Towards an incremental automata-based approach for software product-line model checking

Cordy, Maxime; Schobbens, Pierre; Heymans, Patrick; Legay, Axel

Published in:
ACM International Conference Proceeding Series

DOI:
10.1145/2364412.2364425

Publication date:
2012

Document Version
Peer reviewed version

Towards an Incremental Automata-Based Approach for Software Product-Line Model Checking

Maxime Cordy*,
Pierre-Yves Schobbens
PReCISE Research Center
University of Namur, Belgium.
{mcr,pys}@info.fundp.ac.be

Patrick Heymans
PReCISE Research Center,
University of Namur, Belgium.
INRIA Lille-Nord Europe –
Université Lille 1, France.
LIFL – CNRS , France.
phe@info.fundp.ac.be

Axel Legay
INRIA Rennes, France
Aalborg University, Denmark
University of Liège, Belgium.
axel.legay@inria.fr

ABSTRACT

Most model-checking algorithms are based on automata theory. For instance, determining whether or not a transition system satisfies a Linear Temporal Logic (LTL) formula requires computing strongly connected component of its transition graph. In Software Product-Line (SPL) engineering, the model checking problem is more complex due to the huge amount of software products that may compose the line. Indeed, one has to determine the exact subset of those products that do not satisfy an intended property. Efficient dedicated verification methods have been recently developed to answer this problem. However, most of them does not allow incremental verification. In this paper, we introduce an automata-based incremental approach for SPL model checking. Our method makes use of previous results to determine whether or not the addition of conservative features (*i.e.*, features that do not remove behaviour from the system) preserves the satisfaction of properties expressed in LTL. We provide a detailed description of the approach and propose algorithms that implement it. We discuss how our method can be combined with SPL dedicated verification methods, *viz.* Featured Transition Systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Theory, Verification

Keywords

Model Checking, Software Product Lines, Modularity.

*FNRS research fellow, Project FC 91490

SPLC Vol. II, September 2 - 7 2012, Salvador, Brazil

1. INTRODUCTION

Software Product-Line (SPL) engineering is a recent software engineering paradigm that aims at making the development of similar software products faster, safer and cheaper through systematic *reuse* of assets. Those reusable artefacts can take different forms, e.g., requirements, components, code, test cases, and others. A given product of the SPL is then obtained through a combination of assets at each stage of the development life cycle. Commonality and variability between the products is usually captured via the concept of *features*. In a nutshell, a *feature* can be regarded as a unit of difference between products. For instance, a feature may model the presence or absence of an optional component, an alternative behaviour, or an additional functionality. In this paper, we assume that features have no attribute and cannot be cloned. Then a given software variant of the product line can be modelled by its (unique) set of features. As we will see, this assumption is needed for our current approach to be applicable.

The verification problem in the context of SPLs is more difficult than in traditional software engineering. Indeed, since an SPL engineering process results in multiple software products, engineers have to provide solid evidence that every variant works properly and in accordance to its intended requirements. In the past years, efficient methods for SPL model checking have been designed to address this problem (see notably [11, 8, 10, 6, 2, 1]). However, only a few of them are *incremental* [11]. It means that when a new feature is added into a specific product after this one has been verified, most of these approaches do not allow the reuse previous verification results to simplify upcoming verifications. Likewise, the definition and the integration of a new feature to the whole SPL would oblige the engineer to start over the whole checking procedure. This inability clearly opposes the principles of SPL engineering, which advocates systematic reuse.

Efficient reuse, however, remains an open problem in formal verification. Li *et al.* [11] tackled it in cases where properties to check are expressed in Computation Tree Logic (CTL) [5]. Basically, they model both the base product (*i.e.* a product without optional features) and features with finite state machines and they derive conditions for the satisfaction of a CTL formula to be preserved upon the addition of one or more features to the base product. In this case, the product resulting from the addition of a feature is obtained by connecting the state machines of the feature onto

the state machine of the base product. Their method relies on the fact that algorithms for CTL model checking are based on (1) decomposition of the formula into a set of subformulae; and (2) recursive computation of the set of states satisfying those subformulae. In contrast, usual algorithms for checking ω-regular properties expressed in logics like Linear Temporal Logic (LTL) [14] and modal μ-calculus [9] are automata-based. Unlike the aforementioned CTL algorithms, automata-based methods rely on language emptiness checking rather than formula decomposition and recursive computation. The approach of Li *et al.* is thus impracticable as such. Moreover, their method is not complete because the non-satisfaction of the preservation conditions does not imply the violation of the formula. Given that LTL and CTL have incomparable expressiveness, there is thus a need for incremental approaches to LTL model checking.

In [15], Thang extends the approach of Li *et al.* so that it becomes complete. This is achieved through the definition of sufficient conditions for the preservation to hold. In the same vein, Liu *et al.* [12] propose a sound and complete approach for incremental CTL model checking. Their algorithm is based on variation points obligations, *i.e.*, sets of formulae that connected states of the system have to satisfy to ensure the preservation of a formula upon the integration of a new feature. Unlike the previous one, this method uses new CTL model checking algorithms and therefore cannot be built on top of existing model checkers.

Although it considers adaptive systems, the work of Zhang *et al.* [17] is comparable to ours. Their modular LTL verification algorithms could be adapted to determine the preservation of formula upon the connection of the system with a feature's state machine. The resulting algorithms could be built on top of existing model checkers. Unfortunately, they would not benefit from previous verification results.

In this paper, we propose an automata-based approach for determining if the addition of a feature to a system makes it violate an LTL formula it previously satisfied. More particularly, we focus here on *conservative* features as defined in [7], that is, features that do not remove existing behaviour of the system. As shown in this paper, the effect of conservative features on systems behaviour can be reduced to the addition of states and transitions. This allows us to derive necessary *and* sufficient conditions for the preservation of LTL formulae. The extension of this method to larger classes of features is more difficult and left for future work. Our approach does not visit the entire state space of the new system and is thus potentially more efficient than a new verification performed from scratch. Also, the implementation of our method makes use of traditional LTL model-checking algorithms and can thus be built on top of existing tools.

Structure. The rest of the paper is organised as follows. Section 2 recapitulates the theoretical background needed for a thorough understanding of our approach. In Section 3, we present how we model a system, a conservative feature, as well as the result of integrating a conservative feature into the system. Next, Section 4 describes the conditions needed to ensure that the addition of a conservative feature preserves the satisfaction of a property, whereas Section 5 presents algorithms to check if these conditions are met.

2. BACKGROUND

We first summarize the principles of LTL model checking for systems modelled as transition systems. Then, we focus on the definition of conservative feature as it has been introduced in our previous work [7].

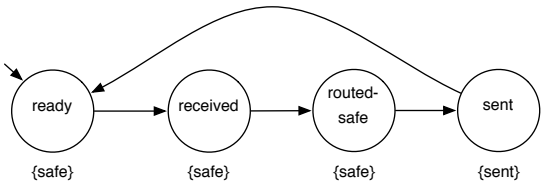

Figure 1: A TS modelling a routing protocol.

2.1 Automata-Based Model Checking

Transition systems (TSs) are commonly used to model the behaviour of systems. These are defined as follows.

Definition 1 *A TS is a tuple $\mathcal{M} = (S, Act, \rightarrow, I, AP, L)$, where S is a set of states, Act a set of actions, $\rightarrow \subseteq S \times Act \times S$ the transition relation, $I \subseteq S$ a set of initial states, AP a set of atomic propositions, and $L : S \rightarrow \mathcal{P}(AP)$ function that labels each state with the set of atomic propositions it satisfies.*

An *execution* of $\mathcal{M}$ is a non-empty, infinite sequence $\pi = s_0\alpha_1 s_1 \alpha_2 \ldots$ where $s_0 \in I$ and $(s_i, \alpha_{i+1}, s_{i+1}) \in \rightarrow$ (also noted $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$) for all $0 \leq i$. The semantics of an execution is given as the sequence of sets of atomic propositions satisfied at each step of the execution. The semantics of a TS $\mathcal{M}$ (also called the behaviour of $\mathcal{M}$), written $[\![\mathcal{M}]\!]$, is then the union of the semantics of its executions.

An example of TS is given in Figure 1. This small transition system depicts a simple packet routing protocol. The system starts in `ready` state. Once it receives a packet, it enters `received` state before routing the packet through a reliable channel (`routed-safe` state) and reaching `sent` state. Finally, it goes back to `ready` state.

In this paper, we focus on verifying the behaviour of a TS against LTL formulae. Vardi and Wolper [16] have proven that such a verification reduces to detecting accepting cycles in a Büchi automaton.

Definition 2 *A Büchi automaton (BA) is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where Q is a set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ the transition relation, $Q_0 \subseteq Q$ a set of initial states and $F \subseteq Q$ a set of accepting states. A BA accepts the words that visit accepting states infinitely often.*

The language of a BA is thus a set of infinite words. The model checking algorithm proposed by Vardi and Wolper relies on the fact that a TS $\mathcal{M}$ satisfies an ω-regular property expressed as an LTL formula φ, noted $\mathcal{M} \models \varphi$ if and only if there exists no infinite word accepted by both $\mathcal{M}$ and the BA representing the complement of φ, that is, $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset$. This last equation holds if and only if the language of the synchronous product of $\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$ is empty.

Definition 3 *The synchronous product of a TS $\mathcal{M} = (S, Act, \rightarrow, I, AP, L)$ and a BA $\mathcal{B} = (Q, \mathcal{P}(AP), \delta, Q_0, F)$ is the BA $\mathcal{M} \otimes \mathcal{B} = (S \times Q, Act, \delta', I \times Q_0, S \times F)$ where $(s, q, \alpha, s', q') \in \delta' \Leftrightarrow s \xrightarrow{\alpha} s' \wedge (q, L(s), q') \in \delta$.*

Formally, the non-emptiness condition in a BA (Q, $\mathcal{P}(AP)$, δ, Q_0, F) is given by

$$\exists i \in I \bullet \exists f \in F \bullet \exists (w,v) \in Q^* \times Q^+ \bullet f \in \delta^*(i,w) \cap \delta^*(f,v) \quad (1)$$

where $\delta^*(s,w)$ is the set of states that can be reached from s by reading the word w. We can thus determine if the language of the synchronous product is empty by first computing the set of reachable accepting states and, for each such accepting state s, checking if there exists a non-empty path from s to itself.

2.2 Conservative Features

Variability as represented by features can take a wide variety of forms. The presence of a feature may modify the architecture of the software (by adding or removing components), enable or disable functionalities, or even alter the behaviour of the system in particular situations spread through its whole execution. Let $\mathcal{M}$ be a TS modelling the behaviour of a software product p. The activation of a feature f in p yields a new product $p \oplus f$ whose behaviour is modelled by a TS $\mathcal{M}'$. Given the wide variety of effects a feature can have on the system, there is in general no relation between $\mathcal{M}$ and $\mathcal{M}'$. It is thus particularly difficult to analyse the behavioural impact of features incrementally. For example, if a feature modifies the type of acknowledgement used in a routing protocol, it is difficult to assess whether or not the new type of acknowledgement has drawbacks that the old one did not without in-depth analyses.

However, when a considered feature satisfies specific conditions, we can define a preorder relation between the behaviour of $\mathcal{M}$ and the behaviour of $\mathcal{M}'$. In this paper, we consider features that do not remove existing behaviour from the product to which they are added. These are called *conservative features* [4, 7]. In this specific case, $\mathcal{M}'$ can reproduce any behaviour in $\mathcal{M}$. One can formally characterize this behavioural inclusion via the notion of *simulation* [13].

Definition 4 *Let $\mathcal{M}_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i), i \in \{1,2\}$ be transition systems over AP. A simulation for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that*

1. $\forall s_1 \in I_1 \bullet \exists s_2 \in I_2 \bullet (s_1, s_2) \in \mathcal{R}$ *and*
2. $\forall (s_1, s_2) \in \mathcal{R}$ *it holds that*
   (a) $L_1(s_1) = L_2(s_2)$ *and*
   (b) $\forall s_1' \in Post(s_1) \bullet \exists s_2' \in Post(s_2) \bullet (s_1', s_2') \in \mathcal{R}$.

where $Post(s) = \{s_2 \mid \exists \alpha \bullet s \xrightarrow{\alpha} s_2)\}$ denotes the set of states that can be reached from s in one transition. Then, $\mathcal{M}_2$ simulates $\mathcal{M}_1$, denoted by $\mathcal{M}_1 \preccurlyeq \mathcal{M}_2$ if there exists a simulation for $(\mathcal{M}_1, \mathcal{M}_2)$. If $\mathcal{M}_2 \preccurlyeq \mathcal{M}_1$ also holds then $\mathcal{M}_1$ and $\mathcal{M}_2$ are called simulation-equivalent, noted $\mathcal{M}_1 \simeq \mathcal{M}_2$.

Accordingly, if f is a conservative feature then $\mathcal{M} \preccurlyeq \mathcal{M}'$. Given that this relation holds between the two TS, one can infer properties about $\mathcal{M}'$ knowing the properties of $\mathcal{M}$. In particular, if $\mathcal{M}$ violates an LTL formula φ then so does $\mathcal{M}'$ [13, 3]. If we transpose this result in the context of product lines, we obtain the following.

Property 5 *Let f be a conservative feature, $\mathcal{M}$ the TS modelling the behaviour of a product p and $\mathcal{M}'$ the TS modelling $p \oplus f$. Then for any LTL formula φ, $\mathcal{M} \not\models \varphi \Rightarrow \mathcal{M}' \not\models \varphi$.*

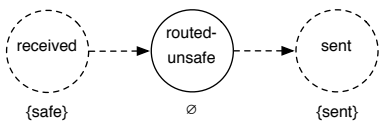

Figure 2: A TS$^+$ modelling a feature of the routing protocol.

Thus, the LTL formulae violated by $\mathcal{M}$ do not need to be verified in $\mathcal{M}'$ – the latter TS violates these formulae as well. On the contrary, we cannot infer anything about the formulae that $\mathcal{M}$ satisfies. The objective of this paper is to provide methods that aim at reducing the verification effort of those formulae in $\mathcal{M}'$.

3. FEATURES COMPOSITION

Like Li *et al.* [11], we propose to model the behaviour of an SPL composed of conservative features by a set of automata. One of these automata – *viz.* a TS – models the behaviour of a rough system called the *base product*. We denote it by $\mathcal{C}$. Basically, only mandatory features are part of this product, which is thus not necessarily valid according to the constraints between the features. Then, we propose to model optional features by a set of TS.

In addition to the usual constructs, we provide any such TS $\mathcal{F}$ with *activation conditions* that determine when the feature makes the system reach that TS, and *return conditions* that define when the execution reaches anew a state of the base product. In this paper, we propose to specify both the activation condition and the return condition as a subset of states in $\mathcal{C}$. In $\mathcal{F}$, the concept of initial state is irrelevant since, as we will see, the initial states of the composition of $\mathcal{F}$ with $\mathcal{C}$ are the initial states of $\mathcal{C}$. We assume that all the automata are defined over the same set AP of atomic propositions. This results in a Transition System + (TS$^+$), defined as follows. Note that in order to avoid ambiguity, we annotate each component of a TS $\mathcal{M}$ with a subscript $\mathcal{M}$, e.g. $S_\mathcal{C}$ denotes the set of states of $\mathcal{C}$.

Definition 6 *A TS$^+$ defined over a TS $\mathcal{C}$ is a tuple (S, Act, $\rightarrow$, $AP_\mathcal{C}$, L, A, R) where S, Act, AP, L are defined as in TS, $S \cap S_\mathcal{C} = \emptyset$, $A \subseteq S_\mathcal{C}$ is the activation condition, $R \subseteq S_\mathcal{C}$ is the return condition, and $\rightarrow \subseteq (S \cup A) \times Act \times (S \cup R)$ is the transition relation.*

We call activation (resp. return) state any state $s \in A$ (resp. $s \in R$). An example of TS$^+$ is given in Figure 2. It models the feature allowing our routing protocol to send packets in unreliable channels. Activation and return states are discontinuous circle; the transitions leaving (resp. reaching) these states are represented as discontinuous lines.

Accordingly, the behavioural model for a whole product line is defined as follows.

Definition 7 *An SPL is a tuple $(\mathcal{C}, d, \gamma)$ where $\mathcal{C}$ is the base product, $d = (F, [\![d]\!] \subseteq \mathcal{P}(\mathcal{P}(F)))$ is a feature model, and γ is a function that associates each feature in F with a (possibly empty) set of TS$^+$ $\{\mathcal{F}_{f_1}, \ldots, \mathcal{F}_{f_k}\}$ over $\mathcal{C}$.*

The composition of the base product with a feature f yields a new TS that is obtained by connecting each TS$^+$ $\mathcal{F}_{f_i}$

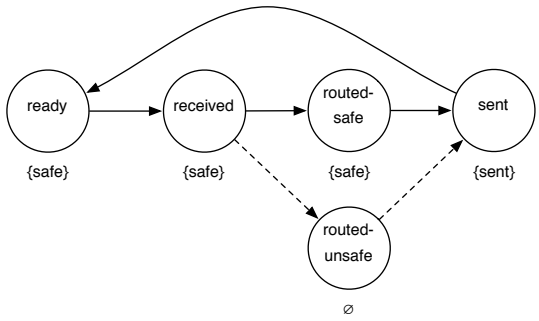

Figure 3: The composition of a TS with a TS$^+$.

associated with f on to $\mathcal{C}$. The states that are actually linked to each others are determined according to the activation conditions and the return conditions.

Definition 8 *The composition of a TS $\mathcal{C}$ with a TS$^+$ $\mathcal{F}$ over $\mathcal{C}$ is the TS $\mathcal{C} \oplus \mathcal{F} = (S_{\mathcal{C}} \cup S_{\mathcal{F}}, Act_{\mathcal{C}} \cup Act_{\mathcal{F}}, \rightarrow_{\mathcal{C}} \cup \rightarrow_{\mathcal{F}}, I_{\mathcal{C}}, AP_{\mathcal{C}}, L_{\mathcal{C}} \cup L_{\mathcal{F}})$.*

Intuitively, any transition in $\mathcal{C}$ or $\mathcal{F}$ is also a transition in $\mathcal{C} \oplus \mathcal{F}$. An execution of the composed system is able to reach the state space of the feature via the states of the base product that meet the activation condition. Similarly, the system can go back to the state space of the base product via a return state. For instance, we show in Figure 3 the result of composing the routing protocol (see Figure 1) with the above TS$^+$ (see Figure 2).

Note that these definitions are restrictive because they consider only features that do not interact. In other words, each feature modifies the behaviour of the base product only. Although it could raise additional issues, we can extend our approach to handle such cases. We briefly discuss this extension at the end of Section 5.

Next, we establish the link between features modelled as above and conservative features. Given that composition removes neither states nor transitions in the base product, any feature modelled as a set of TS$^+$ is conservative. The following theorem shows that the converse also holds, *i.e.* that the effect of any conservative feature can be expressed by a TS$^+$.

Theorem 9 *Let f be a conservative feature and $f(\mathcal{M})$ the result of integrating f into a TS $\mathcal{M}$. Then there exists a TS$^+$ $\mathcal{F}$ such that $\mathcal{M} \oplus \mathcal{F}$ is simulation-equivalent to $f(\mathcal{M})$.*

Proof Sketch. *Let $\mathcal{F}$ the TS$^+$ $(S_{f(\mathcal{M})}, Act_{f(\mathcal{M})}, \rightarrow_{f(\mathcal{M})}, AP_{f(\mathcal{M})}, L_{f(\mathcal{M})}, I_{\mathcal{M}}, \emptyset)$. Then $\mathcal{L}(\mathcal{M} \oplus \mathcal{F}) = \mathcal{L}(\mathcal{M} \uplus f(\mathcal{M}))$ where $\uplus$ denotes the disjoint union. We thus have $\mathcal{M} \oplus \mathcal{F} \approx \mathcal{L}(\mathcal{M} \uplus f(\mathcal{M}))$ and consequently $f(\mathcal{M}) \preccurlyeq \mathcal{M} \oplus \mathcal{F}$. Since f is conservative we have $\mathcal{M} \preccurlyeq f(\mathcal{M})$, which implies $\mathcal{M} \uplus f(\mathcal{M}) \preccurlyeq f(\mathcal{M}) \uplus f(\mathcal{M}) = f(\mathcal{M})$. By transitivity of the simulation relation, $\mathcal{M} \oplus \mathcal{F}$ and $f(\mathcal{M})$ are thus simulation-equivalent.* □

4. NON-PRESERVATION CONDITIONS

When a feature is added to a system that has already been checked against an LTL formula, it is in general difficult to assess whether or not the behavioural modifications performed by the feature endanger the satisfaction of that formula. However, for a conservative feature f, the composition of a system with f yields a new system with more behaviour, that is, the semantics of the new system includes that of the old one. The burden of reverification can thus be reduced by analysing only the new execution paths created as a result of the composition. In this section, we present necessary and sufficient conditions for the satisfaction of an LTL formula to be preserved upon the addition of a conservative feature into the system.

Our approach relies on the fact that the synchronous product $\otimes$ is distributive over the composition operator $\oplus$. To formalise this, we introduce the notions of Büchi Automaton + (BA$^+$) and composition between BA and BA$^+$. We define the synchronous product between a TS$^+$ $(S, Act, trans, AP, L, A, R)$ and a BA $(Q, \mathcal{P}(AP), \delta, Q_0, F)$ as the BA$^+$ $(Q' = S \times Q, Act, \delta', F' = S \times F, A' = A \times Q,, R' = R \times Q)$ where δ' is defined as is Definition 3, A' and R' are respectively called the activation sets and the return sets as in Definition 8. The composition of this automaton with a BA $(Q'', Act'', \delta'', Q_0{}'', F'')$ yields a new BA $(Q' \cup Q'', Act \cup Act'', \delta' \cup \delta'', Q_0{}'', F' \cup F'')$.

By construction, $\otimes$ is indeed distributive over $\oplus$. This implies that we can analyse *modularly* the language emptiness of the BA$^+$ resulting from the composition of the TS$^+$ of a conservative feature with the BA of the formula.

As mentioned in Section 2, an LTL formula is violated if and only if there exists a reachable cycle that contains an accepting state in the synchronous product. Let us suppose that the base product satisfies the formula. Let us assume a conservative feature modelled by a single TS$^+$ noted $\mathcal{F}$. Then the integration of the feature to the base product creates a violation of the formula if and only if one of the following four conditions is met :

1. There exists an accepting cycle in $\mathcal{F} \otimes \mathcal{B}$ that is reachable from an activation state reachable in $\mathcal{C} \otimes \mathcal{B}$.
2. There exists an accepting cycle in $\mathcal{C} \otimes \mathcal{B}$ that is reachable only via $\mathcal{F} \otimes \mathcal{B}$.
3. The feature creates an accepting cycle partly in $(\mathcal{C} \otimes \mathcal{B})$ and partly in $(\mathcal{F} \otimes \mathcal{B})$ that has an accepting state in $\mathcal{F} \otimes \mathcal{B}$.
4. The feature creates an accepting cycle partly in $(\mathcal{C} \otimes \mathcal{B})$ and partly in $(\mathcal{F} \otimes \mathcal{B})$ that has an accepting state in $\mathcal{C} \otimes \mathcal{B}$.

We formalise these conditions in the following four subsections.

4.1 Accepting cycle in $\mathcal{F} \otimes \mathcal{B}$

In this case, the state space introduced by the feature contains an accepting cycle that is reachable from the base product. Formally, this condition is given by

$$\exists a \in A_{\mathcal{F}\otimes\mathcal{B}} \cap \mathcal{R}(\mathcal{C} \otimes \mathcal{B}) \bullet \exists f \in F_{\mathcal{F}\otimes\mathcal{B}} \bullet \exists (w, v) \in Q^*_{\mathcal{F}\otimes\mathcal{B}} \times Q^+_{\mathcal{F}\otimes\mathcal{B}} \bullet f \in \delta^*(a, w) \cap \delta^*(f, v) \quad (2)$$

where $\mathcal{R}_{\mathcal{A}}$ denotes the set of reachable states in an automaton $\mathcal{A}$, that is,

$$\mathcal{R}_{\mathcal{A}} = \{q \in Q_{\mathcal{A}} \mid \exists q_0 \in (Q_0)_{\mathcal{A}} \bullet \exists w \in Q^*_{\mathcal{A}} \bullet q \in \delta^*_{\mathcal{A}}(q_0, w)\}$$

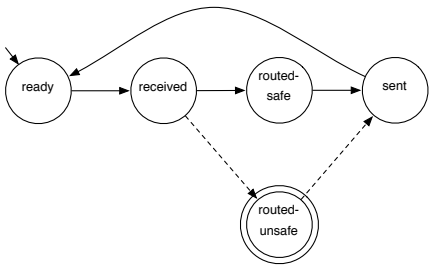

Figure 4: Third non-preservation condition.

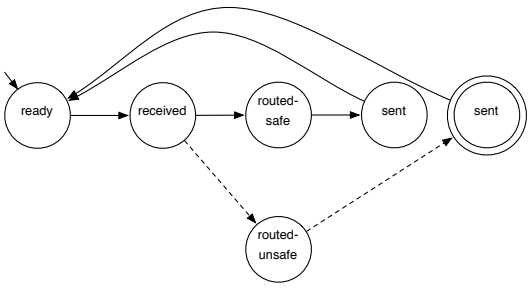

Figure 5: Fourth non-preservation condition.

Given that the accepting cycle is reachable from the state space of the base product, it turns out that the addition of the feature does not preserve the satisfaction of the property. This first condition is almost independent from the base product; the only information needed is whether or not the base product can reach an activation state of the feature leading to the accepting cycle. Thus, its analysis is modular as it does not require to perform a complete reverification of the base product.

4.2 Accepting cycle in $\mathcal{C}\otimes\mathcal{B}$ reachable only from $\mathcal{F}\otimes\mathcal{B}$

Even if no violation were reported when checking the base product, the absence of accepting cycle in the synchronous product $\mathcal{C}\otimes\mathcal{B}$ is not guaranteed. Indeed, this cycle may exist and not be reachable from any initial state $\mathcal{C}\otimes\mathcal{B}$. We thus have to verify that the addition of the feature does not create an additional path to this accepting cycle. Formally, this condition is given by

$$\exists a \in A_{\mathcal{F}\otimes\mathcal{B}} \cap \mathcal{R}(\mathcal{C}\otimes\mathcal{B}) \bullet \exists f \in F_{\mathcal{C}\otimes\mathcal{B}} \setminus \mathcal{R}(\mathcal{C}\otimes\mathcal{B}) \bullet \exists (w,v) \in Q^*_{\mathcal{F}\otimes\mathcal{B}} \times Q^+_{\mathcal{F}\otimes\mathcal{B}} \bullet f \in \delta^*(a,w) \cap \delta^*(f,v) \quad (3)$$

Again, we do not need to reverify the base product to assess this condition.

4.3 Accepting Cycle in $(\mathcal{C}\otimes\mathcal{B})\oplus(\mathcal{F}\otimes\mathcal{B})$ with an Accepting State in $\mathcal{F}\times\mathcal{B}$

In this third case, the acceptance cycle is not due to the behaviour of the feature alone, but from a combination of the feature and the base product. More precisely, there exists an accepting cycle which is partly in $\mathcal{F}\otimes\mathcal{B}$ and partly in $\mathcal{C}\otimes\mathcal{B}$. Figure 4 illustrates this situation, which can occur when checking the routing protocol against the LTL formula $\neg\Box\Diamond\neg(safe \vee sent)$ (that is, the system cannot reach infinitely often a state where neither *sent* nor $safe$ holds). The base product eventually reaches an activation state (`receive`) and enters the state space introduced by the feature. Then, the feature pursues the execution, passes through an accepting state (viz. `routed-unsafe`) and eventually reaches a return state (`send`) which is a predecessor of the activation state. Formally, this condition can be expressed as follows :

$$\exists a \in A_{\mathcal{F}\otimes\mathcal{B}} \cap \mathcal{R}(\mathcal{C}\otimes\mathcal{B}) \bullet \exists f \in F_{\mathcal{F}\otimes\mathcal{B}} \bullet \exists r \in R_{\mathcal{F}\otimes\mathcal{B}} \bullet \exists (w,v,u) \in Q^*_{\mathcal{F}\otimes\mathcal{B}} \times Q^+_{\mathcal{F}\otimes\mathcal{B}} \times Q^*_{\mathcal{C}\otimes\mathcal{B}} \bullet f \in \delta^*_{\mathcal{F}\otimes\mathcal{B}}(a,w) \wedge r \in \delta^*_{\mathcal{F}\otimes\mathcal{B}}(f,v) \wedge a \in \delta^*_{\mathcal{C}\otimes\mathcal{B}}(r,u) \quad (4)$$

Although this condition involves the base product, it does not require to recompute information that has been already computed during the verification of the base product.

4.4 Accepting Cycle in $(\mathcal{C}\otimes\mathcal{B})\oplus(\mathcal{F}\otimes\mathcal{B})$ with an Accepting State in $\mathcal{C}\times\mathcal{B}$

The last of the four conditions defines that a feature introduces a violation if it creates a cycle containing an accepting state of $\mathcal{C}\otimes\mathcal{B}$. Formally,

$$\exists a \in A_{\mathcal{F}\otimes\mathcal{B}} \cap \mathcal{R}(\mathcal{C}\otimes\mathcal{B}) \bullet \exists r \in R_{\mathcal{F}\otimes\mathcal{B}} \bullet \exists f \in F_{\mathcal{C}\otimes\mathcal{B}} \bullet \exists (w,v,u) \in Q^*_{\mathcal{F}\otimes\mathcal{B}} \times Q^*_{\mathcal{C}\otimes\mathcal{B}} \times Q^*_{\mathcal{F}\otimes\mathcal{B}} \bullet f \in \delta^*_{\mathcal{C}\otimes\mathcal{B}}(r,w) \wedge a \in \delta^*_{\mathcal{C}\otimes\mathcal{B}}(f,v) \wedge r \in \delta^*_{\mathcal{F}\otimes\mathcal{B}}(a,u) \quad (5)$$

We exemplify this case in Figure 5, where we observe that the addition of the feature has created a cycle that did not exist before. This situation may occur when checking the routing protocol against the LTL formula $\neg\Box\Diamond(\neg safe \wedge \bigcirc sent)$ (the system cannot reach an unsafe state then send a package infinitely often). From the state `routed-unsafe`, the system may reach a state `sent` which is accepting. The system may cycle around this state and thus violates the formula. Unlike the previous ones, this condition needs to update information computed during the verification of the base product, and is thus not modular.

4.5 Correctness and Completeness

It turns out that the four aforementioned conditions are necessary and sufficient to assess the non-preservation of a formula that the system satisfied prior to the addition of the feature. We first prove that any violation discovered thanks to the four conditions indeed corresponds to a violation of the formula represented by $\mathcal{B}$ in $\mathcal{C}\oplus\mathcal{F}$.

Theorem 10 *Let C be a TS, F be a TS^+ and $\mathcal{B}$ a BA such that Equations (2), (3), (4), or (5) hold for $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{B}$. Then there exists a reachable accepting cycle in $(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}$.*

Proof. *We first show that Equation (2) implies Equation (1). Given that the activation state a is reachable in $\mathcal{C}\otimes\mathcal{B}$, we have*

$$\exists i \in I_{\mathcal{C}\otimes\mathcal{B}} \bullet \exists u \in Q^*_{\mathcal{C}\otimes\mathcal{B}} \bullet a \in \delta^*_{\mathcal{C}\otimes\mathcal{B}}(i,u)$$

which, in conjunction with Equation (2), implies that

$$\exists i \in I_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}} \bullet \exists u,w \in Q^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}, w \in Q^+_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}} \bullet f \in \delta^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}(i,uw) \cap \delta^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}(f,v).$$

The above equation is equivalent to Equation (1) applied to $(\mathcal{C} \oplus \mathcal{F}) \otimes \mathcal{B}$. *The proof for the second condition is very similar and omitted here.*

Let us consider the last two conditions. Equation (4) implies that

$$\exists i \in I_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}} \bullet \exists t, w, u \in Q^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}, v \in Q^+_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}} \bullet f \in \delta^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}(i, tw) \cap \delta^*_{(\mathcal{C}\oplus\mathcal{F})\otimes\mathcal{B}}(f, vuw)$$

which is equivalent to Equation (1) as well. The proof that Equation (5) implies Equation (1) is very similar and omitted here. □

In the following theorem, we show that the four conditions are also sufficient. In other words, any violation of the formula found in $(\mathcal{C} \oplus \mathcal{F}) \otimes \mathcal{B}$ which was not found in $\mathcal{C} \otimes \mathcal{B}$ satisfies at least one of the four conditions.

Theorem 11 *Let C be a TS, F be a TS$^+$ and $\mathcal{B}$ a BA such that there exists a reachable accepting cycle in $(\mathcal{C} \oplus \mathcal{F}) \otimes \mathcal{B}$ which does not exist in $\mathcal{C} \otimes \mathcal{B}$. Then Equations (2), (3), (4), or (5) holds for $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{B}$.*

PROOF. *If there exists a reachable accepting cycle in $(\mathcal{C} \oplus \mathcal{F}) \otimes \mathcal{B}$, then it falls into one of the following categories:*

1. *The complete cycle is in $\mathcal{C} \otimes \mathcal{B}$. Then either it was discovered when checking the language emptiness of $\mathcal{C} \otimes \mathcal{B}$ or it is reachable only if the system goes through a reachable state in $\mathcal{F} \otimes \mathcal{B}$. The latter case is equivalent to Equation (3)*
2. *The complete cycle is in $\mathcal{F} \otimes \mathcal{B}$. Given that the cycle is reachable, any such case is considered in Equation (2).*
3. *The cycle is spread in both $\mathcal{C} \otimes \mathcal{B}$ and $\mathcal{F} \otimes \mathcal{B}$, and one accepting state is in $\mathcal{F} \otimes \mathcal{B}$. This condition is captured by Equation (4).*
4. *The cycle is spread in both $\mathcal{C} \otimes \mathcal{B}$ and $\mathcal{F} \otimes \mathcal{B}$, and one accepting state is in $\mathcal{C} \otimes \mathcal{B}$. This case is considered in Equation (5).*

This concludes the proof. Note that these conditions are not exclusive, e.g., it may happen that the cycle contains an accepting state in both $\mathcal{C} \otimes \mathcal{B}$ and $\mathcal{F} \otimes \mathcal{B}$. Nevertheless, such cases are considered by our conditions and thus do not harm the correctness of the proof. □

5. ALGORITHMS

Thanks to the above non-preservation conditions, we are able to determine whether or not the addition of a feature leads to the violation of a temporal property. Next, we present an incremental algorithm that aims at verifying these conditions while reusing as much as possible the verification results related to the base product. Alternative algorithms could be designed. However, the one we propose here have the following strengths :

- It is sound and complete, *i.e.* it reports a violation if and only if there actually exists one.
- It does not rely on a modified algorithms for checking the base product, that is, we keep on the traditional automata-based algorithms for LTL. It means that our approach can be implemented on top of existing model checkers.
- We systematically reuse information provided by a verifier when it verifies the base product.

Our algorithm is decomposed in three steps.

1. Checking the Base Product.

The first step consists in verifying that the base product alone satisfies the formula. This is performed using the classical LTL model checking algorithms (see [16]). If this procedure reports a violation then the addition of any conservative feature would yield a new system that violates the formula as well. If no violation is found in the base product, we record the following :

- The set of reachable states in $\mathcal{C} \otimes \mathcal{B}$.
- For each reachable accepting state f, the set of states that are reachable from f.

Given that the formula is satisfied, all these set are eventually computed during the search. Thus this first step does not increase the time complexity of the classical algorithms.

2. Checking the Feature.

Next, we verify the TS$^+$ modelling the feature against the formula. The objectives of this step are to compute :

- The set of states in $\mathcal{F} \otimes \mathcal{B}$ reachable from an activation state that is reachable in $\mathcal{C} \otimes \mathcal{B}$.
- The set of reachable states in $\mathcal{C} \otimes \mathcal{B}$ that were not reachable previously.
- For each accepting state f met during the above two computations, the set of states in $(\mathcal{C} \otimes \mathcal{B}) \oplus (\mathcal{F} \otimes \mathcal{B})$ reachable from f.

The first two computations allow us to update the set of reachable states that was computed in the previous step. Thanks to the third one, we can determine if one of the first three non-preservation conditions hold (namely if an accepting state reached during the exploration belongs to a cycle) and we can record the set of states reachable from every accepting state for other future verifications.

This search is performed through a slightly modified variant of the classical LTL model checking algorithms. Function ExtDFS provides this algorithm. Its input are a BA (*i.e.,* $B = \mathcal{C} \otimes \mathcal{B}$), a BA$^+$ (that is, $B^+ = \mathcal{F} \otimes \mathcal{B}$), the set of reachable states in B ($\mathcal{R}_B$), and a function $\mathcal{R}^F_B$ that associates each accepting state f of B with the set of states reachable from f in B. It performs a nested deep-first search to determine whether or not there exists an accepting cycle that meets one of the first three non-preservation conditions. If there does exist one, it returns $\perp$. Otherwise, it returns $\mathcal{R}_{B\oplus B^+}$ as well as a function $\mathcal{R}^F_{B\oplus B^+}$ that associates each accepting states f of $B \otimes B^+$ with a set of states reachable from f in $B \otimes B^+$. For an accepting state f that was not reached during step 1, $\mathcal{R}^F_{B\oplus B^+}(f)$ contains all the states reachable from f. However, for an accepting state f' visited during step 1, $\mathcal{R}^F_{B\oplus B^+}(f')$ contains only the states that f' can reach in B. Such a set will be updated during the next step. Apart from these return values, the function is very similar to the standard implementation for automata-based LTL model checking.

The function ExtInner, which determines the set of states in $B \otimes B^+$ reachable from an accepting state f, is however

Input: A BA B, a BA$^+$ B^+, $\mathcal{R}_B$ and $\mathcal{R}^F_B$.
Output: $\bot$ or $(\mathcal{R}_{B\oplus B^+}, \mathcal{R}^F_{B\oplus B^+})$.

```
1  R_{B⊕B+} ← R_B;
2  R^F_{B⊕B+} ← R^F_B;
3  Stack ← [];
4  foreach q0 ∈ A_{B+} ∩ R_B do
5  |  push(q0, Stack);
6  end
7  while Stack ≠ [] do
8  |  q ← top(Stack);
9  |  Post = (⋃_{α∈Act_{B⊕B+}} δ_{B⊕B}(q)) \ R_{B⊕B+};
10 |  if Post = ∅ then
11 |  |  pop(Stack);
12 |  |  if q ∈ F_B ∪ F_{B+} then
13 |  |  |  inner ← ExtInner(q, B, B+, Stack, R^F_{B⊕B+});
14 |  |  |  if inner =⊥ then
15 |  |  |  |  return ⊥;
16 |  |  |  end
17 |  |  |  R^F_{B⊕B+} ←
   |  |  |  ExtInner(q, B, B', Stack, R^F_{B⊕B+});
18 |  |  end
19 |  end
20 |  foreach q' ∈ Post do
21 |  |  R_{B⊕B+} ← R_{B⊕B+} ∪ {q'};
22 |  |  push(q', Stack);
23 |  end
24 end
```

Function ExtDFS(B, B^+, $\mathcal{R}_B$, $\mathcal{R}^F_B$)

different as it makes use of the results obtained previously. At each iteration, three cases may occur depending on the newly reached state :

(L. 7–9) The state is on the *Outer* stack. It means there exists a path from this state to f, and thus a reachable cycle including f. Consequently, the function returns $\bot$.

(L. 11–21) It is an accepting state f' such that we already computed the set of states reachable from f'. We immediately check if one such state is on the stack. If it does, then one of the first three conditions is met, *i.e.,* there exists an accepting cycle that includes states from both B and B^+. Otherwise we insert the states reachable from f' into $\mathcal{R}^F_{B\oplus B^+}(f)$. Moreover, if f' has been discovered during the first step, we pursue the DFS starting from the activation states reachable from f' (see L. 16–19).

(L. 22–24) It is none of the above. We simply push this state on the stack and continue exploring.

3. Checking the fourth condition

The last step involves the verification of the fourth non-preservation condition (*i.e.,* the existence of a reachable accepting cycle containing states of both B and B^+ when the accepting states have already been visited during the first step). If no violation is detected, it means that the system augmented with the feature satisfies the formula. In this case, this step updates the sets $\mathcal{R}^F_{B\oplus B^+}(f)$ for any accepting state f of B discovered during step 1. Function ExtUpdate performs this verification and the update; its inputs are B, B^+, $\mathcal{R}_B$ and $\mathcal{R}^F_{B\oplus B^+}$. It returns either $\bot$ or an updated function $\mathcal{R}^F_{B\oplus B^+}$. To update the set $\mathcal{R}^F_{B\oplus B^+}(f)$ for a given f, it calls Function ExtUpdateInner, which is very similar to ExtInner and uses the same optimisations. Therefore, we do not provide a detailed algorithm for the latter function.

Input: A BA B, a BA$^+$ B^+, $f \in F_B$, a stack of states *Outer*, and $\mathcal{R}^F_{B\oplus B^+}$.
Output: $\bot$ or $\mathcal{R}^F_B(f)$.

```
1  R^F_{B⊕B+}(f) ← ∅;
2  Stack ← push(f, []);
3  while Stack ≠ [] do
4  |  q ← pop(Stack);
5  |  Post = (⋃_{α∈Act_{B⊕B+}} δ_{B⊕B+}(q)) \ R^F_{B⊕B+}(f);
6  |  foreach q' ∈ Post do
7  |  |  if onStack(Outer, q') then
8  |  |  |  return ⊥;
9  |  |  end
10 |  |  R^F_{B⊕B+}(f) ← R^F_{B⊕B+}(f) ∪ {q'};
11 |  |  if q' ∈ F_B ∪ F_{B+} • R^F_{B⊕B+}(q') ≠⊥ then
12 |  |  |  if ∃q'' ∈ R^F_{B⊕B+}(q') • onStack(Outer, q'')
   |  |  |  then
13 |  |  |  |  return ⊥;
14 |  |  |  end
15 |  |  |  R^F_{B⊕B+}(f) ← R^F_{B⊕B+}(f) ∪ R^F_{B⊕B+}(q');
16 |  |  |  if q' ∈ F_B then
17 |  |  |  |  foreach q'' ∈ A_{B+} ∩ R^F_{B⊕B+}(q') do
18 |  |  |  |  |  push(q'', Stack);
19 |  |  |  |  end
20 |  |  |  end
21 |  |  end
22 |  |  else
23 |  |  |  push(q', Stack);
24 |  |  end
25 |  end
26 end
```

Function ExtInner(B, B', f, *Outer*, $\mathcal{R}^F_B$)

Discussion. If the formula is shown to be satisfied at the end of step 3, then $\mathcal{R}_{(\mathcal{C}\otimes\mathcal{B})\oplus(\mathcal{F}\otimes\mathcal{B})}$ and $\mathcal{R}^F_{(\mathcal{C}\otimes\mathcal{B})\oplus(\mathcal{F}\otimes\mathcal{B})}$ are as if step 1 had been performed directly on the BA $(\mathcal{C}\otimes\mathcal{B})\oplus(\mathcal{F}\otimes\mathcal{B})$. Therefore, we could verify if the addition of a new conservative feature still maintains the satisfaction of the formula. This implies that we can relax some of the assumptions we made earlier:

1. The conservative feature can be modelled by more than one TS$^+$. Indeed, each of these TS$^+$ can be successively added and incrementally verified through the application of step 2 and 3 upon each addition.

2. The TS$^+$ of a conservative feature can be connected onto the TS$^+$ of a previously verified conservative feature. The only condition for the correctness of the algorithm is that the features are incrementally added in a consistent manner. For instance, a feature that connects to another should not be integrated to the system before the latter. The definition of formal rules to ensure the consistency in such cases is left for future work.

Input: A BA B, a BA$^+$ B^+, $\mathcal{R}_B$ and $\mathcal{R}^F_B$.
Output: $\bot$ or updated $\mathcal{R}^F_{B\oplus B^+}$.

```
foreach f ∈ R_B do
    Outer ← push(f, []);
    inner ← ExtUpdateInner(f, B, B', Outer, R^F_{B⊕B+});
    if inner = ⊥ then
        return ⊥;
    end
    else
        R^F_{B⊕B+} ← R^F_{B⊕B+} ∪ inner;
    end
end
return R^F_{B⊕B+}
```

Function ExtUpdate(B, B^+, $\mathcal{R}_B$, $\mathcal{R}^F_{B\oplus B^+}$)

6. CONCLUSION

In this paper, we presented an automata-based approach to verify conservative feature by reusing results obtained during the verification of the base product. We showed that such features can be modelled as a TS$^+$, and defined the composition of this TS$^+$ to the TS modelling the behaviour of the system. We proposed necessary and sufficient conditions for properties (*viz.* LTL formulae) of the system to be preserved after such a feature has been added.

In our future work, we plan to extend our approach to wider classes of features. For instance, *regulative* features do not augment the behaviour to the system. By adding such feature to a system, one may make this system satisfy a formula that it violated before, that is, by suppressing a reachable accepting cycle in the synchronous product. The study of incremental approaches for arbitrarily cross-cutting features also remains an open challenge. An alternative is to combine the above approach with variability-aware behavioural formalism like Featured Transition Systems [6]. Cross-cutting features can be represented as part of this formalism, whereas special features (e.g., conservative features) are verified thanks to our method.

There are two other limitations we are aiming to overcome. In our current approach, we assumed the features to stand in a closed world. It implies that when several features are added, how those will connect to each other is pre-determined. On the contrary, it is likely that features are developed independently; this can result in unexpected interactions between them and creates behaviour that was not anticipated. Moreover, features are not always commutative; the effect of a new feature may cancel the changes of a previous one. We thus plan to study the definition of formal rules to assess what a sound ordering between features would be.

Finally, after studying all the above extensions, we will implement the complete approach in a tool and will evaluate to what extent the checking time decreases, with respect to a verification from scratch of the new system.

7. REFERENCES

[1] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *Proceedings of ASE'11*. IEEE Computer Society, 2011.

[2] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *Proceedings of SPLC'11*, pages 130–139. Springer-Verlag, 2011.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[4] F. Cassez, M. D. Ryan, and P. yves Schobbens. Proving feature non-interaction with alternating-time temporal logic. In *Language Constructs for Describing Features – proceedings of the FIREworks workshop*. Springer, 2001.

[5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of ICSE '10*, pages 335–344, New York, NY, USA, 2010. ACM.

[7] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Managing evolution in software product lines : A model-checking perspective. In *Proceedings of VaMoS'12*, pages 183–191. ACM, 2012.

[8] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *IFIP WG 6.1 FMOODS '08*, pages 113–131. Springer, 2008.

[9] D. Kozen. Results on the propositional μ-calculus. In *Proceedings of ICALP '82*, pages 348–359, London, UK, UK, 1982. Springer-Verlag.

[10] K. Lauenroth, S. Toehning, and K. Pohl. Model checking of domain artifacts in product line engineering. In *IEEE/ACM ASE*, pages 269–280, 2009.

[11] H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *SIGSOFT FSE*, pages 89–98, 2002.

[12] J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engg.*, 18:39–76, March 2011.

[13] R. Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[14] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[15] N. T. Thang and T. Katayama. Specification and verification of inter-component constraints in ctl. In *Proceedings of the 2005 conference on Specification and verification of component-based systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.

[16] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS '86*, pages 332–344. IEEE CS, 1986.

[17] J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of AOSD '09*, pages 161–172, New York, NY, USA, 2009. ACM.