely related works (Fisler and Krishnamurthi 2001, 2002; Li et al. 2002a, 2002b, 2005; Blundell et al. 2004) and discuss how our work advances the state-of-art verification techniques in a product line setting.

The pacemaker product line whose verification motivated the work described here has features whose interfaces cannot be accurately model checked with existing compositional techniques. This is a more general problem in that product lines can have variation features that connect to each other and to the common features in many different ways. There may be multiple interfaces states connecting two features, with transitions to and from states in the same feature or in other features. There may be a cycle in the composed state space with transitions that connect a common feature to a variation feature, and later connect the variation feature back to the common feature. The order in which features are added may vary or be unpredictable. It is also important that any solution accommodate product line evolution so that the features composing a product line will not have to all be known before model checking can begin.

Our solution This paper presents an incremental and compositional model checking technique that allows reuse of model checking results associated with the features in a product line.

Our compositional model checking technique generates obligations at the features' variation points such that the composition satisfies the desired property if and only if the features that are added at these variation points themselves satisfy the corresponding obligations. This is realized by re-exploring the state space of a feature only when such re-exploration is unavoidable (i.e., is necessary for the completeness of the verification process). Moreover, we use the *variation point obligations* to guide the verification of those features subsequently composed at the variation points as new products in the product line are built. The algorithms that generate the obligations incorporate optimization strategies to reduce the unnecessary model checking load.

The technique presented here is sound and complete. It removes limitations imposed by existing approaches. We show how it can be used to model check the pacemaker product line (Sect. 2) in a compositional fashion, which was not previously possible. For example, in the pacemaker, the same feature may be entered more than once in a behavioral trace (i.e., a sequence of states). Such behavior could not previously be handled.

Contributions Our approach is more general than existing work in two important ways. First, this approach allows a generic form of sequential composition of features. It supports not just pipelined composition, but also cyclic dependency in the composed state space, better mirroring the way that real-world product lines are built. Second, the variation points allow new features to be added while keeping the system open. We build the product by adding one feature at a time, in conformance with typical product-line development practice. When a product is realized, i.e., when no more features will be added, the satisfaction of the obligation at each variation point is computed, and the final verification result at the start state of the whole product is then derived. This way, the set of features no longer needs to be known ahead of time.

Product-line engineering is typically partitioned into two phases: Domain Engineering and Application Engineering (Weiss and Lai 1999). A product line is initially defined by its common and variable features in the Domain Engineering phase. The primary benefits of product-line engineering come in the Application Engineering phase when the reusable assets defined in the Domain Engineering phase are exploited to create product-line members. The technique described in this paper incorporates model checking into both phases of product-line development.

Domain engineering phase The feature behavioral models constructed for the product-line features, the properties specified for the product line, as well as the variation point obligations generated for each of the common features regarding those shared properties, are among the reusable assets created in this phase. If model checking finds any error in the models, the design, the requirements, or the properties that capture the requirements, updates are made to fix those errors, and those updates re-verified before advancing to the next phase.

Application engineering phase By reusing the product-line assets previously created, different feature compositions are specified in this phase for different products. Thus, the obligations generated in the domain-engineering phase are reused in the application-engineering phase to model check new products. The verification results, as well as the checked features, are among the product-specific assets created in this phase. As described in Sect. 7.2, these assets are maintained and updated during product-line evolution.

Incorporating model checking into both phases helps reduce the number of verification runs needed to ensure that a new product satisfies a property which holds for all (or a subset of) products in the product line. It also allows the model checking effort to be conducted in an incremental fashion. In the future, we plan to integrate the work described here with a prototype tool we have developed (Liu et al. 2008). This will allow for management of the models, properties and verification results, as well as traceability of the assets generated in the product-line setting. This, in turn, is a prerequisite for needed demonstrations of the scalability of this approach on larger product lines.

This approach provides four important advantages for model checking product lines:

1. The flexibility described above means that many more real-world systems can be model-checked. This moves model checking closer to product-line development practice.

2. The implementation stores the variation point obligations obtained for each feature during earlier model checking runs, thus enabling reuse of previous model checking results when a new product is composed. Re-verification is only performed when needed, saving time over non-compositional model checking.
3. As a product line evolves, new variation points are typically introduced. The technique described in this paper accommodates such changes by identifying obligations at these new variation points from previous obligation computations done at those points of change.
4. The technique is incremental with model checking being done one feature at a time. This allows the difficulty of checking a large state model to be reduced by decomposing it into smaller, manageable feature models.

We have implemented the technique, and demonstrate and evaluate it in the context of a medical-device product line.

The rest of the paper is organized as follows. Section 2 provides background and a motivating example. Section 3 presents preliminary information. Section 4 illustrates each step of the compositional model checking. Section 5 presents the correctness proof of the algorithms used in this technique. Section 6 provides some useful implementation details, demonstrates our technique on a simplified pacemaker product line and discusses test results. Section 7 discusses the effectiveness and applicability of this work for product line reuse. Section 8 describes related work, and compares our work with several recent efforts on formal verification of sequentially composed features. Finally, Sect. 9 offers concluding remarks.

2 Background and illustrative example

Feature-based modeling has been widely explored both for the development of software systems, e.g., Zave (1993); Batory et al. (2006) and for product lines, e.g., Kang et al. (2002). A major benefit is that features "help localize the effects of adding or making changes to units of functionality" (Ossher and Tarr 2000). Feature modeling has been especially useful for product-line development because "a feature-based model provides a basis for developing, parameterizing, and configuring reusable assets" (Kang et al. 2002).

The concept of a *feature* is here differentiated from the concept of a *component*. While both can model a functional unit or a service, a *feature* often denotes "an end-user visible characteristic of a system" (Kang et al. 1990), while a *component* does not have to be end-user visible, but is a modular unit with well-defined interfaces and can be deployed separately (Szyperski 1998). Thus, a feature is closely tied to the requirements and reflects more of the user's point of view, while a component is closely tied to the implementation, e.g., in the form of objects or collections of objects, and reflects more of the developer's point of view. Though we use both notions in the running example, we prefer feature models for checking the product line properties as it is more natural to align the verification with the product-line requirements. The product line requirements in our example are specified as CTL properties (Huth and Ryan 2004).

The work reported here was motivated by the difficulty of reusing model-checking results during the development and evolution of safety-critical product lines. Our effort is directed at enabling reuse of previous model-checking results so that system properties can be efficiently verified when a new product is built in the product line. The paper uses an example of a simplified pacemaker product line to illustrate our approach. A pacemaker (Ellenbogen and Wood 2005) is an embedded medical device designed to monitor and regulate the heart beat when it is not beating at a normal rate. It is safety-critical because some failures can damage the patient's health or even lead to loss of life (Ellenbogen and Wood 2005; Littlewood and Strigini 1993). Figure 1 shows four products in the pacemaker product line (Liu et al. 2007a, 2007b):

BasePacemaker has the basic functionality shared by all pacemakers: generating a pulse if no heart beat is detected during the sensing interval by the Base Sensor. This mode of execution is called the Inhibited Mode because the existence of a heartbeat inhibits pulse generation.

ModeTransitivePacemaker has an additional feature called ModeTransitive Extension that enables it to switch between InhibitedMode and TriggeredMode during

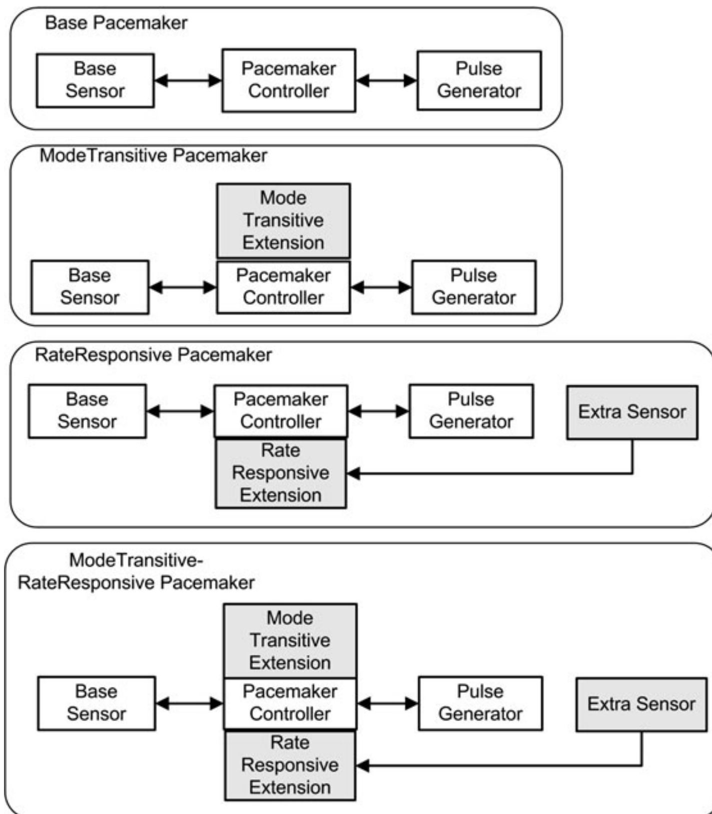


Fig. 1 Pacemaker product line overview

execution. In the TriggeredMode, a pulse follows every heartbeat to regulate the heartbeat.

RateResponsivePacemaker adds an Extra Sensor that can detect a patient's activity level (i.e., respiration rate while resting vs. while exercising). This product has an additional feature, the RateResponsive Extension, that adjusts the duration of the sensing interval (either to normal or to the upperRateLimit) according to the patient's current activity level.

ModeTransitive-RateResponsivePacemaker combines the features of the two pacemakers (ModeTransitivePacemaker and the RateResponsivePacemaker) to provide both inhibited and triggered heartbeat regulation and adaptation to the patient's activity level.

Certain properties must be shown to be true for every product in the product line in order to assure patient safety. An example of such a property is: *In the InhibitedMode, the pacemaker shall always generate a pulse when no heartbeat is detected during the normal sensing interval.*

Since verification of this property involves BasePacemaker functionality common to all the products, we would like to avoid checking the same feature again and again as each new product is built if that rechecking can be guaranteed to be unnecessary. We next describe our approach to achieving this through appropriate reuse of the results from previous model-checking runs. By generating and managing obligations at the features' variation points, the model-checking effort is aligned with the inherent variation points that a product-line development approach provides.

3 Preliminaries

In this section, we describe the preliminaries of this technique, including the formal modeling of the functional behavior of features, and the formal specification of the functional requirements.

3.1 Feature behavioral modeling

We follow standard approaches in modeling the functional behavior of product-line features using finite state machines where states represent the configurations of the functional behavior, and transitions from one state to another represent the evolution of the behavior between configurations. This is similar to the Labeled Transition System (Huth and Ryan 2004), except that we added a set of variation points. Formally, the model is defined as follows:

Definition 1 (Feature behavioral model) A feature behavioral model $\mathtt{FM} = (S, S_0, V, T, L)$ where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $V \subseteq S$ is the set of variation points, $T \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^P$ is the labeling function which associates each state $s \in S$ with the set of propositions in P that are true in that state. We will denote $(s, s') \in T$ by $s \rightarrow s'$.

In the above, $s \in V$ acts as the variation point (Pohl et al. 2005) where one FM can be plugged into another, i.e., when two FMs are *sequentially* composed, new

transitions are added from some variation points of one to states in the other. For each composition between $\mathtt{FM}_1$ and $\mathtt{FM}_2$, we use $T_c^{\mathtt{FM}_1,\mathtt{FM}_2} \subseteq V^1 \times S^2$ where V^1 is the set of variation points of $\mathtt{FM}_1$ and S^2 is the set of states in $\mathtt{FM}_2$. The relation $T_c^{\mathtt{FM}_1,\mathtt{FM}_2}$ denotes how the states in $\mathtt{FM}_2$ are connected to the variation points of $\mathtt{FM}_1$.

Also in the above model, the set of propositions P includes both data propositions and control propositions. The former reflect values on shared data, while the latter model control signals. Both need to be explicitly labelled in a state if their values are known in that state.

We define sequential composition as follows:

Definition 2 (Sequential composition) Given $\mathtt{FM}_1 = (S^1, S_0^1, V^1, T^1, L^1)$, $\mathtt{FM}_2 = (S^2, S_0^2, V^2, T^2, L^2)$, $T_c^{\mathtt{FM}_1,\mathtt{FM}_2}$, and $T_c^{\mathtt{FM}_2,\mathtt{FM}_1}$, the sequential composition $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2) = (S^1 \cup S^2, S_0^1, V^{12}, T^{12}, L^{12})$,

1. $V^{12} = V^1 \cup V^2$,
2. $T^{12} = T^1 \cup T^2 \cup T_c^{\mathtt{FM}_1,\mathtt{FM}_2} \cup T_c^{\mathtt{FM}_2,\mathtt{FM}_1}$, and
3. $L^{12}(s) = \begin{cases} L^1(s) \ \mathtt{if}\ s \in S^1 \\ L^2(s) \ \mathtt{otherwise} \end{cases}$

Observe that the above definition allows $\mathtt{FM}_1$ to be connected to $\mathtt{FM}_2$ and vice versa, resulting in possible loops between the behaviors of the two features. An example cycle in the composed state space from the pacemaker product line (Fig. 1) will be shown in Fig. 7, and a simplified version of such a loop is shown in Fig. 3. Since existing techniques for compositional model checking do not adequately handle this type of cycle in the composed state space, we seek a more generic way to perform compositional model checking such that verification can be performed in situations like this.

In Definition 2, $\mathtt{FM}_1$ is the start feature for whose start states (S_0^1) we want a given property to be satisfied. The set of variation points V^{12} of $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2)$ includes the states in V^i ($i \in \{1, 2\}$) that may have been used in the composition. They are also the states that can be used as variation points for future additions of other features.

A *closed* FM is one which does not have any variation points ($V = \emptyset$). In other words, a closed FM cannot be augmented with new features. An open FM is one whose set of variation points is non-empty. Open FM is important for a feature to be reused in the product-line setting, as it can be composed with different features in different products.

3.2 Temporal logic CTL

Properties are described using computation tree logic (CTL) (Huth and Ryan 2004). We present a brief overview of the syntax and semantics of CTL formulas. The syntax of CTL can be defined as follows:

$$\phi \rightarrow \mathtt{tt} \mid \mathtt{ff} \mid P \mid \neg\phi \mid \phi \vee \phi \mid \mathtt{EX}(\phi) \mid \mathtt{E}(\phi\ \mathtt{U}\ \phi) \mid \mathtt{EG}(\phi)$$

The semantics of the CTL formulas are given in terms of the states of finite state systems (FM) that satisfy the formulas. The propositional constant tt is satisfied in

all states while ff is not satisfied by any state. (The notion of tt and ff are specific forms for *true* and *false*, respectively.) The proposition p ($\neg p$) is satisfied by state s such that $p \in L(s)$ ($p \notin L(s)$). $\neg\varphi$ is satisfied by states where φ is not satisfied. $\varphi_1 \vee \varphi_2$ is satisfied by states that satisfy φ_1 or φ_2. $\mathtt{EX}(\varphi)$ is satisfied by a state which has at least one transition to a state that satisfies φ. $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ is satisfied by a state which has a path where φ_1 holds in every state in that path until a state satisfying φ_2 is reached. $\mathtt{EG}(\varphi)$ is satisfied by a state which has a path where every state in the path satisfies φ.

The above syntax forms the adequate set of CTL formula syntax. Some other widely used syntactic constructs such as $\mathtt{EF}(\varphi)$, $\mathtt{AX}(\varphi)$, $\mathtt{AF}(\varphi)$, $\mathtt{A}(\varphi_1\ \mathtt{U}\ \varphi_2)$, $\mathtt{AG}(\varphi)$ can be obtained from the adequate set; for example: $\mathtt{AX}(\varphi) \equiv \neg\mathtt{EX}(\neg\varphi)$ and $\mathtt{EF}(\varphi) \equiv \mathtt{E}(\mathtt{tt}\ \mathtt{U}\ \varphi)$.

A state belonging to the semantics of φ implies that the state satisfies φ, denoted by $s \models \varphi$. We say that a closed $\mathtt{FM} = (S, S_0, \emptyset, T, L)$ satisfies a CTL formula φ, denoted by $\mathtt{FM} \models \varphi$, if and only if $\forall s \in S_0 : s \models \varphi$. For a detailed discussion of CTL model checking of closed systems see Huth and Ryan (2004).

4 Detailed approach

In this section we first provide an overview of our compositional model checking technique, followed by a detailed description of each of its steps. As noted in Sect. 3, a closed FM can be verified immediately to check whether or not it satisfies a desired CTL property. However, for an open FM such as ours, satisfiability of CTL properties may depend on the behavior of the features being connected to its variation points.

Given an FM and a desired property, our solution relies on generating a set of CTL formulas as obligations for each of its variation points. A composition satisfies the desired property if and only if the added features at each variation point satisfy the corresponding obligations. We refer to these obligations as *variation point obligations*. As our definition of the feature composition allows loops between the features, such circular dependency is handled by recording in a global database, *answer set* (denoted by aSet), whether or not variation point obligations are satisfied by a composition.

Thus, checking whether a sequential composition $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m)$ satisfies a CTL formula φ amounts to checking whether all the start states of $\mathtt{FM}_1$ satisfy φ and can be compositionally resolved as follows:

Step 1 Generate the variation point obligations for satisfying φ in all the variation points of $\mathtt{FM}_i$ (initially $i = 1$). Record the variation point obligations in aSet.

Step 2 Use $T_c^{\mathtt{FM}_i, \mathtt{FM}_k}$ to identify all the features $\mathtt{FM}_k$s connected to the variation points of $\mathtt{FM}_i$: if state t_k in $\mathtt{FM}_k$ is connected to variation point s_i of $\mathtt{FM}_i$, where the variation point obligation is φ_i, iterate from Step 1 (with $i = k$) to compute the variation point obligations for each $\mathtt{FM}_k$, such that t_k satisfies φ_i. If t_k satisfies its obligation φ_i, then update the aSet entry for s_i in $\mathtt{FM}_i$.

Step 3 If the aSet cannot be further updated from computing variation point obligations, break from the iteration. Analyze aSet to identify loops between features and update aSet accordingly.

Once a product is realized (or "closed"), all its variation point obligations will be replaced with `tt` or `ff` in the `aSet`. However, we still keep a copy of those obligations in those variation points for future additions of features, as described in Sect. 6.

If the final `aSet` records that the start states of $\mathtt{FM}_1$ satisfy φ, then the composition $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m)$ satisfies φ.

In the rest of this section, we describe the generation of variation point obligations, how to update the answer set to identify cycles in the composed state space (i.e., inter-feature loops), and our algorithm for compositional model checking.

4.1 Variation point obligations

Variation point obligations are sets of formulas associated with the variation points that must be satisfied by the features connected to them. Our approach needs to compute dependency of the form $(\varphi, s) \leftarrow \psi$ (seen in Sect. 4.2), meaning that for a formula φ to be satisfied in state s, obligation ψ needs to hold.

The obligations are annotated with the boolean operators $\wedge$ and $\vee$ to handle cases where multiple features are connected to the same variation point. They are formally defined as follows:

Definition 3 (Variation point obligation) Given an $\mathtt{FM} = (S, S_0, V, T, L)$, an obligation at a variation point is a formula of the form: $\Psi \rightarrow (\phi, s, op) \mid \neg\Psi \mid \Psi \vee \Psi \mid \Psi \wedge \Psi$ where ϕ is a CTL formula, $s \in V \cup \{\epsilon\}$, $op \in \{\vee, \wedge, \bot\}$.

The variation point obligation $(\varphi, s, \vee)$ states that one of the features added at the variation point (state s) must satisfy φ; $(\varphi, s, \wedge)$ means that any new feature added at state s must satisfy φ. A variation point obligation of the form $(\varphi_1, s_1, \vee) \vee (\varphi_2, s_2, \vee)$ (resp. $(\varphi_1, s_1, \vee) \wedge (\varphi_2, s_2, \vee)$) states that $(\varphi_1, s_1, \vee)$ or (resp. and) $(\varphi_2, s_2, \vee)$ must be satisfied. Finally, $\neg(\varphi, s, \vee) \equiv (\neg\varphi, s, \wedge)$ is satisfied at the variation point if φ is not satisfied in any of the new features added at s.

We use $(\varphi, \epsilon, \bot)$ to indicate that φ is not an obligation at any variation point. We also use the following simplification rules: $(\mathtt{tt}, \epsilon, \bot) \vee \psi \equiv (\mathtt{tt}, \epsilon, \bot)$; $(\mathtt{tt}, \epsilon, \bot) \wedge \psi \equiv \psi$; $\neg(\mathtt{tt}, \epsilon, \bot) \equiv (\mathtt{ff}, \epsilon, \bot)$; $(\mathtt{ff}, \epsilon, \bot) \wedge \psi \equiv (\mathtt{ff}, \epsilon, \bot)$; $(\mathtt{ff}, \epsilon, \bot) \vee \psi \equiv \psi$; $\neg(\mathtt{ff}, \epsilon, \bot) \equiv (\mathtt{tt}, \epsilon, \bot)$; $\neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2$; $\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2$. We use $\psi, \psi_1, \psi_2, \ldots$ to denote variation point obligation formulas while using $\varphi, \varphi_1, \varphi_2, \ldots$ to represent CTL formulas.

Generation of variation point obligation follows in similar fashion to local CTL model checking (Huth and Ryan 2004) where, given a state and a CTL formula to be satisfied at that state, the algorithm proceeds by recursively exploring the reachable state space and by unfolding the CTL formula. We use the following equivalences of CTL formulas for formula unfolding: $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \mathtt{EX}(\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2))$; $\mathtt{EG}(\varphi) \equiv \varphi \wedge \mathtt{EX}(\mathtt{EG}(\varphi))$.

4.2 Step 1: Computing variation point obligations

Given an `FM` and a CTL formula φ, we define for every state s in `FM` the functions `t_Obl` and `Obl`, which generate the obligations at the variation points of `FM` required

for s to satisfy φ. The functions take five parameters: φ, the CTL formula that is required to be satisfied at the current state; s, the current state of the FM; H, the history set recording the state-formula pairs that have been visited in the recursive execution of the functions (to handle loops in the FM); $\texttt{aSet}_{in}$ and $\texttt{aSet}_{out}$ (the answer sets before and after the invocation of the function).

The answer set contains elements of the form $(\varphi, s) \leftarrow \psi$ where φ is a CTL formula and ψ is a variation point obligation. We say that satisfiability of φ at s depends on the satisfiability of ψ. Specifically,

1. $(\varphi, s) \leftarrow (\varphi', s', op')$ denotes that for s to satisfy φ, all (at least one of the) features connected via the variation point s' must satisfy φ' when op' is equal to $\wedge$ (resp. $\vee$).
2. $(\varphi, s) \leftarrow (pc, \epsilon, \bot)$ denotes that s satisfies (does not satisfy) φ when pc is a propositional constant equal to tt (resp. ff).
3. $(\varphi, s) \leftarrow \psi_1 \wedge \psi_2$ denotes that s satisfies φ if both ψ_1 and ψ_2 are satisfied. Similarly, $(\varphi, s) \leftarrow \psi_1 \vee \psi_2$ denotes that s satisfies φ if one of ψ_1 and ψ_2 is satisfied.

The aSet is necessary to handle loops across multiple features (see Step 3 in Sect. 4.4). It also allows us to reuse the previous results to remove redundant computations.

The recursive definition of the functions t_Obl and Obl is presented in Fig. 2. Each function updates the last argument, $\texttt{aSet}_{out}$, during the execution of the function; each function also returns an obligation as the result, so that the algorithm can be computed recursively and eventually get to the final form, $(\texttt{tt}, \epsilon, \bot)$, or $(\texttt{ff}, \epsilon, \bot)$.

Rule A at the top of Fig. 2 corresponds to t_Obl (top-level call) which states that a variation point obligation corresponding to state s and formula φ is equal to the result present in $\texttt{aSet}_{in}$ if t_Obl has been invoked on the same state-formula pair before. If the current invocation of t_Obl is the first-time call with the corresponding state-formula pair, then Obl is invoked, and its result ψ'' is used to update the answer set. Note that the call to Obl may update the $\texttt{aSet}_{in}$ to $\texttt{aSet}_{temp}$. If the latter already contains an entry of the form $(\varphi, s) \leftarrow \psi'$, then the mapping for (φ, s) is updated to $\psi''\ op\ \psi'(= \psi)$ where op is decided on the basis of the formula being universal (e.g. AG, AU) or existential (e.g. EG, EU).[1]

The choice of op can be explained as follows. ψ and ψ' are the variation point obligations that need to be satisfied for φ to hold at s. If φ is an universal (resp. existential) formula, the obligation at the variation point will also require all (resp. at least one) features connected to that variation point to satisfy that obligation. Accordingly, the result is obtained via conjunction or disjunction operation(s).

Rules 1–8 correspond to the Obl function. Observe that Obl invokes t_Obl to appropriately use the aSet. The first three rules in Fig. 2 state that for propositional constants and propositions, there is no obligation at the variation points; satisfiability of these types of CTL formulas can be decided at the current state s. As these function rules do not update the answer set, $\texttt{aSet}_{in}$ and $\texttt{aSet}_{out}$ remain unchanged. In Rule 5, the answer set updates are chained from one t_Obl call to the other.

[1]We are considering the adequate set containing the existential path temporal formulas EG and EU; op will be disjunction in this case.

$$\text{A.}\quad \mathtt{t_Obl}(\varphi, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \begin{cases} \psi & \text{if } (\varphi, s) \leftarrow \psi \in \mathtt{aSet}_{in}; \\ & \text{where } \mathtt{aSet}_{out} := \mathtt{aSet}_{in} \\[1ex] \psi & \text{otherwise} \\ & \text{where} \\ & \psi'' := \mathtt{Obl}(\varphi, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{temp}) \\ & \mathtt{aSet}_{out} := \begin{cases} \mathtt{aSet}_{temp}[(\varphi, s) \leftarrow \psi' / \\ (\varphi, s) \leftarrow \psi'' \; op \; \psi'] \\ \quad \text{if } (\varphi, s) \leftarrow \psi' \in \mathtt{aSet}_{temp} \\ \quad \text{where } op := \begin{cases} \wedge & \text{if } \varphi \text{ is a universal} \\ \vee & \text{otherwise} \end{cases} \\ \quad \psi = \psi'' \; op \; \psi' \\[1ex] \mathtt{aSet}_{temp} \cup \{(\varphi, s) \leftarrow \psi''\} \\ \quad \text{otherwise} \\ \quad \text{where } \psi = \psi'' \end{cases} \end{cases}$$

$$\text{1.}\quad \mathtt{Obl}(\mathtt{tt}, s, H, \mathtt{aSet}, \mathtt{aSet}) := (\mathtt{tt}, \epsilon, \bot)$$

$$\text{2.}\quad \mathtt{Obl}(\mathtt{ff}, s, H, \mathtt{aSet}, \mathtt{aSet}) := (\mathtt{ff}, \epsilon, \bot)$$

$$\text{3.}\quad \mathtt{Obl}(p, s, H, \mathtt{aSet}, \mathtt{aSet}) := \begin{cases} (\mathtt{tt}, \epsilon, \bot) & \text{if } p \in L(s) \\ (\mathtt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases}$$

$$\text{4.}\quad \mathtt{Obl}(\neg\varphi, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \neg\mathtt{t_Obl}(\varphi, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out})$$

$$\text{5.}\quad \mathtt{Obl}(\varphi_1 \vee \varphi_2, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \mathtt{t_Obl}(\varphi_1, s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{temp}) \vee \mathtt{t_Obl}(\varphi_2, s, H, \mathtt{aSet}_{temp}, \mathtt{aSet}_{out})$$

$$\text{6.}\quad \mathtt{Obl}(\mathrm{E}(\varphi_1 \,\mathrm{U}\, \varphi_2), s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \begin{cases} (\mathtt{ff}, \epsilon, \bot) & \text{if } (\mathrm{E}(\varphi_1 \,\mathrm{U}\, \varphi_2), s) \in H; \\ & \text{where} \\ & \mathtt{aSet}_{out} := \mathtt{aSet}_{in} \\[1ex] \mathtt{t_Obl}(\varphi_2 \vee (\varphi_1 \wedge \mathrm{EX}(\mathrm{E}(\varphi_1 \,\mathrm{U}\, \varphi_2))), s, \\ H \cup \{(\mathrm{E}(\varphi_1 \,\mathrm{U}\, \varphi_2), s)\}, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) \\ \quad \text{otherwise} \end{cases}$$

$$\text{7.}\quad \mathtt{Obl}(\mathrm{EG}(\varphi), s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \begin{cases} (\mathtt{tt}, \epsilon, \bot) & \text{if } (\mathrm{EG}(\varphi), s) \in H; \\ & \text{where} \\ & \mathtt{aSet}_{out} := \mathtt{aSet}_{in} \\[1ex] \mathtt{t_Obl}(\varphi \wedge \mathrm{EX}(\mathrm{EG}(\varphi))), s, \\ H \cup \{(\mathrm{EG}(\varphi), s)\}, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) \\ \quad \text{otherwise} \end{cases}$$

$$\text{8.}\quad \mathtt{Obl}(\mathrm{EX}(\varphi), s, H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) := \bigvee_{s \to s'} \mathtt{t_Obl}(\varphi, s', H, \mathtt{aSet}_{in}, \mathtt{aSet}_{out}) \vee \begin{cases} (\varphi, s, \vee) & \text{if } s \in V \\ (\mathtt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases}$$

Fig. 2 Variation point obligations

Rules 6 and 7 use the history set H to decide the satisfiability of EU and EG properties in the presence of a loop (in the same feature model). If the state-formula pair is present in the history set, this shows circular dependency in a path. Thus, for the least fixed point formula EU, the result is $(\mathtt{ff}, \epsilon, \bot)$. For the greatest fixed point formula EG, the result is $(\mathtt{tt}, \epsilon, \bot)$. On the other hand, if the state-formula pair is not present in the history set, the formula is expanded to its equivalent form, t_Obl is invoked, and the history set is updated.

Finally, Rule 8 deals with $\texttt{EX}(\varphi)$ formulas. The obligation is computed by expanding or moving to all possible next states of the current state s. There are two disjuncts in the result. The first disjunct shows that for each $s \to s'$, `t_Obl` is computed using s' and φ, and the results are OR-ed. This is because $\texttt{EX}(\varphi)$ is satisfied at s if there *exists* one next state that satisfies φ. The second disjunct states that if s is a variation point, then one of its future next states (there could be one or several), which is a state of a new feature connected to it, will have the obligation to satisfy φ.

Example Figure 3 shows three features with the behavior of each represented by a state with a self-loop. Inter-feature transitions are shown as broken lines. All states in the example are variation points, and the proposition p does not label any state. Given the CTL property φ=$\texttt{E(tt U}\ p)$ to verify over the three-feature composition $Comp_{seq}(F_1, F_2, F_3)$, we first compute the variation point obligation for F_1, shown in Fig. 4. The downward arrows in Fig. 4 show the invocation of the `t_Obl` and `Obl` functions, while the upward arrows show the updates to `aSet`. The variation point

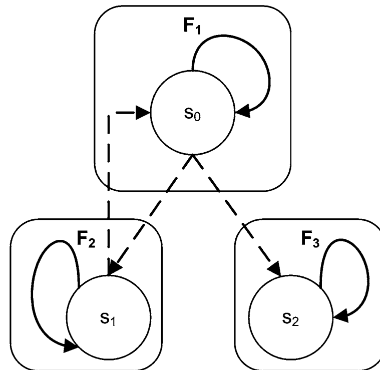


Fig. 3 Feature composition example

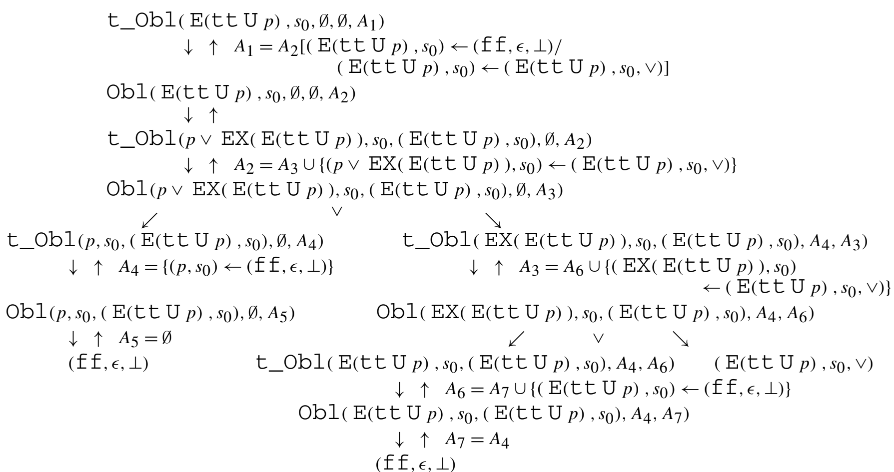


Fig. 4 Example of computing variation point obligations

Table 1 Content of each `aSet` in Fig. 4

Answer set	Evaluation
A_1	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot), (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee),$ $(\mathrm{EX}(\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p)), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee),$ $(p \vee \mathrm{EX}(\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p)), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee)\}$
A_2	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot), (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot),$ $(\mathrm{EX}(\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p)), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee),$ $(p \vee \mathrm{EX}(\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p)), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee)\}$
A_3	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot), (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot),$ $(\mathrm{EX}(\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p)), s_0) \leftarrow (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0, \vee)\}$
A_4	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot)\}$
A_5	$\{\}$
A_6	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot), (\mathrm{E}(\mathtt{tt}\ \mathrm{U}\ p), s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot)\}$
A_7	$\{(p, s_0) \leftarrow (\mathtt{ff}, \epsilon, \bot)\}$

obligations for F_2 and F_3 are computed in a similar fashion. The content of each `aSet` is presented in Table 1.

4.3 Step 2: Updating `aSet`

This step takes as input $\mathtt{aSet}_{out}$ which is provided by `t_Obl` obtained in the previous step (Fig. 2). After the computation of variation point obligation terminates for one `FM`, the input `aSet` is updated with new results $(\mathtt{tt}, \epsilon, \bot)$ and $(\mathtt{ff}, \epsilon, \bot)$ in order to incorporate information regarding whether the state s satisfies a formula:

$$\mathtt{update(aSet)} := \mathtt{aSet}[(\varphi, s) \leftarrow \psi / (\varphi, s) \leftarrow \psi[\psi_i / \psi_i']]$$
$$\text{where } \psi_i := (\varphi_i, s_i, op_i) \text{ and } s_i \rightarrow t_i \in T_c^{\mathrm{FM}_m, \mathrm{FM}_n} \text{ and}$$
$$(\varphi_i, t_i) \leftarrow (pc, \epsilon, \bot) \in \mathtt{aSet} \wedge pc \in \{\mathtt{tt}, \mathtt{ff}\} \text{ and}$$
$$\psi_i' = \begin{cases} \psi_i & \text{if } (pc = \mathtt{tt}) \wedge (op_i = \wedge) \\ \psi_i & \text{if } (pc = \mathtt{ff}) \wedge (op_i = \vee) \\ (\mathtt{tt}, \epsilon, \bot) & \text{if } (pc = \mathtt{tt}) \wedge (op_i = \vee) \\ (\mathtt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases}$$

The function states that the entry $(\varphi, s) \leftarrow \psi$ in `aSet` is updated to $(\varphi, s) \leftarrow \psi[\psi_i / \psi_i']$ (ψ_i, a subformula of ψ, is replaced by ψ_i'). ψ_i is a variation point obligation of the form (φ_i, s_i, op_i) that was computed for a $\mathtt{FM}_m$. If the state t_i of $\mathtt{FM}_n$ is connected to the variation point s_i and there exists an entry $(\varphi_i, t_i) \leftarrow (pc, \epsilon, \bot)$ ($pc \in \{\mathtt{tt}, \mathtt{ff}\}$) in the `aSet`, then we can use $(pc, \epsilon, \bot)$ to update ψ_i in ψ. For example, if $op_i = \wedge$, indicating that all next states of s_i should satisfy φ_i, then in the case $pc = \mathtt{tt}$, ψ_i remains unaltered since the satisfiability of φ_i in one next state does not

prove that φ_i is satisfied in all next states; on the other hand, if $pc = \texttt{ff}$, then it can be concluded that the variation point obligation has not been satisfied at s_i.

Example Continuing our example, after variation point obligations for F_1, F_2 and F_3 are computed, $\texttt{aSet} = \{(\varphi, s_0) \leftarrow (\varphi, s_0, \vee), (p \vee \texttt{EX}(\varphi), s_0) \leftarrow (\varphi, s_0, \vee), (p, s_0) \leftarrow (\texttt{ff}, \epsilon, \bot), (\texttt{EX}(\varphi), s_0) \leftarrow (\varphi, s_0, \vee), (\varphi, s_1) \leftarrow (\varphi, s_1, \vee), (p \vee \texttt{EX}(\varphi), s_1) \leftarrow (\varphi, s_1, \vee), (p, s_1) \leftarrow (\texttt{ff}, \epsilon, \bot), (\texttt{EX}(\varphi), s_1) \leftarrow (\varphi, s_1, \vee), (\varphi, s_2) \leftarrow (\texttt{ff}, \epsilon, \bot), (p \vee \texttt{EX}(\varphi), s_2) \leftarrow (\texttt{ff}, \epsilon, \bot), (p, s_2) \leftarrow (\texttt{ff}, \epsilon, \bot), (\texttt{EX}(\varphi), s_2) \leftarrow (\texttt{ff}, \epsilon, \bot)\}$. In the above aSet, $(\varphi, s_0) \leftarrow (\varphi, s_0, \vee)$ and there exists $s_0 \rightarrow s_2$ in $T^{F1,F3}$: $(\varphi, s_2) \leftarrow (\texttt{ff}, \epsilon, \bot)$. When performing update to the answer set, the second branch of the function is applied. This does not change the existing obligations in the aSet. On the other hand, if both $(\varphi, s_0) \leftarrow (\varphi, s_0, \vee)$ and $(\varphi, s_2) \leftarrow (\texttt{tt}, \epsilon, \bot)$ had existed in the above answer set, then $(\varphi, s_0) \leftarrow (\varphi, s_0, \vee)$ would be replaced by $(\varphi, s_0) \leftarrow (\texttt{tt}, \epsilon, \bot)$ after performing the update function.

4.4 Step 3: Resolving inter-feature loops from aSet

To summarize, once the variation point obligations have been computed for all FMs (Steps 1 and 2) and for every $(\varphi, s) \leftarrow \psi$ in aSet, each subformula (φ_i, s_i, op_i) of ψ has a corresponding $(\varphi_i, s_i) \leftarrow \psi'$ in aSet, we can conclude that no further updates to aSet can be computed.

We can now search for any chain of variation point obligations from aSet to resolve the circular dependency between features (i.e., obtain the final verification result in terms of tt or ff). An example of such a chain is the circular dependency between F_1 and F_2 in Fig. 3: $(\varphi, s_0) \leftarrow (\varphi, s_0, \vee)$ and $(\varphi, s_1) \leftarrow (\varphi, s_1, \vee)$, where $s_0 \rightarrow s_1$ and $s_1 \rightarrow s_0$. The circular dependency is resolved by applying the $\texttt{update}_F(\texttt{aSet})$ function on each element in aSet, where $\texttt{update}_F(\texttt{aSet})$ is defined as follows:

$$\texttt{update}_F(\texttt{aSet}) = \texttt{aSet}[(\varphi, s) \leftarrow \psi/(\varphi, s) \leftarrow (pc, \epsilon, \bot)]$$
$$\text{where } pc = \textsc{Interp}((\varphi, s), \emptyset)$$

Algorithm 1 computes Interp, taking as input parameters (φ, s) and the set Dep which records the elements on which the mapping result of (φ, s) depends. In Line 2, if $(\varphi, s) \leftarrow (pc, \epsilon, \bot)$, then the result does not depend on other elements, and the procedure immediately returns pc. In this case the answer has been resolved. Otherwise, if the result depends on an element in the set Dep (Line 3), a circular dependency is identified, and the return result is computed based on whether or not φ is a greatest or a least fixed point formula (Line 4). This is similar to the way we detected intra-feature loops during variation point obligation computation using the history set (see Fig. 2). If neither of the above, in Lines 6–15, the algorithm computes the dependency of results across features. Line 6 depicts the generic form of ψ in an answer set element, $(\varphi, s) \leftarrow \psi$ (according to Fig. 2), where ψ is composed of multiple sub-obligations connected by $\vee$ or $\wedge$ operators among them. Then in Line 7, for each sub-obligation ψ_i of ψ, where $\psi_i = (\varphi_i, s_i, op_{2i-1})$, t_i (the state connected to s_i) is identified and collected in the set $\texttt{Next}_i$. In the following lines (Line 10–12), Interp is recursively invoked on each element of $\texttt{Next}_i$. The intermediate result $\texttt{res}_i$, with its default value set according to op_i being $\wedge$ or $\vee$ (Line 9), is aggregated with the

result computed for previous element(s) of $\texttt{Next}_i$ (Line 11). The final result to return, $\texttt{res}$, is computed by connecting all the intermediate $\texttt{res}_i$ using the same op_{2i-2} that connects all subformula ψ_i in ψ (Line 13).

Algorithm 1 Analysis for inter-feature loops

1: **procedure** INTERP((φ, s), Dep)
2:    **if** $(\varphi, s) \leftarrow (pc, \epsilon, \perp)$ **return** pc
3:    **if** $(\varphi, s) \in \texttt{Dep}$ **then**
4:       **if** φ is gfp **return** tt **else return** ff
5:    **end if**
6:    $(\varphi, s) \leftarrow (\varphi_1, s_1, op_1) op_2 \cdots op_{2k-2} (\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet}$
7:    $\texttt{Next}_i := \bigcup_{s_i \to t_i} \{(\varphi_i, t_i) \mid s_i \to t_i \in T_c^{\mathrm{FM}_m, \mathrm{FM}_n}\}$
8:    **for** $1 \leq i \leq k$ **do**
9:       **if** $op_i = \wedge$ res_i = tt **else** $\texttt{res}_i$ = ff
10:       **for** $(\varphi_i, t_i) \in \texttt{Next}_i$ **do**
11:          $\texttt{res}_i = \texttt{res}_i\ op_i$ INTERP($(\varphi_i, t_i), \texttt{Dep} \cup \{(\varphi, s)\}$)
12:       **end for**
13:       **if** $i = 1$ $\texttt{res} = \texttt{res}_i$ **else** $\texttt{res} = \texttt{res}_{i-1}\ op_{2i-2}\ \texttt{res}_i$
14:    **end for**
15:    **return** res
16: **end procedure**

Note that the above algorithm handles the situation where a variation point s_i has no state in another feature to connect to it. In such a situation, $\texttt{Next}_i$ is empty, and $\texttt{res}_i$ has its default value set as tt or ff according to op_i. Thus, any form of variation point obligation can be updated using INTERP and eventually be resolved to the final form of $(pc, \epsilon, \perp)$.

Example We perform $\texttt{update}_F(\texttt{aSet})$ on each element in the previously-computed aSet (the sequence does not matter). For example, if the following element is picked first: $(\varphi, s_1) \leftarrow (\varphi, s_1, \vee)$, we compute INTERP$((\varphi, s_1), \emptyset)$ = INTERP$((\varphi, s_0), \{(\varphi, s_1)\})$ = INTERP$((\varphi, s_1), \{(\varphi, s_1), (\varphi, s_0)\})$ = ff. We then replace $(\varphi, s_1) \leftarrow (\varphi, s_1, \vee)$ with $(\varphi, s_1) \leftarrow (\texttt{ff}, \epsilon, \perp)$ in the aSet. Other elements of aSet are updated in a similar fashion until no change can be done to aSet. This means that all circular dependencies between these features have been resolved.

4.5 Summary: Compositional algorithm

Algorithm 2 presents the compositional algorithm introduced at the beginning of this section using the functions described above. This algorithm model checks the current product (a composition of features), reusing results from any previous model checking of those features for other products in the product line.

To summarize, given a composition $Comp_{seq}(\mathrm{FM}_1, \mathrm{FM}_2, \ldots, \mathrm{FM}_m)$ and a formula φ, the algorithm first obtains the variation point obligations for FM_1 such that all its start states can satisfy φ (Line 3–7). From Lines 8 to 15, the variation point obligations of the other features connected to the variation points of FM_1 are

Algorithm 2 Compositional model checking

```
1: procedure COMPOSE(Comp_seq(FM_1, FM_2, ..., FM_m), φ)
2:     aSet_current := ∅
3:     for each s_0 ∈ S_0^1 do
4:         t_Obl(φ, s_0, ∅, aSet_current, aSet)
5:         aSet_current := aSet
6:         if (φ, s_0) ← (ff, ϵ, ⊥) ∈ aSet_current return aSet_current
7:     end for
8:     repeat
9:         tmp := aSet_current
10:        for each (φ', s') ← (φ_1, s_1, op_1)op_2 ··· op_{2k−2}(φ_k, s_k, op_{2k−1}) ∈ aSet
               ∧ φ_i ∉ {tt, ff} ∧ s_i → t_i ∈ T_c^{FM_m,FM_n} do
11:            t_Obl(φ_i, t_i, ∅, aSet_current, aSet)
12:            aSet_current := aSet
13:            aSet_current := update(aSet_current)
14:        end for
15:    until (aSet_current = tmp)
16:    return update_F(aSet_current)
17: end procedure
```

computed. The process of computing the variation point obligation is iterated until no more updates on the aSet can be made (Line 15). At this point, the function $\mathtt{update}_F(\mathtt{aSet})$ is invoked to identify loops between features and infer results from variation point obligations represented in greatest and least fixed point formulas in the aSet. We say that $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m) \models \varphi$ when for all start states s_0 of $\mathtt{FM}_1$, $((\varphi, s_0) \leftarrow (\mathtt{tt}, \epsilon, \bot)) \in \mathtt{update}_F(\mathtt{aSet})$.

We have now performed every step of Algorithm 2 for the example in Fig. 3. Since $(\varphi, s_0) \leftarrow (ff, \epsilon, \bot) \in update_F(aSet)$, *the verification result is that* $Comp_{seq}(F_1, F_2, F_3) \not\models \varphi$.

5 Correctness proof

In this section, we present the correctness proof of the COMPOSE algorithm described in Sect. 4 and Algorithm 2.

Theorem 1 (Sound and complete) $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m) \models \varphi \Leftrightarrow \forall\, s_0 \in S_0^1$: $(\varphi, s_0) \leftarrow (\mathtt{tt}, \epsilon, \bot) \in \mathtt{COMPOSE}(Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m), \varphi)$.

Proof The function $\mathtt{COMPOSE}(Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m), \varphi)$ returns aSet of variation point obligations (ψ) computed by iterative applications of t_Obl and update followed by INTERP. While t_Obl and update are used to compute the variation point obligations of different features and to compose them, INTERP computes the result of composition once all the features have been explored appropriately, i.e., no new variation point obligations can be obtained.

Therefore, the proof of correctness of COMPOSE algorithm can be realized by proving the correctness of t_Obl, update and INTERP functions.

$$1.\ \frac{s \models^{\circ}_{H} \mathtt{tt}\ [(\mathtt{tt}, \epsilon, \bot)]}{\bullet} \qquad 2.\ \frac{s \models^{\circ}_{H} \mathtt{ff}\ [(\mathtt{ff}, \epsilon, \bot)]}{\bullet}$$

$$3.\ \frac{s \models^{\circ}_{H} p\ [(\mathtt{tt}, \epsilon, \bot)]}{\bullet}\ p \in L(s) \qquad 4.\ \frac{s \models^{\circ}_{H} p\ [(\mathtt{ff}, \epsilon, \bot)]}{\bullet}\ p \notin L(s)$$

$$5.\ \frac{s \models^{\circ}_{H} \neg\varphi\ [\neg\psi]}{s \models^{\circ}_{H} \varphi\ [\psi]} \qquad 6.\ \frac{s \models^{\circ}_{H} \varphi_1 \vee \varphi_2\ [\psi_1 \vee \psi_2]}{s \models^{\circ}_{H} \varphi_1\ [\psi_1] \quad s \models^{\circ}_{H} \varphi_2\ [\psi_2]}$$

$$7.\ \frac{s \models^{\circ}_{H} \mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)\ [(\mathtt{ff}, \epsilon, \bot)]}{\bullet}\ \{(\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2), s) \in H\}$$

$$8.\ \frac{s \models^{\circ}_{H} \mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)\ [\psi]}{s \models^{\circ}_{H\cup\{(\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2),s)\}} \varphi_2 \vee (\varphi_1 \wedge \mathtt{EX}(\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)))\ [\psi]}\{(\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2), s) \notin H\}$$

$$9.\ \frac{s \models^{\circ}_{H} \mathtt{EG}(\varphi)\ [(\mathtt{tt}, \epsilon, \bot)]}{\bullet}\ \{(\mathtt{EG}(\varphi), s) \in H\}$$

$$10.\ \frac{s \models^{\circ}_{H} \mathtt{EG}(\varphi)\ [\psi]}{s \models^{\circ}_{H\cup\{(\mathtt{EG}(\varphi),s)\}} \varphi \wedge \mathtt{EX}(\mathtt{EG}(\varphi)))\ [\psi]}\ \{(\mathtt{EG}(\varphi), s) \notin H\}$$

$$11.\ \frac{s \models^{\circ}_{H} \mathtt{EX}(\varphi)\ [\bigvee \psi_i]}{s_i \models^{\circ}_{H} \varphi\ [\psi_i]}\ \{s \to s_i, s \notin V\} \quad 12.\ \frac{s \models^{\circ}_{H} \mathtt{EX}(\varphi)\ [\bigvee \psi_i\ \vee\ (\varphi, s, \vee)]}{s_i \models^{\circ}_{H} \varphi\ [\psi_i]}\ \{s \to s_i, s \in V\}$$

Fig. 5 Tableau rules for $\models^{\circ}_{H}$

Correctness of `t_Obl` We introduce the notion of open-system verification to formalize variation point obligations. Recall that, a state s in FM satisfies CTL formula φ, denoted by $s \models \varphi$, if and only if s belongs to the semantics of φ. In the case of a FM with variation points where new features can be composed, we may not be able to infer whether $s \models \varphi$. Instead, we introduce a new predicate $\models^{\circ}_{H} \subseteq S \times \Phi \times \mathcal{H} \times \Psi$ where S is the set of states, Φ is the set of CTL formulas, $\mathcal{H}$ is the set of history sets and Ψ is the set of variation point obligation formulas. We write $s \models^{\circ}_{H} \varphi\ [\psi]$ to state that s satisfies φ in the context of a history H if and only if the interface obligation ψ is satisfiable. Following tableau-based local model checking technique (Cleaveland 1990), we present in Fig. 5 the tableau rules for $\models^{\circ}_{H}$ predicate which captures the semantics of $\models^{\circ}_{H}$. In each rule of the form $\frac{a}{a_1\ a_2\ \cdots\ a_n}$ denotes $a_1 \wedge a_2 \wedge \cdots \wedge a_n \Rightarrow a$.

Lemma 1 $s \models^{\circ}_{\emptyset} \varphi\ [\psi] \Leftrightarrow (\psi \Leftrightarrow s \models \varphi)$.

Proof The above lemma holds trivially for the tableau Rules 1–4. Rule 5 follows directly from the semantics of boolean connective $\Leftrightarrow$: $\neg\psi \Leftrightarrow s \models \neg\varphi$ is equivalent to $\psi \Leftrightarrow s \models \varphi$. Rule 6 corresponds to the disjunctive formula $\varphi_1 \vee \varphi_2$ where the variation point obligation of the disjunction is equal to the disjunction of the variation point obligations of the corresponding disjuncts φ_1 and φ_2.

Rules 7 and 8 handle the EU formula. The rules take into consideration the fixed point nature of the formula. $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ is a least fixed point formula, the semantics of which is computed by least fixed point of the function: $f = \varphi_2 \vee (\varphi_1 \wedge EXf)$.

Note that proof of satisfiability of a least fixed point formula at a state cannot be obtained from a path with a loop (infinite path); least fixed point formulas always have a finite path proof. The history H keeps track of whether there is a loop in the proof of whether s satisfies $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$. If such a loop is detected, s does not satisfy $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ in this loop and the interface obligation is $(\mathtt{ff}, \epsilon, \perp)$. On the other hand, if a loop is not detected, the formula $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ is expanded to its equivalent form and the history set is updated to include a new entry $(s, \mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2))$ to capture the fact that we have tried to identify the satisfiability conditions of $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ at the state s.

Rules 9 and 10 deal with greatest fixed point formula $\mathtt{EG}(\varphi)$. The semantics of $\mathtt{EG}(\varphi)$ is computed from the greatest fixed point of the function: $f = \varphi \wedge EXf$.

Proof of satisfiability of the greatest fixed point formula at a state will contain loops (infinite paths). In this case, Rule 9 states that s satisfies $\mathtt{EG}(\varphi)$ if the history set contains an entry $(\mathtt{EG}(\varphi), s)$, and the corresponding interface obligation is $(\mathtt{tt}, \epsilon, \perp)$. The entry in H proves that there exists a path from s to itself where in each state in the path the formula φ is satisfied. This in turn proves that there exists an infinite path from s where every state satisfies $\mathtt{EG}(\varphi)$. Rule 10 corresponds to the case where H does not include $(\mathtt{EG}(\varphi), s)$, and as such the tableau rule expands the formula $\mathtt{EG}(\varphi)$ to its equivalent form and updates the history set.

Finally, Rules 11 and 12 correspond to the case where the formula to be satisfied is of the form $\mathtt{EX}(\varphi)$. If s is not a variation point (Rule 11), then the formula $\mathtt{EX}(\varphi)$ can be satisfied if and only if φ is satisfied in any one of its next states. Therefore, the variation point obligation is computed from the *disjunction* of the variation point obligations corresponding to the satisfiability condition of φ at each next state. On the other hand, if s is a variation point (Rule 12), then the variation point obligation for s also includes the disjunction of $(\varphi, s, \vee)$. In short, if a feature, composed at the variation point s, satisfies φ then s satisfies $\mathtt{EX}(\varphi)$.

The tableau rules consider all the syntactic constructs of CTL (constructs with universal path quantifiers can be obtained via negation and existential path formulas), and we have proved that all the rules are sound. The tableau constructed using these rules is always finite where the number of nodes in the tableau is bound by the number of states in the FM and number of subformulas in φ. This concludes the proof of the lemma. □

Lemma 2 *$s \models_H^{\circ} \varphi\ [\psi] \Leftrightarrow (\varphi, s) \leftarrow \psi\ \in\ \mathtt{aSet}_{out}$ where $\psi := \mathtt{t_Obl}(\varphi, s, H, \emptyset, \mathtt{aSet}_{out})$*

Proof We show that the function t_Obl (Fig. 2) correctly implements the tableau rules in Fig. 5. Observe that, t_Obl invokes Obl in a mutually recursive fashion. Each rule for the definition of Obl directly encodes the tableau rules in Fig. 5. For example Rule 3 in Fig. 2 corresponds to Rules 3 and 4 in Fig. 5.

Note that the tableau rules allow repeated computation of $s \models_H^{\circ} \varphi\ [\psi]$ as they do not keep track of whether this computation has been completed in some other part of the tableau. Their implementation t_Obl, however, avoids such repeated computations, with the same parameters s, φ and H, using the fourth argument $\mathtt{aSet}_{in}$ (case 1

in definition A for `t_Obl` in Fig. 5). The second case in definition A is little bit more involved. This is due to the fact that `aSet` is updated when among the recursive calls to `t_Obl` via `Obl` for the same s and φ, at least one of them is returned ahead of others. Therefore, the `aSet` needs to be updated after all such calls have returned. The updates result in disjunction or conjunction of each return depending on whether the formula under consideration is an existential or universal path property. In our setting, we only consider existential path properties which will be the result from the disjunction of all return valuations. This situation occurs when $\varphi := \mathtt{EX}(\varphi')$ and there is a loop on s. □

Proposition 1 $(\varphi, s) \leftarrow (\mathtt{tt}, \epsilon, \bot) \in \mathtt{aSet} \Leftrightarrow s \models \varphi$ *where* $(\mathtt{tt}, \epsilon, \bot) := \mathtt{t_Obl}(\varphi, s, \emptyset, \emptyset, \mathtt{aSet})$.

Proof The proposition follows from Lemmas 1 and 2. □

Correctness of `update`

Lemma 3 $(\varphi, s) \leftarrow \psi \in \mathtt{update(aSet)} \Leftrightarrow s \models^{\circ}_{\emptyset} \varphi\ [\psi]$.

Proof Following the definition of `update` in Sect. 4.3, there are two cases. In the first case, `update` does not alter the variation point obligation

$$\begin{aligned}
&\begin{pmatrix} (\varphi, s) \leftarrow \psi \in \mathtt{update(aSet)} \\ \wedge \forall(\varphi', s', op) \in \mathtt{sub}(\psi) : \nexists(\varphi', t') \leftarrow (pc, \epsilon, \bot) \in \mathtt{aSet} \end{pmatrix} \\
&\quad\Leftrightarrow \quad \psi := \mathtt{t_Obl}(\varphi, s, \emptyset, \emptyset, \mathtt{aSet}) \\
&\quad\Leftrightarrow \quad s \models^{\circ}_{\emptyset} \varphi\ [\psi] \text{ from Lemma 1}
\end{aligned}$$

In the second case, where the variation point obligation is altered, WLOG we will consider $\mathtt{FM}_1$ and $\mathtt{FM}_2$ where s, s' are states in $\mathtt{FM}_1$ and t' is a state in $\mathtt{FM}_2$ such that $s' \to t' \in T_c^{\mathtt{FM}_1,\mathtt{FM}_2}$. Then,

$$\begin{aligned}
&\begin{pmatrix} (\varphi, s) \leftarrow \psi[\psi'/\psi''] \in \mathtt{update(aSet)} \wedge \\ \psi' = (\varphi', s', op) \wedge (\varphi', t') \leftarrow (pc, \epsilon, \bot) \in \mathtt{aSet} \\ \psi'' = \begin{cases} \psi' & \text{if } (pc = \mathtt{tt}) \wedge (op_i = \wedge) \\ \psi' & \text{if } (pc = \mathtt{ff}) \wedge (op_i = \vee) \\ (\mathtt{tt}, \epsilon, \bot) & \text{if } (pc = \mathtt{tt}) \wedge (op_i = \vee) \\ (\mathtt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases} \end{pmatrix} \\
&\quad\Leftrightarrow \begin{pmatrix} \psi = \mathtt{t_Obl}(\varphi, s, \emptyset, \emptyset, \mathtt{aSet}_1) \wedge (pc, \epsilon, \bot) \\ \quad = \mathtt{t_Obl}(\varphi', t', \emptyset, \mathtt{aSet}_1, \mathtt{aSet}) \end{pmatrix} \\
&\quad\Leftrightarrow \quad \psi[\psi'/\psi''] = \mathtt{t_Obl}(\varphi, s, \emptyset, \emptyset, \mathtt{aSet}_1) \\
&\qquad\qquad \text{where } s \text{ is a state in } Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2) \\
&\quad\Leftrightarrow \quad s \models^{\circ}_{\emptyset} \varphi\ [\psi[\psi'/\psi'']] \text{ from Lemma 1}
\end{aligned}$$

□

Correctness of `INTERP` *(in Algorithm 1)*

Lemma 4 *The maximum recursion depth of the function* $\texttt{INTERP}((\varphi, s), \emptyset)$ *is bound by* $|\texttt{aSet}|$.

Proof Assume that $\texttt{INTERP}((\varphi, s), \emptyset)$ terminates at k recursion depth where $k > |\texttt{aSet}|$ where $|\texttt{aSet}|$ is the number of elements of the form $(\varphi_i, s_i) \leftarrow \psi_i$. This implies that there are k calls to `INTERP` with k different pairs (φ, s) which implies there are k elements in $|\texttt{aSet}|$. This leads to a contradiction proving that our assumption is incorrect. □

Lemma 5 *Given an* `aSet` *such that* $\forall(\varphi, s) \leftarrow \psi \in \texttt{aSet} \wedge \forall(\varphi', s', op') \in \texttt{sub}(\psi) \Rightarrow \forall s' \to t' \in T_c^{\text{FM}_m, \text{FM}_n} : (\varphi', t') \leftarrow \psi' \in \texttt{aSet}$, *then the following holds*:

$$(\varphi, s) \leftarrow \psi \in \texttt{aSet} \wedge s \models \varphi \quad \Leftrightarrow \quad \texttt{INTERP}((\varphi, s), \emptyset)$$

Proof We first prove that `INTERP` handles inter-feature loops correctly. In case of loops, `INTERP` can be recursively invoked with the same pair (φ, s). Recall that satisfiability of greatest fixed point formulas (e.g., `EG`) require loops while satisfiability of least fixed point formulas (e.g. `EU`) by a state cannot be inferred from a loop. As such, `INTERP` returns `tt` if the inter-feature loop represents a path corresponding to satisfiability of greatest fixed point; otherwise, it returns `ff`.

For $(\varphi, s) \leftarrow (pc, \epsilon, \bot) \in \texttt{aSet}$, the lemma is proved as follows:

$$\begin{aligned} & (\varphi, s) \leftarrow (pc, \epsilon, \bot) \in \texttt{aSet} \wedge s \models \varphi \\ \Leftrightarrow \quad & s \models^{\circ}_{\emptyset} \varphi\ [(pc, \epsilon, \bot)] \wedge s \models \varphi \\ \Leftrightarrow \quad & \text{From Lemmas 2 and 3} \\ & pc = \texttt{tt} \\ \Leftrightarrow \quad & \texttt{INTERP}((\varphi, s), \emptyset)\ \texttt{INTERP}\ \text{returns} \\ & pc\ \text{(see Line 2 in Algorithm 1)} \end{aligned}$$

For $(\varphi, s) \leftarrow \psi \in \texttt{aSet}$ where $\psi = (\varphi_1, s_1, op_1)op_2 \cdots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet}$ the proof proceeds as follow:

$$\begin{aligned} & ((\varphi, s) \leftarrow (\varphi_1, s_1, op_1)op_2 \cdots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet} \wedge s \models \varphi \\ & \quad \Leftrightarrow \quad ((\varphi_1, s_1, op_1)op_2 \cdots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \quad \Leftrightarrow \quad s \models \varphi) \wedge s \models \varphi \\ & \text{From Lemma 1} \\ & \quad \Leftrightarrow \quad \texttt{INTERP}((\varphi, s), \emptyset)\ \text{using Lines 7–14 in Algorithm 1} \end{aligned}$$

□

Finally, Theorem 1 can be proved as follows. Lines 2–7 in Algorithm 2 compute the variation point obligation of $\texttt{FM}_1$ in the sequence. Lines 8–15 compute the variation point obligation for all $\texttt{FM}_i$ $(1 \leq i \leq m)$. At each iteration the $\texttt{aSet}_{current}$

obtained at the end satisfies the following:

$$(\varphi, s) \leftarrow \psi \in \mathtt{aSet}_{current} \quad \Leftrightarrow \quad s \models^{\circ}_{\emptyset} \varphi\,[\psi] \quad \Leftrightarrow \quad (\psi \;\Leftrightarrow\; s \models \varphi)$$

The above follows from Lemmas 1, 2 and 3.

The terminating condition of the iteration $\mathtt{aSet}_{current} = tmp$ is satisfied only when no new additions and updates are possible in the answer set. That is, $\forall(\varphi, s) \leftarrow \psi \in \mathtt{aSet}_{current} \wedge \forall(\varphi', s', op') \in \mathtt{sub}(\psi) \Rightarrow \forall s' \rightarrow t' : (\varphi', t') \leftarrow \psi' \in \mathtt{aSet}_{current}$. Finally, at Line 16 the algorithm COMPOSE returns the result correctly inferring whether or not $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \ldots, \mathtt{FM}_m) \models \varphi$. This follows from Lemmas 4 and 5. □

This concludes the proof that the COMPOSE algorithm is sound and complete.

6 Implementation and application to an example

We have implemented the compositional model checking algorithm in a research prototype model checker (Liu 2010) to help show the feasibility of our approach. We further provided evaluation of the approach on a pacemaker product line that was not able to be checked using other sequential model checking techniques. In this section, we describe the implementation details, discuss the bounds on space complexity, and provide experimental results.

6.1 Implementation details

Our prototype model checker is implemented using Python 2.6. It faithfully represents the algorithms described in Sect. 4. Some details beyond Sect. 4 are: (1) formulas, obligations, and features are represented as classes with their own attributes and functions; (2) the functions `t_Obl` and `Obl` use only one `aSet` in the argument, as `aSet` gets updated inside each function; (3) if the first disjunct in a disjunctive formula is evaluated to `tt` in `Obl` function, the other disjunct in the same formula is not analyzed; and (4) in the `Interp` function, a given formula is analyzed against all equivalent forms of greatest fixed point formulas (gfp) to decide if `tt` or `ff` result should be returned.

6.2 Bound on space usage and reduction in time complexity

We now discuss the bound on space usage of the above implementation. Suppose that there are N features present in the final product. The max number of states in any feature is $|S|$, where S denotes the set of states; the maximum out-degree of any state in any feature is d (the branching factor); and the given formula to be verified on the final product has $|\Phi|$ subformulas.

The space complexity is guided by the storage requirement for maintaining the history and answer sets. The history set is maintained for each feature in isolation. Once a feature has been analyzed, its history is removed. So the space complexity due to the history set is $O(|S| \times |\Phi|)$.

The answer set is maintained throughout the entire computation across features. It records set of tuples of the form: $((\varphi, s), (\varphi_1, s_1, op_1)op_2 \cdots op_{2k-2}(\varphi_k, s_k, op_{2k-1}))$. The number of such tuples in the answer set for one feature is bounded by the number of different first elements in the above tuple: $O(|S| \times |\Phi|)$.

The second element of the tuple contains formulas of the form $(\varphi_i, s_i, op_{2i-1})$ $(1 \leq i \leq k)$ where the type of op_{2i-1} can be either a conjunction or a disjunction. Therefore, the number of formulas of the form $(\varphi_i, s_i, op_{2i-1})$ is of the order $O(|S| \times |\Phi| \times 2)$. These formulas are combined using the operators op_{2i-2} $(2 \leq i \leq k)$, and the number of such operators (conjunction and disjunction) can be at most $d-1$ (i.e., $k = d$), where d is the maximum out-degree of any state in any feature. The number of different second elements of the tuple is equal to the number of different formulas that can be formed by combining $|S| \times |\Phi| \times 2$ sub-formulas of the form $(\varphi_i, s_i, op_{2i-1})$ in $d-1$ different ways: $O((|S| \times |\Phi| \times 2)^d)$.

Proceeding further, the size of the answer set is of the order $O((|S| \times |\Phi|)^{d+1} \times 2^d)$, and the overall size for N features is $O((|S| \times |\Phi|)^{d+1} \times 2^d \times N)$.

The space bound computed above is just for one product. Suppose that there are K products in the product line. Model checking the entire product line using compositional model checking takes $O((|S| \times |\Phi|)^{d+1} \times 2^d \times N \times K)$. The more features shared in the product line, the smaller the overall space usage is, as the space usage for each new product is essentially $O((|S| \times |\Phi|)^{d+1} \times 2^d \times M)$, where M is the number of new features introduced in that product.

For non-compositional model checking, the space bound for checking a product line is $O(|S| \times |\Phi| \times N) \times K)$. The penalty in space usage for compositional model checking is caused by storing the extra information needed for reusing model checking results. However, the storage of this information lowers the computational effort for property verification in the product line. In Sect. 6.3, we will empirically show in the pacemaker example the advantages of such reuse of model checking results.

The above analysis suggests that the smaller the branching factor (d) and the larger the subset of features that is reused, the higher the space usage efficiency is for compositional model checking.

On the other hand, the time complexity of the model checking is linear to the number of states in the system and the number of sub-formulas in the formula to check.

Suppose there are K products in the product line, each composed of one or several of the N features, the max number of states in any features is $|S|$, and the number of subformulas of the formula to check is $|\Phi|$. For simplicity, suppose that the i-th product is composed of i number of features.

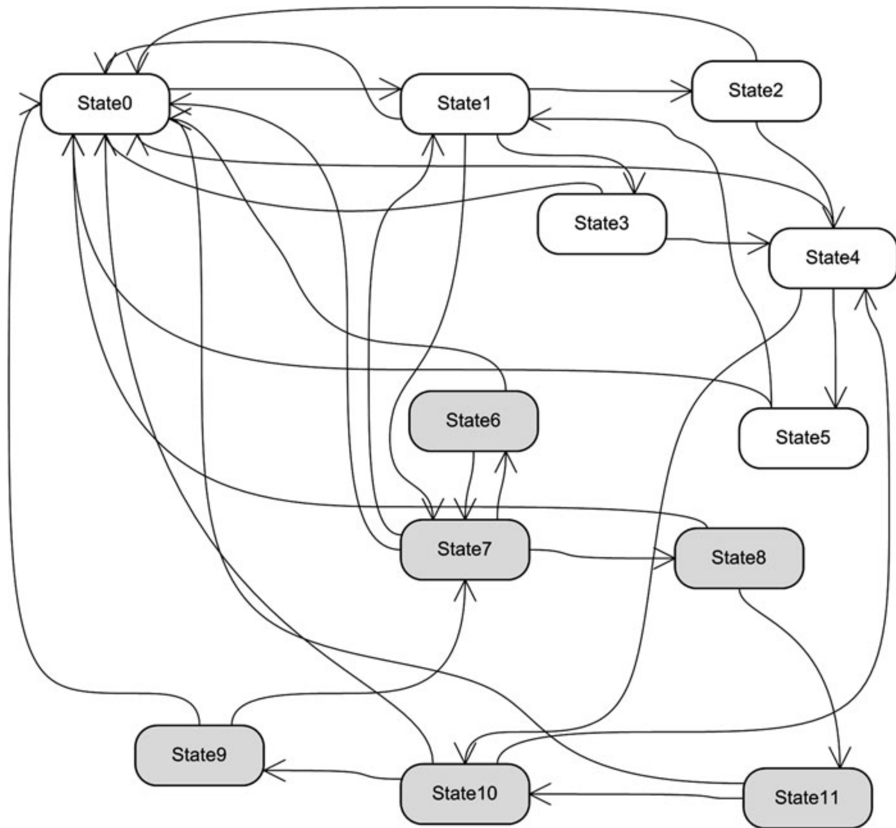
Thus, the time complexity due to the number of states for the non-compositional model checking is: $O(|S| + 2|S| + \cdots + K|S|) = O(|S| \times K^2)$. This is linear in the sum of states of all products. For compositional model checking, we save the effort of re-exploring the states of a previously checked feature. Therefore, the time complexity due to states can be reduced to $O(|S| + (2|S| - |S|) + (3|S| - 2|S|) + \cdots + (K|S| - (K-1)|S|)) = O(|S| \times K)$. This is linear in the number of states in the largest product in the product line.

Adding the formula into the consideration, the time complexity for checking the product line using non-compositional model checking is $O(|S| \times K^2 \times |\Phi|)$; for compositional model checking it is: $O(|S| \times K \times |\Phi|)$.

6.3 Reduction in state exploration

We evaluated our technique by conducting experiments on the pacemaker product line discussed in Sect. 2. As described below, the results showed a reduction in state exploration.

Figures 6 and 7 depict how the Mode Transitive Extension ($\mathtt{FM}_1$) and Rate Responsive Extension ($\mathtt{FM}_2$) are sequentially composed with the BasePacemaker's controller ($\mathtt{FM}_0$). The states in the extensions are shown in grey. The variation points are the states which have outgoing transitions leading to another model in the composition. The propositions satisfied at each state are shown in the corresponding tables below each figure.



State	True Propositions	State	True Propositions
State0	timerOff=1	State6	timerSenseTimeUp=1; sensorOn=1; triggeredMode=1;
State1	sensorOn=1; inhibitedMode=1;	State7	sensorOn=1; triggeredMode=1;
State2	sensed=1; sensorOn=1; inhibitedMode=1;	State8	sensed=1; sensorOn=1; triggeredMode=1;
State3	timerSenseTimeUp=1; pulseGen=1; inhibitedMode=1;	State9	timerRefractoryTimeUp=1; sensorOn=1; triggeredMode=1;
State4	inhibitedMode=1;	State10	triggeredMode=1;
State5	timerRefractoryTimeUp=1; sensorOn=1; inhibitedMode=1;	State11	pulseGen=1; triggeredMode=1;

Fig. 6 Base controller and mode-transitive extension

State	True Propositions
State6	sensorOn=1; inhibitedMode=1; upperRateLimit=1;
State7	sensorOn=1; sensed=1; inhibitedMode=1; upperRateLimit=1;
State8	timerShortSenseTimeUp=1; pulseGen=1; inhibitedMode=1; upperRateLimit=1;

Fig. 7 Base controller and rate-responsive controller extension

In the controller for the fourth product (not shown), the ModeTransitive and RateResponsive features, $\mathtt{FM}_1$ and $\mathtt{FM}_2$ are sequentially composed with $\mathtt{FM}_0$, as shown in Figs. 6 and 7. $\mathtt{FM}_1$ and $\mathtt{FM}_2$ do not connect directly one to the other since a pacemaker controller can be in either the Triggered Mode or Inhibited Mode, but not in both at the same time.

The following CTL formula describes the required property for the product line that was introduced in Sect. 2:

$$\mathtt{AG}((\mathtt{sensed{=}0} \wedge \mathtt{timerSenseTimeUp{=}1} \wedge \mathtt{inhibitedMode{=}1}) \Rightarrow \mathtt{EF}(\mathtt{pulseGen{=}1} \wedge \mathtt{inhibitedMode{=}1}))$$

This formula was used in the following evaluation.

To evaluate the space and time performance of our approach, we compared our compositional model checking (CMC) technique with non-compositional model checking (NMC) for the four products in the product line. Table 2 records the experimental results. In the table, MT denotes ModeTransitive and RR denotes RateResponsive. The variables (`t_Obl`) and (`Obl`) record the number of times the `t_Obl` and `Obl` functions are performed, respectively. The count of their invocations helps evaluate the reduction in the number of times the same state space is explored. Note that the `Obl` function is the actual state-exploration function in our model checker, as `t_Obl` invokes `Obl` only if a (formula, state) pair is not already in the `aSet` (otherwise `t_Obl` will return the obligation found in `aSet` directly).

The results of NMC are obtained from using our approach to model check each of the four products in its entirety, without breaking it into separate features. In other words, each sub-formula of the given property is checked at the composed product, generating $(\mathtt{tt}, \epsilon, \bot)$ or $(\mathtt{ff}, \epsilon, \bot)$ results at the initial state of that composed product, denoting the property being satisfied or not. In addition, `aSet` is not used for NMC to store any intermediate results.

The results of CMC are obtained from calculating the test data for the incrementally added features and connections in each product. For example, verifying the BasePacemaker involves checking $\mathtt{FM}_0$ with states 1, 4, and 5 as its variation points (i.e., the union set of the variation points needed by the MT and RR extensions), plus applying the $\mathtt{update}_F(\mathtt{aSet})$ to get the final result. Verifying the RateResponsivePacemaker involved checking $\mathtt{FM}_2$ with states 6, 7, and 8 as its variation points, plus checking the connections from $\mathtt{FM}_2$ to $\mathtt{FM}_0$, and applying the `update` and $\mathtt{update}_F(\mathtt{aSet})$ functions. Test results were similarly collected for the other two products.

Both NMC and CMC results we obtained running on the same machine with the same configuration (1.60 GHz Intel (R) Pentium (R) M Processor, 1.00 GB of RAM).

Table 2 shows that our compositional model checking approach did provide savings in the product line. For example, the second row shows that in NMC the cost for checking the product line (measured in (`Obl`)) was $250 + 415 + 543 + 708 = 1916$. However, as seen in the second row of the bottom half of the table, the cost in CMC

Table 2 Test data for pacemaker product line

# of invocations	Base pacemaker	MT pacemaker	RR pacemaker	MT-RR pacemaker
	Non-compositional model checking (NMC)			
(`t_Obl`)	250	415	543	708
(`Obl`)	250	415	543	708
# of state visits	21	36	46	61
	Compositional model checking (CMC)			
(`t_Obl`)	126	391	290	277
(`Obl`)	113	60	112	0
# of state visits	7	4	7	0
Size of `aSet`	111	171	282	282

was only $113 + 60 + 112 + 0 = 285$. The savings in invocation were due to the fact that the common features had to be checked repeatedly in NMC but not in our CMC approach, and that CMC stores prior checking results of each state in `aSet`. NMC saves the space usage of `aSet`, but incurs more overhead with the invocations. Note that if no prior checking for any of the features had been done (i.e., if there had been no reuse), then the cost for checking the RateResponsivePacemaker (measured in (`Obl`)) would have been $113 + 60 = 173$. As with the product-line approach itself, CMC shows more savings when more features are reused.

7 Discussion

In this section, we discuss the usefulness and applicability of this technique in different systems, in different types of changes, and in different phases of product-line development.

7.1 Handling parallel composition of features

Our work falls into the category of compositional verification (Abadi and Lamport 1995). We use sequential composition (Definition 2) rather than parallel composition, as in, e.g., Giannakopoulou et al. (2002); Basu and Ramakrishnan (2006), because it would add unnecessary complexity to the state space by obscuring the interfaces among features that we want to maintain in a product-line setting for effective reuse. For example, parallel composition of interacting processes produces a global product whose number of states is exponential to the number of component processes (Berezin et al. 1998). Moreover, one interface states of a feature appears in multiple states of the global product as a result of the composition.

Although our work targets sequential composition of features (Sect. 3.1), the technique described in this paper can also handle many instances of parallel composition of features. Specifically, it manages the situation in which a Feature A, composed in parallel with a Feature B, provides only control signals to Features B, and the CTL property to be verified does not involve the internal behavior of A. In such a case, the control signals from A can be modeled as abstract events in B. In other words, these abstract events represent the effect of feature A on Feature B.

Recall that in sequential composition, the set of transitions of the composite model is the union of the transitions from the composed features (local transitions). This means that a transition in the composite model (a global transition) is just one of the local transitions. However, in parallel composition, a global transition is a combination of several local transitions that may occur at the same time Huth and Ryan (2004).

For an example of parallel composition, consider the interaction of the Extra Sensor feature (the functionality of the extra sensor component) and the RateResponsiveExtension feature in the RateResponsivePacemaker. These two features are composed in parallel because both feature models can make a transition at the same time. Since the CTL property to be verified here does not depend on the internal behavior of the Extra Sensor, we model the Extra Sensor's effect on the extension to

the base controller's functionality by introducing an abstract event in the extension ("`upperRateLimit=1`" in Fig. 7(b)'s table).

Another example of parallel composition that our technique can handle is the Timer feature (whose behavior is parallel to that of the controller's, not shown in Fig. 1). We again model the effect of the Timer by introducing abstract events ("`timerSenseTimeUp = 1`", "`timerRefractoryTimeUp = 1`" and "`timerShortSenseTimeUp = 1`" in Fig. 7(b)'s table) in the controller's model. This allows us to manage the rest of the feature compositions in a pure sequential fashion.

Abstracting such events is currently done manually. In the future, work on generating assumptions from the environment, e.g., Giannakopoulou et al. (2002), Gheorghiu et al. (2007), could possibly be incorporated to handle such an abstraction process in an automatic fashion.

7.2 Suitability for product-line evolution

During product-line evolution, structural changes to a composed feature model may occur, e.g., adding a new feature, replacing or changing an existing feature, or deleting a feature. The model checking technique described in this paper supports evolution by minimizing the amount of re-checking that is required in these cases.

Adding a new feature through a pre-specified variation point s_i means that one or several states are introduced–$s_i \rightarrow t_i \in T_c^{\mathrm{FM}_m,\mathrm{FM}_n}$ (Line 10 of Algorithm 2). Thus, `t_Obl`(φ_i,t_i,$\emptyset$,`aSet`$_{current}$,`aSet`) is invoked for each such t_i, together with each of the formula φ_i that appears in the obligation of s_i (Line 11 of Algorithm 2). This change does not require re-checking other features because we can have the `t_Obl` function update the answer set obtained from the previous check of the feature composition. The subsequent `update` and `update`$_F$ function will be performed on the updated answer set.

On the other hand, adding a new feature through an ad-hoc variation point (i.e., one not specified in the checked model prior to the addition) is more complicated as there will not have been any obligations previously generated for such an ad-hoc variation point. An example of this in the pacemaker example is when variation points s_1 and s_5 are specified for the base controller feature to check the composition with the RateResponsiveExtension. Later when it is composed with the ModeTransitiveExtension, an extra variation point s_4 needs to be added, but no variation point obligation has yet been generated for s_4 in the prior checking of the base controller feature.

To take advantage of previous model checking, for such an ad-hoc variation point s_4, we identify all elements in the answer set from the previous check that have the form $(\varphi, s_4) \mapsto \psi$. For each such element, rechecking φ at s_4 using the function `t_Obl` (with s_4 a variation point) will generate the variation point obligations needed to check the added new feature. The rest of the process for model checking an added feature is the same as the above situation. Rechecking φ is very light-weight as all the states such as s_3 and s_8, $s_3 \rightarrow s_4$ in base controller feature, and $s_8 \rightarrow s_4$ in RateResponsiveExtension feature have been checked already. Applying `t_Obl` on those states can readily reuse the existing elements like $(\varphi', s_3) \leftarrow \psi'$ in the answer set.

The process for handling other changes in the composed feature model (e.g., replacing or changing an existing feature, or deleting a feature) is similar to the above

description. However, since the affected features in this case may already be connected to other features, updates to variation point obligations of the affected changed features may, if needed, result in rechecking all features directly and indirectly connected to them (Line 10–15 of Algorithm 2). (Indirect connections refer to the situation in which some features do not have inter-feature transitions to the changed feature but are connected to those features that have such transitions.) The important point is that features that are not affected will not be rechecked.

When structural changes to a feature occur (e.g., new transitions and states are added, or existing transitions and states are modified or deleted), we can divide the original feature into several "mini features", representing the isolated changed parts, and the original model without those changed parts. Rechecking the original model can thus be carried out in a way similar to sequentially composing the changed parts with previously checked parts, and the variation points affected by such a structural change will be rechecked. This allows us to reuse as much of the still-valid answer set elements from the previous checking as possible, and to check if some of the original obligations are preserved after the change. (By "preserved" we mean that the corresponding answer set elements involving those obligations do not need to be updated).

In summary, variation point obligations provide the modularity needed to model check product lines. The fact that any state in a model can become a variation point, and the storing of previous obligation-generation results in the answer set, help reduce the rechecking load in case of changes. Because product lines routinely experience significant change over their lifetimes, the continued usefulness of previous model checking results contributes to the practicality of this technique.

8 Related work

Several recent works have investigated representations of variability within a product line in behavioral models. Gruler et al. (2008) describe PL-CCS as a way to capture the dynamics of variabilities in a product line in terms of process algebra and present a model checking algorithm for verifying properties in each of the individual products. Fantechi and Gnesi (2007) identify whether a product belongs to a product line. Fischbein et al. (2006) propose a technique to determine whether a variability undermines a product-line property. Lauenroth and Pohl (2007) describe how variability complicates the consistency checking of a product in the product line. Kishi et al. (2005) verify many of the test scenarios in an industrial product line, using a reusable verification model and the SPIN model checker. In these approaches, the product line offers a well-defined base with relatively small variations (e.g., transitions in a finite state machine) that may be verified through techniques like behavioral conformance (Fischbein et al. 2006). In contrast, the approach described here offers compositional verification of the product line and treats the common and variable functionality as equal units to be composed.

Larsen et al. (2007) have modeled product line assets as modal I/O automata to detect whether there can exist any composition of the two automata that is error free, under the assumption (dropped in our work) that no new features will be added to the

product line. More recently, Classen et al. (2010) have introduced Feature Transition Systems, where the state space of a product line is represented as a single such system and all possible products are verified against a property at once.

In this section, we summarize the related works on applying compositional verification of products in product line and provide an overview of distinguishing aspects of our technique. We also present in Sect. 8.1 a detailed comparison between our technique and the most closely related work (Fisler and Krishnamurthi 2001, 2002; Li et al. 2002a, 2002b, 2005; Blundell et al. 2004) and discuss how our work enables additional verification in the product line setting.

Blundell et al. (2004), like us, propose an approach in which interface obligations are generated as temporal properties. Their technique differs from ours in requiring the interface states (here, the variation points) to be terminal states with no outgoing transitions. By relaxing this constraint we can handle cycles in the composed state space (e.g., inter-feature loops as shown in Fig. 6). This provides the flexibility we needed to accurately model the pacemaker product line.

Moreover, their definition of feature model includes a set of data propositions. Different from control propositions, which are determined by labeling functions, data propositions are assigned explicitly by the feature model designer, and are persistent across features until changed. This provides a way to model shared data among features, especially for open systems in which some data values may not be available when analyzing certain features. In our approach, we do not require data propositions in the feature model. In the case where the value for some propositions are not available when analyzing a single feature, we are still able to check properties containing those propositions, because obligations containing those data propositions for the next feature are generated at the variation states. A detailed comparison follows in Sect. 8.1.

Wang (2005) extends the work of Blundell et al. (2004) and allows cycles in the composed state space (i.e., inter-feature loops). It is assumed there that interface states suffice for feature composition, so non-interface states are never re-explored during verification. Inter-feature loops where non-interface states need to be re-verified through different entering paths (e.g., the initial state of the base feature in the pacemaker model) are thus not addressed, unlike in our technique. For example, in the pacemaker, we wanted to check if the desired property was satisfied when the ModeTransition Extension re-connects with the BaseController feature through the initial state of the BaseController feature in a new product.

Thang (2005) presents necessary conditions which, when satisfied by a base feature and one or more extension features, ensure that property verification results hold both before and after the base is extended. Although this work allows loops between the base and the extensions, it does not provide insights into the cases where the necessary conditions are violated.

Krishnamurthi and Fisler (2007) have described compositional model checking of systems using aspects to verify the behavioral properties. Composition is described in terms of advice, modeled as a state machine, that alters the system's behavior at specified locations (join points, or function calls). Their technique requires that the specification of where the aspect will apply (the pointcuts, or set of states) be done prior to verification, whereas the technique we describe allows any state to potentially

become an interface in the future. Our approach is also less constrained in that it removes the need to know beforehand the order in which features will be added, and it permits re-entry to any feature, not just to the initial feature. Like ours, their model checking is incremental and aimed at reducing the amount of re-verification needed as a system changes. However, we believe that by imposing fewer constraints on the structure of the state machines and the type of circular dependency between the state machines (e.g., not requiring cycles in state machines to include specific entry and exit states), the more-general approach presented here places fewer burdens on the product-line developer.

Since how product lines will evolve is notoriously hard to predict, we have tried to make modeling decisions that keep all options open for the choice of which variability features will be selected to become part of each new product and the order in which those features will be composed.

Xie and Browne (2003) model each component (e.g., of a product line) as an Asynchronous Interleaving Message-passing (AIM) computation model (i.e., only one model can execute at a time), and use assume-guarantee to generate assumption on the environment (i.e., other components in the composition) for a given component. The interface between two components is the input and output message types between the two. The system verification is carried out on the abstraction of the composite model, which is obtained from the environmental assumptions, the verified properties, and the messaging interfaces of the constituting components. Their work differs from ours in that both their computation model (especially the type of interface) and model checking strategy are suitable for verifying the communication between components, while ours targets the increments of functionality of one or more systems.

Nejati et al. (2008) describe a compositional algorithm for synthesizing pipeline arrangements of features that are "safe", i.e., the ordering of the features will not bring undesirable interactions specified in the given properties. The synthesis operation is done in a pair-wise (incremental) fashion. Once the synthesis operation has completed, they conduct the verification operation on the global composition features, although the number of possible compositions are much smaller as the "unsafe" ones have been pruned away by the synthesis step. Like us, their approach plans for change to allow reuse of previous results as possible. Their work differs from ours in that they are interested in synthesizing a safe composition rather than in verifying that a proposed composition satisfies the properties. Their work requires that the features satisfy the transparency pattern (i.e., a feature remains unobservable to other features when not providing service). It also differs from ours in that they use parallel rather than sequential composition. Their approach can cause circular dependencies between features.

8.1 Comparison with related compositional verification approaches

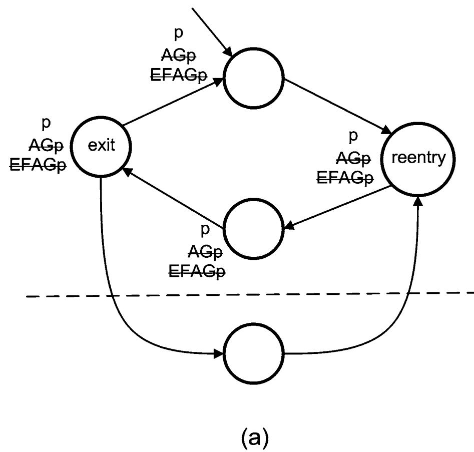
Works by Fisler and Krishnamurthi (2001, 2002) and subsequent extensions in Li et al. (2002a, 2002b, 2005) and in Blundell et al. (2004) are similar to our approach of applying compositional verification in product lines. In this section, we present a detailed discussion distinguishing our work from these existing techniques and showing what we can model check for the first time. The primary differences between these

works and ours are (a) the type of inter-feature loops being considered (loops only between features as a whole vs. loops between feature states, where the latter also allows circular dependency in the (un-) satisfiability of the properties by the features), and (b) the methods used to reason about properties of products with inter-feature loops.

Modular verification of collaboration based software designs (Fisler and Krishnamurthi 2001, 2002) report an algorithm for modular verification of features that are composed sequentially. It first performs CTL model checking on the base feature, and labels the re-entry state (i.e., the target state of an incoming transition from a new feature) and the exit state (i.e., the source state of an outgoing transition to a new feature). When a new feature is added, the algorithm gives placeholder states "in" and "out" to the new feature, and copies the labeling on the base feature's "re-entry" state to the "out" state in the new feature. Model checking of the new feature determines if each sub-formula of the original property always holds on the "in" state. If at the end of the checking, the labels derived for the "in" state of the new feature match those on the "exit" state of the base feature, then the property is preserved in the composed feature.

Note that, if the labels of the "in" and "exit" states do not match, the algorithm fails to infer whether or not the property under consideration is preserved after the feature addition; in such a situation, the algorithm terminates with an "Unknown" result. In other words, the algorithm is incomplete. For example, in Fig. 8 (from the extended technical report for FSE'01—Fisler and Krishnamurthi 2002), a new feature (the single state below the dotted line) is added to the base feature in both the model in Figs. 8(a) and 8(b). In both cases, the labels on the "in" state of the new feature do not match that on the "exit" state of the base feature. The algorithm in Fisler and Krishnamurthi (2002) reports this violation, but cannot determine whether this violation will make the original property *EF AGp* fail on the composed system. In contrast, the technique that we describe in this paper *will generate results* for both cases and successfully infer that *EF AGp* is satisfied at the start state of the model in Fig. 8(a) and is not satisfied at the start state of the model in Fig. 8(b).

Modular verification of open features through three-valued model checking (Li et al. 2002a, 2002b, 2005) extends the above algorithm (Fisler and Krishnamurthi 2001) to support model checking properties containing propositions with unknown values (i.e., where the values of those propositions are determined by some other features). Their algorithm adopts Bruns and Godefroid's 3-valued model checking (Bruns and Godefroid 1999) to label the states in a feature for a specific proposition as `tt`, `ff`, or Unknown. Their algorithm then checks whether the atomic propositions marked on the "in" state of the new feature match those of the "exit" state of the base feature by different cases, i.e., whether the new interpretation of an existing data proposition introduced by the new feature strengthens, weakens, or is logically equivalent to its previous interpretation in the base feature. If none of the above cases apply, the original property needs to be re-verified in the base feature using the new interpretation of the data proposition. Similar to Fisler and Krishnamurthi (2001), Li et al. note that their approach is incomplete and may not be able to successfully infer the (un)satisfiability of certain properties (Li et al. 2005, Sect. 5.3).



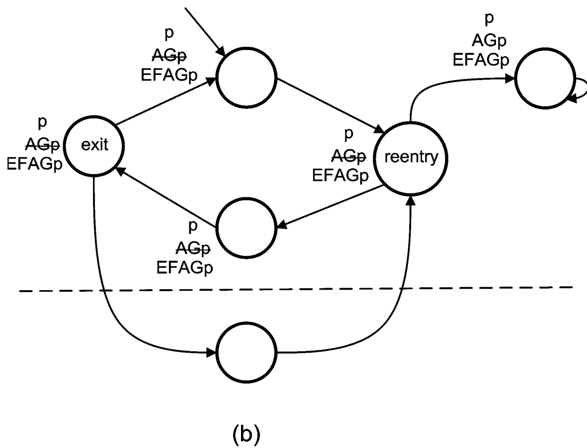


Fig. 8 Example of mismatched labels (based on Fisler and Krishnamurthi 2002)

Parameterized interfaces for open system verification of product lines (Blundell et al. 2003, 2004) extends the above algorithm to generate temporal formulas ("constraints") at the initial states of a base feature which are then parameterized over the possible successor states of the exit states.

To check if a property holds in a composed feature (e.g., a base feature and a new feature), this approach takes the following main steps:

Step 1 Checks every sub-formula of the property at the base feature and the new feature separately, generating temporal constraints at the initial state of each feature. Each temporal constraint is created by tagging applicable sub-formulas with some exit state of this feature.

Step 2 For the last feature present at the end of the sequential composition (e.g., the new feature here), evaluates the constraint at its initial state to `tt` or `ff`.

Step 3 Propagates backward these truth values to the feature that appears before it in the sequence, until the base feature is reached. In this step, the sub-formulas in the constraint which are tagged with some exit state are evaluated to `tt` or `ff` based on

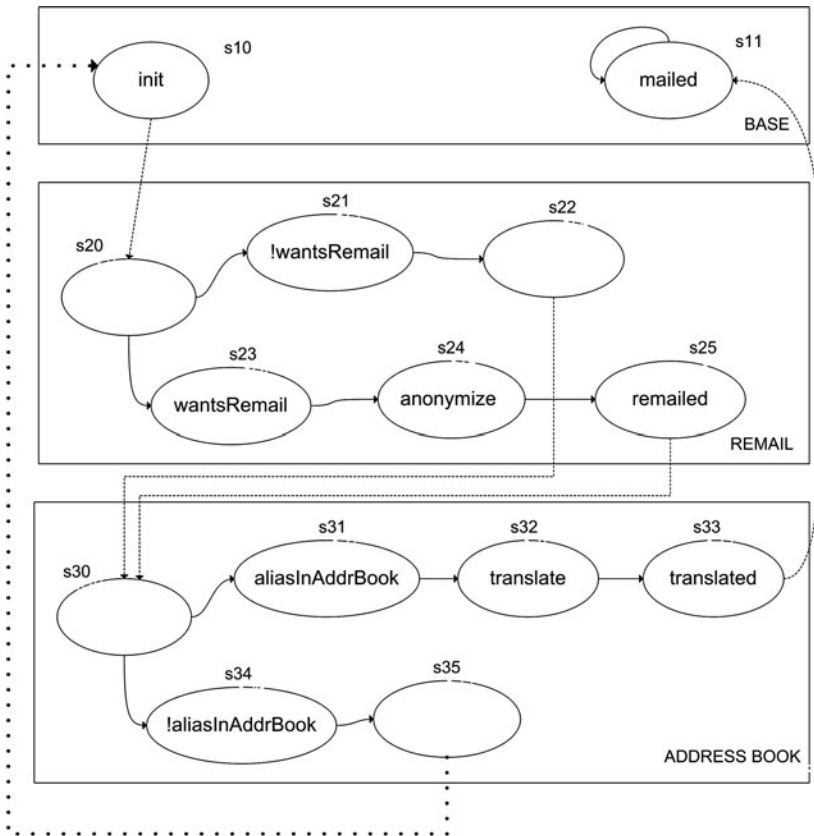


Fig. 9 Email example with loop

the truth value of the states connected to those exit states, e.g., the initial state of its subsequent feature.

Note that such backward propagation is possible only when there are no loops in states belonging to multiple features in the sequence (as will be explained in the example in Fig. 9). If such a loop exists, either Step 2 or Step 3 will yield "Unknown" as a result, rendering the approach incomplete.

Example We now discuss a simple example, adapted from Hall (2000) and previously used in Li et al. (2002a), and Blundell et al. (2004). We introduce loops across the state space of composed features in the example in order to show that, in contrast to existing techniques (Fisler and Krishnamurthi 2001, 2002; Li et al. 2002a, 2002b, 2005; Blundell et al. 2004), our technique can successfully compute verification results even in the presence of such loops.

As shown in the bottom box of Fig. 9, the Address Book feature looks up the alias given by the user in the address book. If the alias is in the address book, the Address Book feature will translate it to a valid address stored in the address book and

eventually mail the message; otherwise, it will go back to the initial state of the Base feature and have the user specify the options again. This looping back to the Base feature in case of failure of the Address Book lookup is an essential fault-tolerance requirement of the email product that allows us to demonstrate the advantages of our technique over the existing ones.

We show below how existing approaches are not able to handle this kind of inter-feature loop. We use a dotted line for the loop to indicate that we first describe the existing approaches ignoring the loop and then study the effect on the approaches of considering the loop.

Given a formula ϕ representing a temporal property, Blundell et al. (2004) considers all possible subformulas of the formula and checks for their satisfiability. For example, for the property $\phi = AG\ (remailed \Rightarrow EF\ mailed)$, the value of ϕ_{s_i} is set to true, false or unknown depending on whether it is satisfied, unsatisfied or undecided at state s_i.

We use the same property that is checked in Blundell et al. (2004): $\phi = AG$ $(remailed \Rightarrow EF\ mailed)$, which states that "a message marked as remailed can eventually be mailed". We first show below how Blundell et al.'s work accurately checks the above feature composition if the transition from state s35 to s10 in the email product is ignored (i.e., without a loop in the state space). We refer to their extended version of the ASE'04 paper (Blundell et al. 2004) in the technical report (Blundell et al. 2003) for their formalization.

Without an inter-feature cycle Both the ϕ_{s10} and $EF(mailed)_{s10}$ constraints can be reduced to `tt`. Thus, existing approaches suffice to show that the original formula ϕ holds at $s10$ of the Email product, if there is no inter-feature cycle (i.e., no transition from state $s35$ to $s10$).

With an inter-feature cycle However, when we add the transition from $s35$ to $s10$ back to the above email product, the cyclic dependency thus introduced cannot be handled. Specifically, the value of ϕ_{s35} and $EF(mailed)_{s35}$ should be replaced with the value of checking ϕ and $EF(mailed)$ at $s10$, which are ϕ_{s10} and $EF(mailed)_{s10}$ separately. ϕ_{s10} and $EF(mailed)_{s10}$ cannot be reduced to `tt` or `ff` value until the value of ϕ_{s20} and $EF(mailed)_{s20}$ are resolved, which in turn depend on ϕ_{s35} and $EF(mailed)_{s35}$ to be resolved.

Thus, existing approaches do not suffice to determine whether the original formula ϕ holds at $s10$ of the Email product when there is an inter-feature cycle. For example, the approach in Blundell et al. (2004) explicitly requires that the graph of connections between features must form a DAG.

The approach described in this paper handles this cyclic dependency, since, given the above obligations generated and stored in the Answer Set, the INTERP algorithm detects this cyclic dependency and reduces both ϕ_{s10} and $EF(mailed)_{s10}$ to `tt`. Since some of the variations in the real-world product lines that we propose to formally verify, contain inter-feature cycles (involving loops over composed state-space), have a sound and complete technique, such as that described in this paper, is needed for the practical application of model checking techniques in product lines.

9 Conclusion

This paper presented an incremental and compositional model checking technique for performing sequential composition of features in a product line. The resulting composition was shown to be sound and complete. This technique generates obligations at the variation points such that the feature composition satisfies the desired property if and only if the features added at variation points satisfy the corresponding obligations. By computing and managing variation point obligations, we enable reuse of previous verification results when a new product is composed. Re-checking occurs only when and as needed.

Additionally, this approach removes restrictions on how features can be sequentially composed, providing more flexibility in how features interact than existing techniques and bringing models more in line with real-world product lines. The technique accommodates product-line evolution by identifying obligations at these new variation points from previous obligations computed at those points. Evaluation done using a prototype implementation to model check a simplified pacemaker product line showed reduction in the amount of re-verification needed to assure that a property holds for each new product in the product line.

Acknowledgements This research was supported by the National Science Foundation under grants 0541163, 0702758, 0709217 and 0916275. The authors would like to thank members of the IFIP Working Group 2.9, Software Requirements Engineering, for valuable feedback on a presentation describing this work.

References

Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. **17**(3), 507–535 (1995). http://doi.acm.org/10.1145/203095.201069

Basu, S., Ramakrishnan, C.R.: Compositional analysis for verification of parameterized systems. Theor. Comput. Sci. **354**(2), 211–229 (2006). doi:10.1016/j.tcs.2005.11.016

Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: challenges ahead. Commun. ACM **49**(12), 45–47 (2006). http://doi.acm.org/10.1145/1183236.1183264

Berezin, S., Campos, S., Clarke, E.M.: Compositional reasoning in model checking. In: Proc. COMPOS, pp. 81–102. Springer, Berlin (1998)

Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: A constraint-based approach to open feature verification. Tech. Rep. CS-03-07, Department of Computer Science, Brown University (2003)

Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: Parameterized interfaces for open system verification of product lines. In: ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 258–267. IEEE Comput. Soc., Washington (2004). 10.1109/ASE.2004.53

Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, pp. 274–287. Springer, London (1999)

Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)

Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: LICS, pp. 353–362 (1989)

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: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 335–344. ACM, New York (2010). http://doi.acm.org/10.1145/1806799.1806850

Cleaveland, R.: Tableau-based model checking in the propositional mu-calculus. Acta Inform. **27**(8), 725–748 (1990). citeseer.ist.psu.edu/61863.html

Ellenbogen, K.A., Wood, M.A.: Cardiac Pacing and ICDs, 4th edn. Blackwell, Oxford (2005)

Fantechi, A., Gnesi, S.: A behavioural model for product families. In: ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 521–524. ACM, New York (2007). http://doi.acm.org/10.1145/1287624.1287700

Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: ROSATEA '06: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 39–48. ACM, New York (2006). http://doi.acm.org/10.1145/1147249.1147254

Fisler, K., Krishnamurthi, S.: Modular verification of collaboration-based software designs. In: ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 152–163. ACM, New York (2001). http://doi.acm.org/10.1145/503209.503231

Fisler, K., Krishnamurthi, S.: Modular verification of feature-oriented software models. Tech. Rep. WPI-CS-TR-02-28, Department of Computer Science, WPI (2002). http://web.cs.wpi.edu/kfisler/Pubs/tr-02-28.pdf

Gheorghiu, M., Giannakopoulou, D., Pasareanu, C.S.: Refining interface alphabets for compositional verification. In: TACAS, pp. 292–307 (2007)

Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, pp. 3–12. IEEE Comput. Soc., Washington (2002)

Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: FMOODS '08: Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 113–131. Springer, Berlin (2008). 10.1007/978-3-540-68863-1_8

Hall, R.: Feature interactions in electronic mail. In: Feature Interactions in Telecommunications Systems, pp. 67–82. IOS Press, Amsterdam (2000)

Havelund, K., Lowry, M.R., Penix, J.: Formal analysis of a space-craft controller using SPIN. Softw. Eng. **27**(8), 1000–9999 (2001). citeseer.ist.psu.edu/havelund98formal.html

Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University Press, Cambridge (2004). http://pubs.doc.ic.ac.uk/logic-computer-science-second/

Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, Reading (1997)

Kaivola, R.: Formal verification of Pentium 4 components with symbolic simulation and inductive invariants. In: Etessami, K., Rajamani, S.K. (eds.) CAV. Lecture Notes in Computer Science, vol. 3576, pp. 170–184. Springer, Berlin (2005)

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)

Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Softw. **19**(4), 58–65 (2002). http://doi.ieeecomputersociety.org/10.1109/MS.2002.1020288

Kishi, T., Noda, N., Katayama, T.: Design verification for product line development. In: SPLC, pp. 150–161 (2005)

Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. **16**(2), 7 (2007). http://doi.acm.org/10.1145/1217295.1217296

Larsen, K.G., Nyman, U., Wasowski, A.: Modal i/o automata for interface and product line theories. In: ESOP'07: Proceedings of the 16th European Conference on Programming, pp. 64–79. Springer, Berlin (2007)

Lauenroth, K., Pohl, K.: Towards automated consistency checks of product line requirements specifications. In: ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 373–376. ACM, New York (2007). http://doi.acm.org/10.1145/1321631.1321687

Li, H., Krishnamurthi, S., Fisler, K.: Verifying cross-cutting features as open systems. In: SIGSOFT '02/FSE-10: Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 89–98. ACM, New York (2002a). http://doi.acm.org/10.1145/587051.587066

Li, H.C., Krishnamurthi, S., Fisler, K.: Interfaces for modular feature verification. In: ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, p. 195. IEEE Comput. Soc., Washington (2002b)

Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features using three-valued model checking. Autom. Softw. Eng. **12**(3), 349–382 (2005). 10.1007/s10515-005-2643-9

Littlewood, B., Strigini, L.: Validation of ultrahigh dependability for software-based systems. Commun. ACM **36**(11), 69–80 (1993). http://doi.acm.org/10.1145/163359.163373

Liu, J.: Research prototype model checker for compositional model checking of software product lines (2010). http://www.cs.iastate.edu/~janetlj/PrototypeModelChecker/

Liu, J., Dehlinger, J., Lutz, R.: Safety analysis of software product lines using state-based modeling. J. Syst. Softw. **80**(11), 1879–1892 (2007a). 10.1016/j.jss.2007.01.047

Liu, J., Dehlinger, J., Sun, H., Lutz, R.: State-based modeling to support the evolution and maintenance of safety-critical software product lines. In: ECBS '07: Proceedings of the Fourteenth Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 596–608. IEEE Comput. Soc., Washington (2007b). 10.1109/ECBS.2007.66

Liu, J., Hauptman, M., Lutz, R., Geppert, B., Rossler, F.: A tool-supported technique for specification & management of model-checking properties for software product lines. Tech. Rep. 08-05, Department of Computer Science, Iowa State University (2008). http://archives.cs.iastate.edu

Nejati, S., Sabetzadeh, M., Chechik, M., Uchitel, S., Zave, P.: Towards compositional synthesis of evolving systems. In: International Symposium on Foundations of Software Engineering (2008). http://publicaciones.dc.uba.ar/Publications/2008/NSCUZ08

Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer Academic, Norwell (2000). citeseer.ist.psu.edu/ossher00multidimensional.html

Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)

Queille, J., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Proceedings of the International Symposium in Programming (1982)

Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley, New York (1998)

Thang, N.T.: Incremental verification of consistency in feature-oriented software. PhD Thesis, Japan Advanced Institute of Science and Technology (2005)

Wang, X.: A modular model checking algorithm for cyclic feature compositions. Master's Thesis, Worcester Polytechnic Institute (2005)

Webber, D.L., Gomaa, H.: Modeling variability in software product lines with the variation point model. Sci. Comput. Program. **53**(3), 305–331 (2004). 10.1016/j.scico.2003.04.004

Weiss, D.M., Lai, R.: Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Reading (1999)

Xie, F., Browne, J.C.: Verified systems by composition from verified components. SIGSOFT Softw. Eng. Notes **28**(5), 277–286 (2003). http://doi.acm.org/10.1145/949952.940109

Zave, P.: Feature interactions and formal specifications in telecommunications. Computer **26**(8), 20–29 (1993). 10.1109/2.223539