Iowa State University
Digital Repository

2008

Generating Variation-point Obligations for Compositional Model Checking of Software Product Lines

Jing (Janet) Liu
Iowa State University

Samik Basu
Iowa State University, sbasu@iastate.edu

Robyn R. Lutz
Iowa State University, rlutz@iastate.edu

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Software Engineering Commons

Recommended Citation

Liu, Jing (Janet); Basu, Samik; and Lutz, Robyn R., "Generating Variation-point Obligations for Compositional Model Checking of Software Product Lines" (2008). *Computer Science Technical Reports*. 334.
http://lib.dr.iastate.edu/cs_techreports/334

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Generating Variation-point Obligations for Compositional Model Checking of Software Product Lines

Jing (Janet) Liu Samik Basu
Department of Computer Science
Iowa State University
Ames, IA, 50010
{janetlj, sbasu}@cs.iastate.edu

Robyn Lutz
Department of Computer Science
Iowa State University & JPL/Caltech
Ames, IA, 50010
rlutz@cs.iastate.edu

Abstract

Software product lines are widely used due to their advantageous reuse of shared features while still allowing optional and alternative features in the individual products. Especially for high-integrity product lines, we would like to use model checking to verify that key properties hold as each new product is built. However, this goal is currently hampered by the complexity of composing model-checking results for the features in a way that allows reuse for subsequent products. This paper presents an incremental and compositional model-checking technique that allows efficient reuse of model checking results associated with the features in a product line. It goes beyond related work in that it removes restrictions on how the features can be sequentially composed. This flexibility is important because it means that many more real-world systems can be model-checked. We have implemented the technique, and demonstrate and evaluate it on a medical device product line.

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, 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 [20]. Each feature carries an increment of functionality for the system [2]. Typically, the set of variations are selected and composed on top of the common base features to create each distinct, new product. The locations in the features (usually the common features) [11, 19] where other features can be added (usually the variable features) to construct the various products are called *variation points*.

Model checking [4, 10] 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 [9, 12]. As such, model checking can play a vital role in verifying key properties of products in high-integrity product lines such as pacemakes, medical imaging systems, and avionics control systems.

However, 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 will 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.

This paper presents an incremental and compositional model-checking technique that allows effective reuse of model checking results associated with the features in a product line. The contribution of the paper is that, in contrast with existing work on compositional model checking of features [3, 17, 18], we impose no restrictions (e.g., regarding the sequence, type of connection points, or number of connections) on how the features can be composed. Any type of sequential composition of features, not just pipelined composition, can be verified. Similarly, behavior of a feature that depends on another feature's behavior (which may, in turn, depend on the first feature) can be verified.

To achieve these extensions, our 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. These *variation-point obligations* guide the verification of features subsequently composed at the variation points as new products in the product line are built.

By allowing more kinds of interactions between features, our approach provides three important advantages for model-checking product lines. First, such flexibility means that many more real-world systems can be model-checked. This moves model checking closer to product-line development practice. Second, 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, providing savings in space and time over non-compositional model checking. Third, 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. We have implemented the technique, and demonstrate and evaluate it on a medical device product line.

The rest of the paper is organized as follows. Section 2 provides a motivating example. Section 3 presents the preliminary information of this work. Section 4 gives an overview of this approach, and Section 5 illustrates each step in more detail. Section 6 demonstrates our technique on a simplified pacemaker product line and discusses test results. Finally, Section 7 describes related work, and Section 8 offers concluding remarks.

2 Illustrative Example

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 evaluate performance of our approach (Sect. 6). A pacemaker [5] is an embedded medical device designed to monitor and regulate the beating of the heart 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 [5, 14]. Figure 1 shows four products in the pacemaker product line [15, 16]:

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

ModeTransitivePacemaker has an additional feature called ModeTransition Extension that enables it to switch between Inhibited Mode and TriggeredMode during execution. In the TriggeredMode, a pulse follows every heartbeat to regulate the heartbeat.

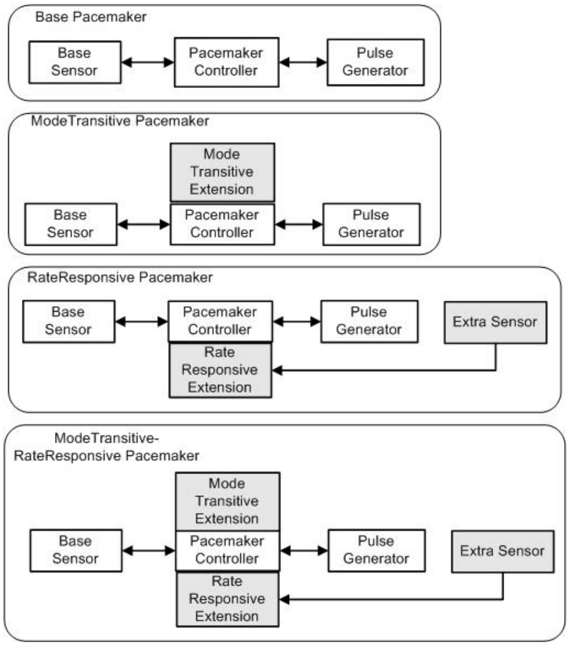


Figure 1. Pacemaker product line overview.

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 sensing interval (to normal or upperRateLimit) according to the patient's current activity level.

ModeTransitive-RateResponsivePacemaker combines the features of the ModeTransitivePacemaker and the RateResponsivePacemaker to provide both inhibited and triggered heartbeat regulation and adaptation to 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 unnecessary, repeated checking of the same feature as each new product is built. 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 variation points, the model-checking effort is aligned with the inherent variation points that a product-line development approach provides.

3 Preliminaries

We represent the functional behavior of features in a product line 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. Formally, the model is defined as follows:

Definition 1 (Feature Behavioral Model)**.** *Feature behavioral model* $\mathit{FM} = (S, S_0, V, T, L)$ *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 where one $\mathtt{FM}$ can be plugged into another, i.e., when two $\mathtt{FM}$s 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 point 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 $\mathtt{FM}_1$. We define sequential composition as follows:

Definition 2 (Sequential Composition)**.** *Given* $\mathit{FM}_1 = (S^1, S_0^1, V^1, T^1, L^1)$, $\mathit{FM}_2 = (S^2, S_0^2, V^2, T^2, L^2)$, $T_c^{\mathit{FM}_1,\mathit{FM}_2}$, *and* $T_c^{\mathit{FM}_2,\mathit{FM}_1}$, *the sequential composition* $\mathit{Comp}_{seq}(\mathit{FM}_1, \mathit{FM}_2) = (S^1 \cup S^2, S_0^1, V^{12}, T^{12}, L^{12})$,

1. $V^{12} = \{s \mid s \in V^1 \cup V^2\}$,
2. $T^{12} = T^1 \cup T^2 \cup T_c^{\mathit{FM}_1,\mathit{FM}_2} \cup T_c^{\mathit{FM}_2,\mathit{FM}_1}$, *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. $\mathtt{FM}_1$ is the start feature at whose start states (S_0^1) we want a given property to be satisfied. The set of variation points V^{12} of $\mathit{Comp}_{seq}(\mathtt{FM}_1, \mathtt{FM}_2)$ 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* $\mathtt{FM}$ is one which does not have any variation points ($V = \emptyset$). In other words, a closed $\mathtt{FM}$ cannot be augmented with new features. An open $\mathtt{FM}$ is one whose set of variation points is non-empty.

Temporal Logic CTL. Properties, in our setting, are described using CTL temporal logic [10]. 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 ($\mathtt{FM}$) that satisfy the formulas. The propositional constant $\mathtt{tt}$ is satisfied in all states while $\mathtt{ff}$ is not satisfied by any state. 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 model checking closed systems see [10].

4 Method Overview

As noted in Section 3, a closed $\mathtt{FM}$ can be verified to check whether or not it satisfies a desired CTL property. However, for an open $\mathtt{FM}$ such as ours, satisfiability of CTL properties may depend on the behavior of the features being connected to those variation points.

Given an $\mathtt{FM}$ and a desired property for its possible compositions with other $\mathtt{FM}$s, 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 $\mathit{Comp}_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \cdots, \mathtt{FM}_m)$ satisfies a CTL formula φ amounts to checking whether all the start states of $\mathtt{FM}_1$ satisfies 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 $\texttt{FM}_k$, such that t_k satisfies φ_i. If t_k satisfies its obligation φ_i, then update the `aSet` entry for s_i in $\texttt{FM}_i$.

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.

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

5 Detailed Approach

In this section, we describe the generation of variation-point obligations, how to update the answer set to identify inter-feature loops, and our algorithm for compositional model checking.

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. The obligations are also 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* $\texttt{FM} = (S, S_0, V, T, L)$*, 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,* $s \in V$*,* $op \in \{\vee, \wedge, \bot\}$*.* □

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: $(\texttt{tt}, \epsilon, \bot) \vee \psi \equiv (\texttt{tt}, \epsilon, \bot)$; $(\texttt{tt}, \epsilon, \bot) \wedge \psi \equiv \psi$; $\neg(\texttt{tt}, \epsilon, \bot) \equiv (\texttt{ff}, \epsilon, \bot)$; $(\texttt{ff}, \epsilon, \bot) \wedge \psi \equiv (\texttt{ff}, \epsilon, \bot)$; $(\texttt{ff}, \epsilon, \bot) \vee \psi \equiv \psi$; $\neg(\texttt{ff}, \epsilon, \bot) \equiv (\texttt{tt}, \epsilon, \bot)$; $\neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2$; $\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2$. 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 and on-the-fly CTL model checking [10] 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: $\texttt{E}(\varphi_1 \, \texttt{U} \, \varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \texttt{EX}(\texttt{E}(\varphi_1 \, \texttt{U} \, \varphi_2))$; $\texttt{EG}(\varphi) \equiv \varphi \wedge \texttt{EX}(\texttt{EG}(\varphi))$.

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 s; s, the current state of the `FM`; H, the history set recording the state-formula pairs that have been visited in the recursive definition 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 $(\varphi, s) \mapsto \psi$ 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. $(\varphi, s) \mapsto (\varphi', s', op')$ 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. $(\varphi, s) \mapsto (pc, \epsilon, \bot)$ denotes that s satisfies (does not satisfy) φ when pc is a propositional constant equal to `tt` (resp. `ff`)
3. $(\varphi, s) \mapsto \psi_1 \wedge \psi_2$ denotes that s satisfies φ if both ψ_1 and ψ_2 are satisfied. Similarly, $(\varphi, s) \mapsto \psi_1 \vee \psi_2$ 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 below). It also allows us to reuse the previous results to remove redundant, repeated computations. The recursive definition of the functions `t_Obl` and `Obl` are presented in Figure 2.

Rule A 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 $(\varphi, s) \mapsto \psi'$, then the mapping for (φ, s) is updated to $\psi \; op \; \psi'$ where op is decided on the basis of the formula being universal (e.g. `AG`, `AU`) or existential (e.g. `EG`, `EU`).

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 Figure 2 state that for propositional constants and propositions, there is no obligation at the variation points; satisfiability of these types of CTL formulas

$$
\text{A.}\quad \texttt{t\_Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \begin{cases}
\psi \quad \text{if } (\varphi, s) \mapsto \psi \in \texttt{aSet}_{in}; \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\[2ex]
\texttt{Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{temp}) \quad \text{otherwise} \\
\quad \text{where } \psi := \texttt{Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{temp}) \\
\quad \texttt{aSet}_{out} := \begin{cases}
\texttt{aSet}_{temp}[(\varphi, s) \mapsto \psi' / (\varphi, s) \mapsto \psi \ op\ \psi'] \\
\quad \text{if } (\varphi, s) \mapsto \psi' \in \texttt{aSet}_{temp} \\
\quad \text{where } op := \left\{ \begin{array}{ll} \wedge & \text{if } \varphi \text{ is a universal} \\ \vee & \text{otherwise} \end{array} \right\} \\[2ex]
\texttt{aSet}_{temp} \cup \{(\varphi, s) \mapsto \psi\} \quad \text{otherwise}
\end{cases}
\end{cases}
$$

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

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

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

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

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

$$
\text{6.}\quad \texttt{Obl}(\texttt{E}(\varphi_1 \,\texttt{U}\, \varphi_2), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \begin{cases}
(\texttt{ff}, \epsilon, \bot) \quad \text{if } (\texttt{E}(\varphi_1 \,\texttt{U}\, \varphi_2), s) \in H; \ \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\
\texttt{t\_Obl}(\varphi_2 \vee (\varphi_1 \wedge \texttt{EX}(\texttt{E}(\varphi_1 \,\texttt{U}\, \varphi_2))), s, \\
\qquad H \cup \{(\texttt{E}(\varphi_1 \,\texttt{U}\, \varphi_2), s)\}, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \quad \text{otherwise}
\end{cases}
$$

$$
\text{7.}\quad \texttt{Obl}(\texttt{EG}(\varphi), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \begin{cases}
(\texttt{tt}, \epsilon, \bot) \quad \text{if } (\texttt{EG}(\varphi), s) \in H; \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\
\texttt{t\_Obl}(\varphi \wedge \texttt{EX}(\texttt{EG}(\varphi)), s, H \cup \{(\texttt{EG}(\varphi), s)\}, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \\
\hfill \text{otherwise}
\end{cases}
$$

$$
\text{8.}\quad \texttt{Obl}(\texttt{EX}(\varphi), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \bigvee_{s \to s'} \texttt{t\_Obl}(\varphi, s', H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \vee \begin{cases} (\varphi, s, \vee) & \text{if } s \in V \\ (\texttt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases}
$$

Figure 2. Variation-point obligations

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.

Rules 6 and 7 use 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 $(\texttt{ff}, \epsilon, \bot)$. For the greatest fixed point formula `EG`, the result is $(\texttt{tt}, \epsilon, \bot)$. On the other hand, if the state-formula pair is not present in the history set, the formula is expanded to its equivalent form with `t_Obl` being 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 $s \rightarrow s'$, `t_Obl` is computed using s' and φ, and the results are ORed. 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.

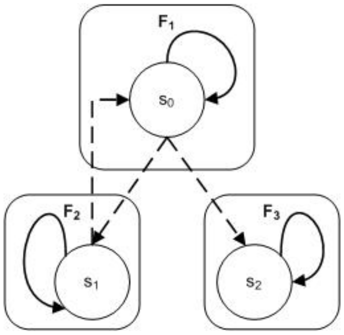


Figure 3. Feature composition example.

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. Given the CTL property* $\varphi = \texttt{E}(\texttt{tt}\ \texttt{U}\ p)$ *to verify over the three-feature composition* $Comp_{seq}(FM_1, FM_2, FM_3)$*, we firstly compute the variation-point obligation for* F_1*, shown in Figure 4. The downward arrows in Figure 4 show the invocation of the* `t_Obl` *and*

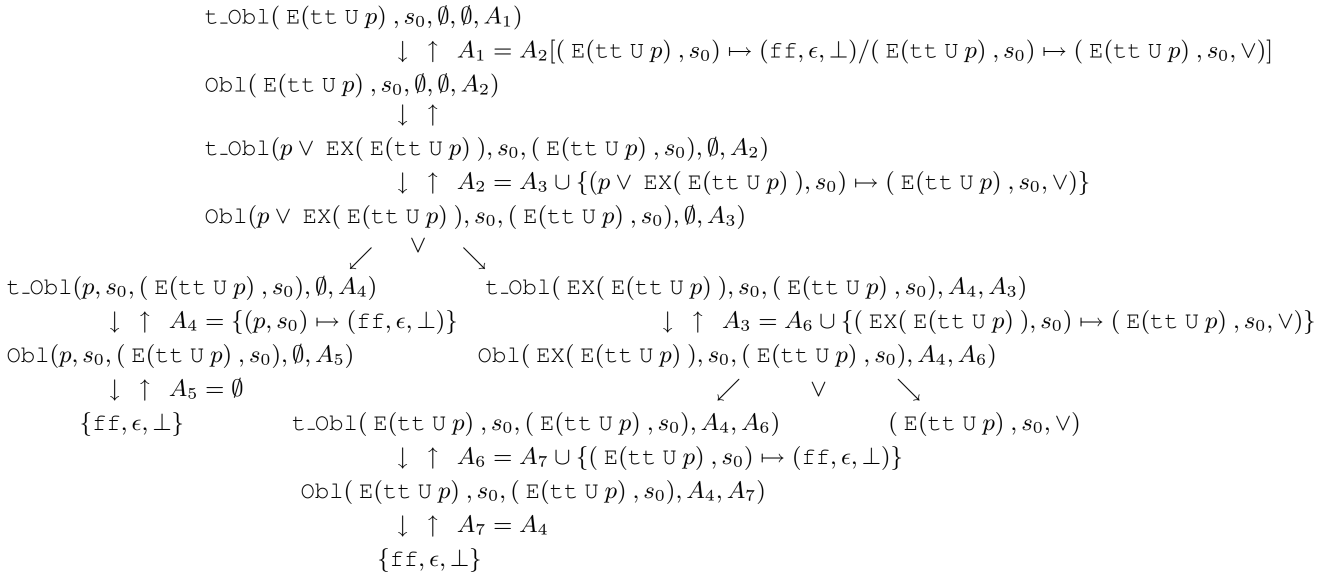


Figure 4. Example of Computing Variation-point Obligations

`Obl` functions, while the upward arrows show the updates to `aSet`. The variation-point obligations for F_2 and F_3 are computed in a similar fashion.

Step 2: Updating `aSet`. After the computation of variation-point obligation terminates for one `FM`, `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:

$$\mathtt{update}(\mathtt{aSet}) := \mathtt{aSet}[(\varphi, s) \mapsto \psi/(\varphi, s) \mapsto \psi[\psi_i/\psi_i']]$$
where $\psi_i := (\varphi_i, s_i, op_i)$ and $s_i \to t_i \in T_c^{\mathtt{FM}_m, \mathtt{FM}_n}$ and
$(\varphi_i, t_i) \mapsto (pc, \epsilon, \bot) \in \mathtt{aSet} \wedge pc \in \{\mathtt{tt}, \mathtt{ff}\}$ 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 $(\varphi, s) \mapsto \psi$ in `aSet` is updated to $(\varphi, s) \mapsto \psi[\psi_i/\psi_i']$ (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 `FM`. If the state t_i of another `FM` is connected to the variation point s_i and there exists an entry $(\varphi_i, t_i) \mapsto (pc, \epsilon, \bot)$ ($pc \in \{\mathtt{tt}, \mathtt{ff}\}$) 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 $pc = \mathtt{ff}$, 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,*
`aSet` = $\{(\varphi, s_0) \mapsto (\varphi, s_0, \vee)$,
$(p \vee EX(\varphi), s_0) \mapsto (\varphi, s_0, \vee)$, $(p, s_0) \mapsto (ff, \epsilon, \bot)$,
$(EX(\varphi), s_0) \mapsto (\varphi, s_0, \vee)$, $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$,
$(p \vee EX(\varphi), s_1) \mapsto (\varphi, s_1, \vee)$, $(p, s_1) \mapsto (ff, \epsilon, \bot)$,
$(EX(\varphi), s_1) \mapsto (\varphi, s_1, \vee)$, $(\varphi, s_2) \mapsto (ff, \epsilon, \bot)$,
$(p \vee EX(\varphi), s_2) \mapsto (ff, \epsilon, \bot)$, $(p, s_2) \mapsto (ff, \epsilon, \bot)$,
$(EX(\varphi), s_2) \mapsto (ff, \epsilon, \bot)\}$.
Applying `update` to the above `aSet` does not change any obligations.

Step 3: Identifying Inter-Feature Loops from `aSet`. To summarize, once the variation-point obligations have been computed for all `FM`s (step 1-2) and for every $(\varphi, s) \mapsto \psi$ in `aSet`, each subformula of ψ, (φ_i, s_i, op_i), has a corresponding $(\varphi_i, s_i) \mapsto \psi'$ 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 identify loops between features. An example of such a chain is the circular dependency between $\mathtt{FM}_1$ and $\mathtt{FM}_2$ in Fig 3: $(\varphi, s_0) \mapsto (\varphi, s_0, \vee)$ and $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$, where $s_0 \to s_1$ and $s_1 \to s_0$. The search for the inter-feature loops is done by applying the $\mathtt{update}_F(\mathtt{aSet})$ function on each element in `aSet`.

$$\mathtt{update}_F(\mathtt{aSet}) = \mathtt{aSet}[(\varphi, s) \mapsto \psi/(\varphi, s) \mapsto (pc, \epsilon, \bot)]$$
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. If $(\varphi, s) \mapsto (pc, \epsilon, \bot)$, then the mapping result does not depend on other

Algorithm 1 Analysis for Inter-Feature Loops

```
1:  procedure INTERP($(\varphi, s)$, $\texttt{Dep}$)
2:     if $(\varphi, s) \mapsto (pc, \epsilon, \bot)$ then
3:        return $pc$
4:     end if
5:     if $(\varphi, s) \in \texttt{Dep}$ then
6:        if $\varphi$ is gfp then
7:           return true
8:        else return false
9:        end if
10:    end if
11:    $(\varphi, s) \mapsto (\varphi_1, s_1, op) op \ldots op (\varphi_k, s_k, op) \in \texttt{aSet}$
12:    $\texttt{Next} := \bigcup_{1 \le i \le k} \{(\varphi_i, t_i) \mid s_i \rightarrow t_i \in T_c^{\texttt{FM}_m, \texttt{FM}_n}\}$
13:    if ($op = \wedge$) res = $\texttt{tt}$ else $\texttt{res}$ = $\texttt{ff}$
14:    for $(\varphi', t') \in \texttt{Next}$ do
15:       $\texttt{res}$ = $\texttt{res}$ op INTERP($(\varphi', t'), \texttt{Dep} \cup \{(\varphi, s)\}$)
16:    end for
17:    return $\texttt{res}$
18: end procedure
```

elements, and the procedure immediately returns *pc*. In this case the answer has been resolved. Otherwise (Lines 5–10), if the mapping result is in the set `Dep`, 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 (similar to the way we detected intra-feature loops during variation-point obligation computation using the history set, see Figure 2). Finally, in Lines 11-17, the algorithm computes the dependency of results across features. This is firstly performed in Lines 11 and 12, where for each (φ_i, s_i, op), t_i (the state connected to s_i) is identified and collected in the set `Next`. Then INTERP is recursively invoked on each element of `Next` and the result is aggregated based on whether *op* is $\wedge$ or $\vee$ (Line 13–17).

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:* $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$*, we compute*
INTERP$((\varphi, s_1), \emptyset)$ = $\texttt{ff} \vee$ INTERP$((\varphi, s_0), \{(\varphi, s_1)\})$
= $\texttt{ff} \vee$ INTERP$((\varphi, s_1), \{(\varphi, s_1), (\varphi, s_0)\})$ = $\texttt{ff}$.
We then replace $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$ *with* $(\varphi, s_1) \mapsto (\texttt{ff}, \epsilon, \bot)$ *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.*

Summary: Compositional Algorithm. Algorithm 2 presents our compositional algorithm introduced in Section 4 using the functions described above. This algorithm model checks the current product (a composition of features), reusing results from previous model-checking of those features for other products in the product line.

Algorithm 2 Compositional Model Checking

```
1:  procedure COMPOSE($Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m), \varphi$)
2:     $\bigwedge_{s_0 \in S_0^1} \texttt{t\_Obl}(\varphi, s_0, \emptyset, \emptyset, \texttt{aSet})$
3:     repeat
4:        $tmp := \texttt{aSet}$
5:        for each $((\varphi', s') \mapsto \psi) \wedge (\psi \neq (pc, \epsilon, \bot))$ do
6:           $\psi := \texttt{getObl}(\psi, \texttt{aSet}, \texttt{aSet}_{\texttt{new}})$
7:           $\texttt{aSet:=update}(\texttt{aSet}_{\texttt{new}})$
8:        end for
9:     until ($\texttt{aSet} = tmp$)
10:    return $\texttt{update}_F(\texttt{aSet})$
11: end procedure
```

To summarize, given a composition $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m)$ and a formula φ, the algorithm first obtains the variation-point obligations for $\texttt{FM}_1$ such that all its start states can satisfy φ (Line 2). In Lines 5 and 6, the variation-point obligations of the other features connected to the variation points of $\texttt{FM}_1$ are computed using the function `getObl`, defined as follows:

$$\begin{aligned}
&\texttt{getObl}((\varphi, s, op), \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \\
&\texttt{Op}_{s \rightarrow s'}[\texttt{t\_Obl}(\varphi, t, \emptyset, \texttt{aSet}_{in}, \texttt{aSet}_{out})] \\
&\text{where } s \rightarrow t \in T_c^{\texttt{FM}_m, \texttt{FM}_n}, op \in \{\vee, \wedge\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{getObl}(\psi_1 \; op \; \psi_2, \texttt{aSet}_{in}, \texttt{aSet}_{out}) := \\
&\quad \texttt{getObl}(\psi_1, \texttt{aSet}_{in}, \texttt{aSet}_{temp}) \; op \\
&\quad\quad \texttt{getObl}(\psi_2, \texttt{aSet}_{temp}, \texttt{aSet}_{out})
\end{aligned}$$

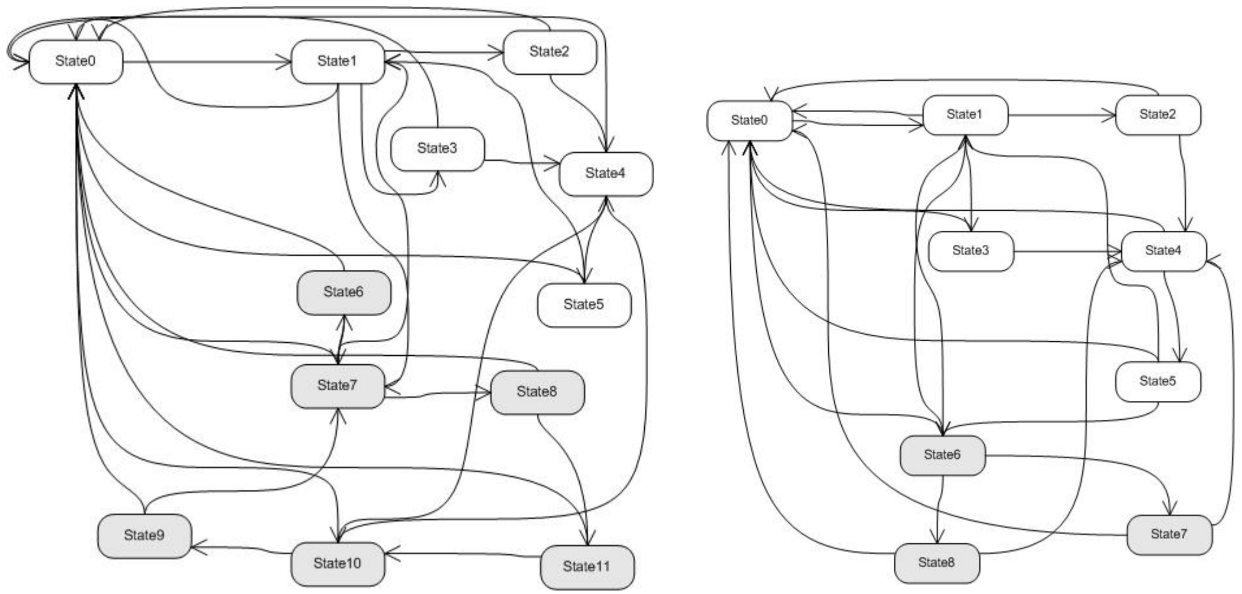
The process of computing the variation-point obligation is iterated (Line 3–9) until no more updates on the `aSet` can be made (Line 9). At this point, the function $\texttt{update}_F(\texttt{aSet})$ is invoked to identify loops between features and infer results from variation-point obligations represented in greatest and least fixed point formulas in the `aSet`. We say that $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m) \models \varphi$ when for all start states s_0 of $\texttt{FM}_1$, $((\varphi, s_0) \mapsto (\texttt{tt}, \epsilon, \bot)) \in \texttt{update}_F(\texttt{aSet})$. At this point, satisfaction of the property of interest in the composed product has been determined. The remark below follows from the above discussion.

Remark 1. $Comp_{seq}(FM_1, FM_2, \cdots, FM_m) \models \varphi \Leftrightarrow \forall s_0 \in S_0^1 : (\varphi, s_0) \mapsto (\texttt{tt}, \epsilon, \bot) \in$ *COMPOSE*$(Comp_{seq}(FM_1, FM_2, \cdots, FM_m), \varphi)$.

Example. *So far, we have performed every step of Algorithm 2. Since* $(\varphi, s_0) \mapsto (\texttt{ff}, \epsilon, \bot) \in \texttt{update}_F(\texttt{aSet})$*, the verification result is that* $Comp_{seq}(FM_1, FM_2, FM_3) \not\models \varphi$.

6 Case Study

We have implemented our algorithm in a research prototype model checker. Detailed information is provided in the following link: http://www.cs.iastate.edu/∼janetlj/ase08/.



State	True Propositions
s_0	timerOff=1
s_1	sensorOn=1; inhibitedMode=1;
s_2	sensed=1; sensorOn=1; inhibitedMode=1;
s_3	timerSenseTimeUp=1; pulseGen=1; inhibitedMode=1;
s_4	inhibitedMode=1;
s_5	timerRefractoryTimeUp=1; sensorOn=1; inhibitedMode=1;

State	True Propositions
s_6	timerSenseTimeUp=1; sensorOn=1; triggeredMode=1;
s_7	sensorOn=1; triggeredMode=1;
s_8	sensed=1; sensorOn=1; triggeredMode=1;
s_9	timerRefractoryTimeUp=1; sensorOn=1; triggeredMode=1;
s_{10}	triggeredMode=1;
s_{11}	pulseGen=1; triggeredMode=1;

(a)

State	True Propositions
s_6	sensorOn=1; inhibitedMode=1; upperRateLimit=1;
s_7	sensorOn=1; sensed=1; inhibitedMode=1; upperRateLimit=1;
s_8	timerShortSenseTimeUp=1; pulseGen=1; inhibitedMode=1; upperRateLimit=1;

(b)

Figure 5. (a) Base Controller and Mode-Transitive Extension. (b) Base Controller and Rate-Responsive Controller Extension.

We evaluated our technique by conducting experiments on the pacemaker product line discussed in Section 2. Figures 5(a) and (b) depict how the Mode Transitive Extension ($\mathtt{FM}_1$) and Rate Responsive Extension ($\mathtt{FM}_2$) are sequentially composed with 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.

It is worth mentioning that although the mode-transitive feature in Fig. 5(a) extends the base controller functionality in a sequential manner, the rate-responsive feature does so in a more complicated manner. This is because it also introduces a new component (extra sensor). We model its effect on the extension to the base controller's functionality by introducing an abstract event in the extension ("`upperRateLimit=1`" in Figure 5(b)'s Table). This modeling strategy works because the CTL property to be verified does not depend on the behavior of the extra sensor.

In the controller for the fourth product (not shown), ModeTransitive-RateResponsive Pacemaker, $\mathtt{FM}_1$ and $\mathtt{FM}_2$ are sequentially composed with $\mathtt{FM}_0$, as in Figure 5 (a) and (b). $\mathtt{FM}_1$ and $\mathtt{FM}_2$ do not connect directly; 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 textually introduced in Section 2:

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

This formula was used in the following evaluation.

To evaluate the space and time performance of our approach, we compared compositional model checking (CMC) with non-compositional model checking (NMC) for the four products in the product line and recorded the experimental results in Table 1. In the table, MT denotes ModeTransitive, while RR denotes RateResponsive. (`t_Obl`) and (`Obl`) record the number of times the `t_Obl`

Table 1. Test Data for Pacemaker Product Line

	Non-Compositional Model Checking (NMC)				Compositional Model Checking (CMC)			
# of Invocations	Base Pacemaker	MT Pacemaker	RR Pacemaker	MT-RR Pacemaker	Base Pacemaker	MT Pacemaker	RR Pacemaker	MT-RR Pacemaker
(`t_Obl`)	33	68	52	87	40	51	30	28
(`Obl`)	20	38	29	47	22	22	12	5
$(\texttt{Obl}_{\texttt{top\_test}})$	65	125	94	154	71	76	39	20
$(\texttt{Obl}_{\texttt{test}})$	64	124	93	153	66	65	32	6
$(\texttt{add}_{obl})$	91	179	135	222	102	126	72	61
# of State Visits	13	30	23	40	16	23	13	14

and `Obl` functions are performed, respectively. Similarly, $(\texttt{Obl}_{\texttt{top\_test}})$ and $(\texttt{Obl}_{\texttt{test}})$ record the times these two assistant functions in our implementation have been invoked. The assistant functions serve to conduct lightweight tests (by "lightweight" we mean that they check only the current state), e.g., to see if one branch of a disjunct formula is true during obligation computation. $(\texttt{add}_{obl})$ records the number of times an obligation is added to our implementation of `aSet`. Finally, # of state visits records the number of times the states in a model are visited.

The results of NMC are obtained from checking each of the four products in its entirety, without breaking it into separate features. No variation points are specified, and the start state is always resolved to a true or false result at the end of the checking.

The results of CMC are obtained from calculating the test data for the added features and connections in each product. For example, verifying the BasePacemaker involves checking $\texttt{FM}_0$ 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 $\texttt{update}_F(\texttt{aSet})$ to get the final result. We found that, as expected, generating obligations at the variation points and performing $\texttt{update}_F(\texttt{aSet})$ introduced a slight overhead for the base product that was amortized over its subsequent reuse. Verifying the RateResponsivePacemaker involves checking the $\texttt{FM}_2$ with states 6, 7, and 8 as its variation points, plus checking the connections from $\texttt{FM}_2$ to $\texttt{FM}_0$, and applying the `update` and $\texttt{update}_F(\texttt{aSet})$ functions. Test results were similarly collected for the other two products.

Table 1 shows that the compositional model checking approach does provide savings in the product line. For example, in NMC the cost for checking the product line (measured in (`t_Obl`)) was 33+68+52+87=240, while the cost in CMC was 40+51+30+28=149. This is because the common features (e.g., $\texttt{FM}_0$) were checked repeatedly in NMC. If no prior checking for any of the features had been done, the cost for checking the RateResponsivePacemaker (measured in (`t_Obl`)) would have been 40+30=70, which is more than the value of 52 for the NMC. This difference is due to the cost of generating assets for reuse, i.e., generating obligations at the variation points and maintaining a different `aSet` (for applying the `update` and $\texttt{update}_F(\texttt{aSet})$ functions) for each composition. To summarize, as with the product-line approach itself, CMC shows savings when features are reused.

We briefly note here the suitability of this approach for *product-line evolution*. Since any state in a model can become a variation point, when structural changes to a feature model occur, the affected states can be treated as variation points. For example, if the state s that was not a variation point becomes a new variation point, the answer set elements involving s identify temporal obligations for any features composed at s. This allows us to model check whether the new variation-point obligations are preserved after the change. Because product lines routinely experience significant change over their lifetimes, the continued usefulness of previous model-checking results to the product line development contributes to the practicality of this technique.

7 Related Work

Several recent works have investigated representations of variability within a product line in behavioral models. Fantechi and Gnesi identify whether a product belongs to a product line [6]. Fischbein, Uchitel and Braberman propose a technique to determine whether a variability undermines a product-line property [7]. Lauenroth and Pohl describe how variability complicates the consistency checking of a product in the product line [13]. This line of work does not treat the common and variable functionality as equal units to be composed. Instead, they have 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 [7]. The variations that they can verify do not cover all the ones that exist in our case.

We now compare our proposed technique with techniques whose main objective is to effectively use compositional verification in a product line setting. In the context of open-system verification where features are added in a sequential fashion, our work is closely related to work by Blundell, Fisler, Krishnamurthi and Hentenryck [3]. The authors propose a framework in which interface obligations are generated as temporal properties. Our technique differs from theirs in that [3] requires interface states (here, the variation points) to be terminal states with no outgoing transitions, while our approach does not have this restriction. Secondly, we permit features to be added in ways that allow intra and inter-feature loops, a flexibility that was needed to accurately model the pacemaker product line. Wang extends [3] in [18] and allows inter-feature loops. However, that work assumes that interface states are sufficient for composing different features and does not re-explore the non-interface states. This implicitly puts a restriction on the type of inter-feature loops that can be verified.

In [17], Thang presents the necessary conditions which, when satisfied by the base and its extension feature(s), ensure that the property verification results hold before and after the base is extended with the corresponding features. Though the work allows loops between the base and the extensions, it does not provide insights into the cases where the necessary conditions are violated.

Our work falls into the category of compositional verification [1]. We use sequential composition (Def. 2) rather than parallel composition, as in, e.g., Giannakopoulou, Pasareanu and Barringer [8], because it would add unnecessary complexity to the state space and obscure the interfaces among features that we want to maintain in a product-line setting for effective reuse.

8 Conclusion

This paper presents an incremental and compositional model-checking technique for performing sequential composition of different features in a product-line setting. 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 provides more flexibility in how features interact than existing techniques, bringing models more in line with real-world product lines. Evaluation done using a prototype implementation to model check a simplified pacemaker product line shows that this technique can reduce the amount of re-verification needed to assure that a required property holds for each new product in the product line.

Acknowledgements This research was supported by the National Science Foundation under grants 0541163, 0702758 and 0709217.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[3] C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. Parameterized interfaces for open system verification of product lines. In *ASE '04*, pages 258–267, Washington, DC, USA, 2004. IEEE Computer Society.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[5] K. A. Ellenbogen and M. A. Wood. *Cardiac Pacing and ICDs (fourth edition)*. Blackwell Publishing, Inc., 2005.

[6] A. Fantechi and S. Gnesi. A behavioural model for product families. In *ESEC-FSE '07*, pages 521–524, New York, NY, USA, 2007. ACM.

[7] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06*, pages 39–48, New York, NY, USA, 2006. ACM.

[8] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE '02*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[9] K. Havelund, M. R. Lowry, and J. Penix. Formal analysis of a space-craft controller using SPIN. *TSE*, 27(8):0098–5589, 2001.

[10] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004.

[11] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[12] R. Kaivola. Formal verification of pentium 4 components with symbolic simulation and inductive invariants. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2005.

[13] K. Lauenroth and K. Pohl. Towards automated consistency checks of product line requirements specifications. In *ASE '07*, pages 373–376, New York, NY, USA, 2007. ACM.

[14] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Commun. ACM*, 36(11):69–80, 1993.

[15] J. Liu, J. Dehlinger, and R. Lutz. Safety analysis of software product lines using state-based modeling. *J. Syst. Softw.*, 80(11):1879–1892, 2007.

[16] J. Liu, J. Dehlinger, H. Sun, and R. Lutz. State-based modeling to support the evolution and maintenance of safety-critical software product lines. In *ECBS '07*, pages 596–608, Washington, DC, USA, 2007. IEEE Computer Society.

[17] N. T. Thang. *Incremental Verification of Consistency in Feature-Oriented Software*. PhD thesis, Japan Advanced Institute of Science and Technology, 2005.

[18] X. Wang. A modular model checking algorithm for cyclic feature compositions. Master's thesis, Worcester Polytechnic Institute, 2005.

[19] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.

[20] D. M. Weiss and R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.