Reducing the Model Checking Cost of Product Lines Using Static Analysis Techniques[⋆]

Hamideh Sabouri[1] and Ramtin Khosravi[1,2]

[1] School of Electrical and Computer Engineering University of Tehran
Karegar Ave., Tehran, Iran
[2] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

Abstract. Software product line engineering is a paradigm to develop software applications using platforms and mass customization. Component based approaches play an important role in development of product lines: Components represent features, and different component combinations lead to different products. The number of combinations is exponential in the number of features, which makes the cost of product line model checking high. In this paper, we propose two techniques to reduce the number of component combinations that have to be verified. The first technique is using the static slicing approach to eliminate the features that do not affect the property. The second technique is analyzing the property and extracting sufficient conditions of property satisfaction/violation, to identify products that satisfy or violate the property without model checking. We apply these techniques on a vending machine case study to show the applicability and effectiveness of our approach. The results show that the number of generated states and time of model checking is reduced significantly using the proposed reduction techniques.

1 Introduction

Software product line engineering is a paradigm to develop software applications using platforms and mass customization. To this end, the commonalities and differences of the applications should be modeled explicitly [1]. Feature models are widely used to model the variability of software product lines. A feature model is a tree of features, containing mandatory and optional features as well as a number of constraints among them. A product is then defined by a combination of features, and product family is the set containing all of the valid feature combinations [2]. A configuration vector can be used to keep track of inclusion or exclusion of features.

The Vending Machine Example: Feature Model. Throughout this paper, we use a product family of vending machines as a running example. A vending machine may serve coffee and/or tea. It also may add milk to the coffee. Figure 1 shows the feature model of the family of vending machines.

[⋆] This research was in part supported by a grant form IPM. (No. CS1390-4-02).

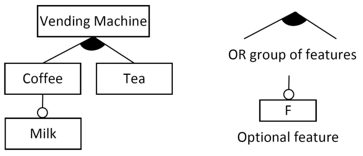

Fig. 1. The feature model of the vending machine example

Software product line engineering enables proactive reuse by developing a family of related products. One of the main approaches to develop software product lines is the compositional approach, in which features are implemented as distinct code units [3]. These code units are reused when the corresponding units are composed to generate each product. Component technology [4] is suitable in this approach as reusability is an important characteristic of software components. In component-based development of product lines, each feature is implemented using a component. Some of the features can be implemented within the components in a fine-grained manner as well, using annotative techniques [5]. Consequently, the behavior of a component may change according to inclusion or exclusion of the features. Software product line engineering is used in the development of embedded and critical systems [6]. Therefore formal modeling and verification of software product lines is essential.

Model checking [7] is a promising technique for developing more reliable systems. Recently, several approaches have been developed for formal modeling of product lines [8–13]. These approaches capture the behavior of the entire product family in a single model by including the variability information in it. In other words, it is specified in the model how the behavior changes when a feature is included or excluded. Model checking of product lines is discussed in [10, 12, 13]. In these approaches, the model checker investigates all of the possible feature combinations when verifying the model of a product family against a property, and the result of model checking is the set of products that satisfy the given property. The focus of these works is on adapting model checking algorithms to verify product families, and they do not address the state space explosion issue. However, the main problem of model checking is its high computational and memory costs which may lead to state space explosion. This problem limits the applicability of model checking technique to verify product lines, as in product families the number of products can be exponential in the number of features. In [14, 15], two incremental approaches are proposed for product line verification. In [14], only sequential composition of features is discussed which is a considerable limitation as the approach is not applicable to concurrent systems. The focus of [15] is on reducing the effort of applying deductive verification techniques (not model checking) on product lines. The main idea of our approach is to use static slicing and static analysis techniques to tackle the state space explosion problem in model checking of component-based software product lines.

We use Rebeca to model product families in a component-based manner, as a basis to explain our approach. However, the approach is not limited to Rebeca models, and it is applicable to any modeling language with slicing analysis support. In our approach, each feature is modeled using one component that captures its corresponding behavior, or using an alternative behavior within a component that changes the behavior of the component based on the presence or absence of the feature accordingly. Each product contains the components associated to the features that are included in the product, and the behavior of each of its components is determined according to the features that are included/excluded in that product. The model checker considers all of the possible combinations of components and alternative behaviors, to verify the product family. The focus of this paper is on reducing the number of combinations that should be investigated in model checking. We propose two techniques for this purpose.

The first technique uses the static slicing approach. Static slicing [16] is an analysis technique that extracts the statements from a program that are relevant to a particular computation. This technique has been used as a reduction technique in model checking of Promela [17], CSP [18], Petri-nets [19], and Rebeca [20, 21] models. In [22], an evaluation of applying static slicing for model reduction is presented. The result shows significant reductions that are orthogonal to a number of other reduction techniques, and applying slicing is always recommended because of its automation and low computational costs. One of the main approaches for slicing is using reachability analysis on program dependence graph. The nodes of a program dependence graph are the statements of the program, and its edges represent data and control dependencies among the statements. In this paper, we adapt the program dependence graph and the reachability algorithm, to use static slicing to identify the features that do not affect the correctness of the property. By discarding these features, the model checker investigates fewer feature combinations when model checking the product family.

In the second technique, we analyze the property statically to extract sufficient conditions of its satisfaction or violation. These conditions are used along with reachability conditions for variables to conclude satisfaction or violation of the given property for certain products, without verification. The model checker does not verify these products, therefore the number of feature combinations that should be verified is reduced. It should be noted that the proposed techniques (slicing, extracting conditions from property, and investigating reachability of variables) can be applied automatically.

This paper is structured as follows. Section 2 explains how product families are modeled and model checked. In Section 3 we describe the slicing technique that is used to identify the features that do not affect a property. Section 4 describes our approach for extracting sufficient conditions of property satisfaction/violation, and identifying products that satisfy or violate the property, without model checking. In Section 5 we present the results of using the two proposed techniques for reducing the feature combinations of a vending machine case study. Finally, we conclude our work in Section 6.

2 Modeling and Model Checking Product Families

This section introduces the Rebeca modeling language [23], and explains how a product family can be modeled and model checked using Rebeca. We select Rebeca as a basis to describe our approach, because it is suitable for modeling concurrent systems, it is supported by the Modere model checking tool [24], it supports components [25], and the slicing technique is adapted to be applicable on Rebeca models [20, 21]. However, our proposed approach is not limited to Rebeca models, and can be applied to other modeling languages with similar facilities as well.

2.1 Rebeca

Rebeca is an actor-based language for modeling concurrent and distributed systems as a set of reactive objects which communicate via asynchronous message passing. A Rebeca model consists of a set of *reactive classes*. Each reactive class contains a set of *state variables* and a set of *message servers*. Message servers execute atomically, and process the receiving messages. The *initial* message server is used for initialization of state variables. A Rebeca model has a *main* part, where a fixed number of objects are instantiated from the reactive classes and execute concurrently. We refer to these objects as *rebecs*. The rebecs have no shared variable, and each rebec has a single thread of execution that is triggered by reading messages from an unbounded message queue. When a message is taken from the queue, its corresponding message server is invoked. In [25], components are added to the Rebeca language to encapsulate tightly coupled reactive objects. In other words, a component is a set of one or more reactive objects.

2.2 Product Family Model

To model product families, we should model optional components (which may be included in some of the products, and excluded in other products), and alternative behaviors of components. Different combinations of optional components and alternative behaviors lead to different products. To this end, we use a special tag $@AC$ before a statement to specify the *application condition* of the statement. An application condition is a propositional logic formula in terms of features. This tag indicates that the statement will be executed only in those products that AC holds. When a feature F corresponds to a component, we use $@F$ tag before all the message server calls to that component. Subsequently, message servers of a component are invoked only if its associated feature is included in a product. If the feature is excluded in a product, no message is sent to its corresponding component, and the component will be excluded. Moreover, these tags can be used to indicate the change of the behavior within components according to presence and absence of features.

The Vending Machine Example: Rebeca Model. Figure 2 shows the Rebeca code for the product family of vending machines. In this model, there is a controller component that manages coffee and tea requests and sends messages to the coffee maker and tea maker components accordingly. The *nextRequest* message server (line 12”) is responsible for handling the requests. When there is request for coffee ($req = 1$), the *serveCoffee* message is put in the queue of *coffeeMaker*, if the machine is capable of serving coffee (line 15”). If the machine does not have the coffee option, the coffee request is ignored and the machine processes the next request (line 17”). The tea request ($req = 2$) is handled in a similar way. Consequently, if the coffee or tea feature is excluded in a product, no message is sent to the corresponding component, and the component will be also excluded. In the coffee maker component, the behavior changes according to the existence of the milk feature. If the milk feature is included in a product, milk is added to coffee (line 15). One of the linear temporal logic (LTL) [26] properties that can be considered for this model is $P : \Box(\neg(addingCoffee \wedge addingTea))$, where $\Box$ stands for globally. This property describes that the machine should not add both coffee and tea to a drink at the same time.

```
1  reactiveclass CoffeeMaker {
2    knownrebecs {
3       Controller ctrl;
4    }
5    statevars {
6       boolean addingCoffee;
7       boolean addingMilk;
8    }
9    msgsrv initial() {
10      addingCoffee = false;
11      addingMilk = false;
12   }
13   msgsrv serveCoffee() {
14      addingCoffee = true;
15      @Milk addingMilk = true;
16      self.serveComplete();
17   }
18   msgsrv serveComplete() {
19      addingCoffee = false;
20      addingMilk = false;
21      ctrl.nextRequest();
22   }
23 }
```

```
1'  reactiveclass TeaMaker {
2'    knownrebecs {
3'       Controller ctrl;
4'    }
5'    statevars {
6'       boolean addingTea;
7'    }
8'    msgsrv initial() {
9'       addingTea = false;
10'   }
11'   msgsrv serveTea() {
12'      addingTea = true;
13'      self.serveComplete();
14'   }
15'   msgsrv serveComplete() {
16'      addingTea = false;
17'      ctrl.nextRequest();
18'   }
19' }
```

```
1”  reactiveclass Controller{
2”    knownrebecs {
3”       CoffeeMaker cm;
4”       TeaMaker tm;
5”    }
6”    statevars {
7”       int req;
8”    }
9”    msgsrv initial() {
10”      self.nextRequest();
11”   }
12”   msgsrv nextRequest() {
13”      req = ?(1,2);
14”      if(req == 1)
15”        @Coffee
           cm.serveCoffee();
16”      if(req == 1)
17”        @!Coffee
           self.nextRequest();
18”      if(req == 2)
19”        @Tea
           tm.serveTea();
20”      if(req == 2)
21”        @!Tea
           self.nextRequest();
22”   }
23” }
```

Fig. 2. The Rebeca code of the product family of vending machines

2.3 Model Checking the Product Family

For a product line with n features (where each feature corresponds to a component or an alternative behavior of a component), potentially there exist 2^n products in its corresponding product family. To model check the product family, a configuration vector $C \in \langle I, E, ?\rangle^n$ (I: Included, E: Excluded, ?: not decided) is used to keep track of inclusion and exclusion decisions that are made for each feature [10]. The validity of configuration vector with respect to the feature model can be checked during model checking by transforming the feature model to a propositional logic formula [27] and using a SAT-solver (like [28]) to investigate its satisfiability. The result of model checking a product family against a property is the set of products (represented through configuration vectors) that satisfy the given property.

The Vending Machine Example: Model Checking. We assume the first, second, and third elements of configuration vector correspond to Coffee, Tea, and Milk features, respectively. The result of model checking the product family of vending machines against the property P is:

$$R = \{\langle E, I, E\rangle, \langle I, E, E\rangle, \langle I, I, E\rangle, \langle I, E, I\rangle, \langle I, I, I\rangle\}$$

Note that the configurations $\langle E, E, E\rangle$, $\langle E, E, I\rangle$, and $\langle E, I, I\rangle$ do not appear in R as they do not represent valid products, according to the feature model.

3 Slicing the Model of a Product Family

The main purpose of slicing is to extract the statements of a program that are relevant to a particular computation. A backward program slice consists of the statements that potentially affect the values computed by some statement of interest (referred to as a slicing criterion). A common approach for program slicing is applying a graph reachability algorithm on the program dependence graph. In this section, we first describe the program dependence graph of Rebeca models that capture the behavior of a product family, and then present the slicing algorithm that computes the slice of the product family model, followed by a short discussion on model checking the computed slice.

3.1 Program Dependence Graph

A program dependence graph models the data and control dependencies that exist among the statements of a program. In such a graph, the nodes represent the statements of a program, and the edges are dependencies among them. A *data dependence* edge exists between two statements if one statement assigns a value to a variable and the other statement may read the value of that variable before it is changed by another statement. A *control dependence* edge exists between two statements if one statement determines whether the other statement is executed.

A special dependence graph named *Rebeca Dependence Graph* (RDG), is introduced for Rebeca in [20]. In this graph, there is a *class* node for each reactive class, and *member dependence* edges connect the class nodes to their message servers. Each message server is modeled by an *entry* node, a set of nodes representing its statements, and data dependence and control dependence edges modeling dependencies within the body of the message server. Sending a message is represented through an *activation* node. In addition, an *activation* edge is used to connect the activation node to the entry node of the corresponding message server. Finally, *intra-rebec dependence* edge represents the dependency between a statement that writes on a state variable in a message server, and a statement which reads the value of that variable in another message server. To adapt the dependence graph for product families, we add a tag to the nodes to specify their application conditions.

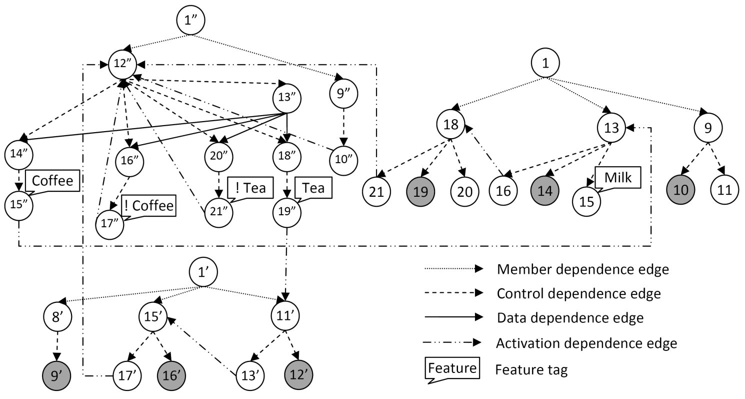

Fig. 3. The RDG of the vending machine example

The Vending Machine Example: RDG. Figure 3 shows the RDG of the vending machine. In this graph the nodes 15, 15”, 17”, 19”, and 21”, are tagged with a feature as their corresponding statements in the Rebeca model are tagged with these features.

3.2 Slicing Algorithm

After constructing the program dependence graph, the slice with respect to a property can be computed using a graph reachability algorithm. The slicing criterion consists of the statements that assign values to the variables that appear in the given property. Figure 4 shows the static slicing algorithm that is adapted to extract the features affecting the property as well. To this end, the algorithm traverses the graph backwards (starting from the slicing criterion nodes), and adds the traversed nodes to the slice, and their corresponding features to the relevant features set. In this algorithm, we assume that *Features*(v) gives the set of features that appear in the application condition of node v. The features in the set F are the components and the alternative behaviors that their presence

```
Input: The set of slicing criterions (C) and RDG (Rebeca Dependence Graph)
Output: Slice S, Relevant features set F

S={};                       /*initialize the slice*/
F={};                       /*initialize the relevant features set*/
for each($c_i$∈C){          /*for each slicing criterion*/
    W={$c_i$};              /*add the slicing criterion node to the work list*/
    S=S∪{$c_i$};            /*add the slicing criterion node to the slice*/
    while(W≠∅){             /*while the work list is not empty*/
        W=W\{w};            /*remove one element (w) from the work list*/
        for each(v⇀w){      /*for each node v on which w depends*/
            if(v∉S){        /*if the node is not included in the slice*/
                W=W∪{v};    /*add it to the work list and the slice*/
                S=S∪{v};
                F=F∪Features(v); /*add the corresponding features to the relevant feature set*/
            }
        }
    }
}
```

Fig. 4. Static slicing algorithm adapted to extract relevant features

or absence affects the correctness of the property. Therefore, the model checker should investigate their different combinations.

The Vending Machine Example: Slicing. The slicing criterion nodes for the property $P : \Box(\neg(\mathit{addingCoffee} \wedge \mathit{addingTea}))$, are indicated by gray nodes in Figure 3. The slice computed by the slicing algorithm contains all of the nodes except 11, 15, 20, and the feature set is $F = \{\mathit{Coffee}, \mathit{Tea}\}$.

3.3 Model Checking the Slice

The features that do not exist in the set F represent the components and alternative behaviors that do not affect the property. Therefore, the combinations of these features can be ignored when model checking the slice of a product family. Having a feature model with n features, there will be at most 2^n feature combinations (products), in the product family. By excluding m features that do not affect the property, the number of products to verify is reduced to $2^{(n-m)}$. The configuration vector is $C \in \langle I, E, ?\rangle^{(n-m)}$, as practically, the value of an element that its associated feature is removed always remains as "?".

The result of model checking the slice of product family against a property is the set R containing the configurations that satisfy the given property. However, these configurations are based on the combinations of $n - m$ features and do not describe identifiable products. As the other m features do not affect the property, we can combine the configurations in R with inclusion and exclusion of each of these features, taking constraints of the feature model into account, to achieve the final result. If we have r configurations such as $C \in \langle I, E\rangle^{(n-m)}$ in R, The ultimate result R' would contain $(r \times 2^m) - u$ configurations in the form $C \in \langle I, E\rangle^n$, where u is the number of feature combinations that are not valid according the feature model.

The Vending Machine Example: Model Checking the Slice. The Milk feature does not affect the property P, and does not appear in the slice. This reduces the number of products in the product family from 2^3 to 2^2. The result of model checking the slice against P is:

$$R = \{\langle E, I\rangle, \langle I, E\rangle, \langle I, I\rangle\}$$

In the next step, the milk feature should be combined with each of the above configurations. So it should be included and be excluded in these configurations (that leads to two new configurations per each configuration). The final result is R' that consists of $(3 \times 2^1) - 1$ configurations ($\langle E, I, I\rangle$ is invalid):

$$R' = \{\langle E, I, E\rangle, \langle I, E, E\rangle, \langle I, E, I\rangle, \langle I, I, E\rangle, \langle I, I, I\rangle\}$$

4 Static Analysis of Property Satisfaction/Violation in Products

In this section, we describe how satisfaction/violation of a property can be inferred for some of the products without model checking. For this purpose, we extract sufficient conditions for property satisfaction/violation in terms of initial values of atomic propositions and the possibility of their change in the model. We assume that a property is described using boolean variables where each variable corresponds to an atomic proposition. Therefore, we can evaluate sufficient conditions using the initial values of the variables and the possibility of their change in different products. The latter is achieved by analyzing the reachability of statements to obtain a condition in terms of presence and absence of features, which describes in which products the value of a variable may change. Using the result of evaluating sufficient conditions, we determine a subset of products that satisfy/violate the property without model checking. In other words, we indicate in which components and in which of their alternative behaviors the value of a variable does not change, and consequently the property is satisfied/violated.

It should be mentioned that this analysis only makes sense for models of product families that capture the behavior of all products. In traditional model checking, the value of a variable changes when the model is executed, and almost always it is not possible to infer satisfaction/violation of a property without model checking.

4.1 Condition Extraction from the Property

In this work, we consider properties expressed in linear temporal logic (LTL) [26]. An LTL formula over the set of AP of atomic propositions is formed according the following grammar:

$$\varphi ::= true \mid false \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \mathrm{U} \varphi_2$$

In the above grammar, $p \in AP$, and $\Box, \Diamond$, and U stand for globally, finally, and until operators respectively.

A transition system *TS* is a tuple $(S, Act, \rightarrow, I, AP, L)$ where S is a set of states, *Act* is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function. For simplicity, in this paper we assume a single initial state s_0 for a transition system. A state s is reachable from the initial state, $s_0 \rightarrow^* s$, if there exists a set of actions $\alpha_i \in Act$ such that $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} s$.

Figure 5 shows the proposed rules for extracting sufficient conditions of property satisfaction/violation. These conditions are statically inferable from the initial values of atomic propositions, and also the atomic propositions that do not vary in *TS*. The notation $\overline{\mathcal{V}}_{TS}(\varphi)$ means that the LTL formula φ does not vary in *TS*, because some of the atomic propositions in φ do not change in *TS*.

Rules 1-8 are trivial. We can infer $TS \models \Box\varphi$ from $TS \models \varphi$ (Rule 9) if φ does not vary in *TS* ($\overline{\mathcal{V}}_{TS}(\varphi)$). From $TS \not\models \varphi$ we can conclude that $TS \not\models \Box\varphi$, as φ should hold in all states and otherwise $\Box\varphi$ is violated (Rule 10). Similar justifications can be made for the other rules.

Using these rules, we extract sufficient conditions for property satisfaction or violation. These conditions are propositional logic formulas in terms of initial values of atomic propositions ($p \in L(s_0)$) and their variability ($\overline{\mathcal{V}}_{TS}(p)$).

The Vending Machine Example: Extracting Satisfaction/Violation Conditions. For the property $P : \Box(\neg(\mathit{addingCoffee} \wedge \mathit{addingTea}))$ we can extract sufficient conditions for satisfaction/violation by applying the rules in Figure 5 in the following order (it is assumed that p is $(\mathit{addingCoffee} = \mathit{true})$, and q is $(\mathit{addingTea} = \mathit{true})$):

$$\begin{array}{lr}
TS \models (\Box(\neg(p \wedge q))) \ \ \textbf{\textit{if}} \ \ (TS \models (\neg(p \wedge q))) \ \wedge \ \overline{\mathcal{V}}_{TS}(\neg(p \wedge q)) & Rule(9) \\
TS \models (\neg(p \wedge q)) \ \ \textbf{\textit{if}} \ \ TS \not\models (p \wedge q) & Rule(3) \\
TS \not\models (p \wedge q) \ \ \textbf{\textit{if}} \ \ (TS \not\models p) \vee (TS \not\models q) & Rule(8) \\
TS \not\models p \ \ \textbf{\textit{if}} \ \ p \notin L(s_0) & Rule(2) \\
TS \not\models q \ \ \textbf{\textit{if}} \ \ q \notin L(s_0) & Rule(2) \\
\overline{\mathcal{V}}_{TS}(\neg(p \wedge q)) \ \ \textbf{\textit{if}} \ \ \overline{\mathcal{V}}_{TS}(p \wedge q) & Rule(18) \\
\overline{\mathcal{V}}_{TS}(p \wedge q) \ \ \textbf{\textit{if}} \ \ \overline{\mathcal{V}}_{TS}(p) \wedge \overline{\mathcal{V}}_{TS}(q) & Rule(22) \\
\overline{\mathcal{V}}_{TS}(p \wedge q) \ \ \textbf{\textit{if}} \ \ (TS \not\models p) \wedge \overline{\mathcal{V}}_{TS}(p) & Rule(23) \\
\overline{\mathcal{V}}_{TS}(p \wedge q) \ \ \textbf{\textit{if}} \ \ (TS \not\models q) \wedge \overline{\mathcal{V}}_{TS}(q) & Rule(24)
\end{array}$$

This way, the three extracted sufficient conditions of property satisfaction would be:

$$\begin{array}{l}
TS \models P \ \ \textbf{\textit{if}} \ \ (p \notin L(s_0) \vee q \notin L(s_0)) \wedge (\overline{\mathcal{V}}_{TS}(p) \wedge \overline{\mathcal{V}}_{TS}(q)) \\
TS \models P \ \ \textbf{\textit{if}} \ \ (p \notin L(s_0) \vee q \notin L(s_0)) \wedge (p \notin L(s_0) \wedge \overline{\mathcal{V}}_{TS}(p)) \\
TS \models P \ \ \textbf{\textit{if}} \ \ (p \notin L(s_0) \vee q \notin L(s_0)) \wedge (q \notin L(s_0) \wedge \overline{\mathcal{V}}_{TS}(q))
\end{array}$$

$TS \models p \quad \textbf{\textit{if}} \quad p \in L(s_0)$	Rule(1)
$TS \not\models p \quad \textbf{\textit{if}} \quad p \notin L(s_0)$	Rule(2)
$TS \models \neg\varphi \quad \textbf{\textit{if}} \quad TS \not\models \varphi$	Rule(3)
$TS \not\models \neg\varphi \quad \textbf{\textit{if}} \quad TS \models \varphi$	Rule(4)
$TS \models (\varphi_1 \vee \varphi_2) \quad \textbf{\textit{if}} \quad (TS \models \varphi_1) \vee (TS \models \varphi_2)$	Rule(5)
$TS \not\models (\varphi_1 \vee \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_1) \wedge (TS \not\models \varphi_2)$	Rule(6)
$TS \models (\varphi_1 \wedge \varphi_2) \quad \textbf{\textit{if}} \quad (TS \models \varphi_1) \wedge (TS \models \varphi_2)$	Rule(7)
$TS \not\models (\varphi_1 \wedge \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_1) \vee (TS \not\models \varphi_2)$	Rule(8)
$TS \models \Box\varphi \quad \textbf{\textit{if}} \quad (TS \models \varphi) \wedge \overline{\mathcal{V}}_{TS}(\varphi)$	Rule(9)
$TS \not\models \Box\varphi \quad \textbf{\textit{if}} \quad TS \not\models \varphi$	Rule(10)
$TS \models \Diamond\varphi \quad \textbf{\textit{if}} \quad TS \models \varphi$	Rule(11)
$TS \not\models \Diamond\varphi \quad \textbf{\textit{if}} \quad (TS \not\models \varphi) \wedge \overline{\mathcal{V}}_{TS}(\varphi)$	Rule(12)
$TS \models (\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad TS \models \varphi_2$	Rule(13)
$TS \not\models (\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_1) \wedge (TS \not\models \varphi_2)$	Rule(14)
$TS \not\models (\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_2) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(15)
$\overline{\mathcal{V}}_{TS}(p) \quad \textbf{\textit{if}} \quad \nexists s \mid (s_0 \to^* s) \wedge \; [(p \in L(s_0)) \wedge (p \notin L(s))]$	Rule(16)
$\overline{\mathcal{V}}_{TS}(p) \quad \textbf{\textit{if}} \quad \nexists s \mid (s_0 \to^* s) \wedge \; [(p \notin L(s_0)) \wedge (p \in L(s))]$	Rule(17)
$\overline{\mathcal{V}}_{TS}(\neg\varphi) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi)$	Rule(18)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \vee \varphi_2) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi_1) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(19)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \vee \varphi_2) \quad \textbf{\textit{if}} \quad (TS \models \varphi_1) \wedge \overline{\mathcal{V}}_{TS}(\varphi_1)$	Rule(20)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \vee \varphi_2) \quad \textbf{\textit{if}} \quad (TS \models \varphi_2) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(21)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \wedge \varphi_2) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi_1) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(22)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \wedge \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_1) \wedge \overline{\mathcal{V}}_{TS}(\varphi_1)$	Rule(23)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \wedge \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_2) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(24)
$\overline{\mathcal{V}}_{TS}(\Box\varphi) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi)$	Rule(25)
$\overline{\mathcal{V}}_{TS}(\Diamond\varphi) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi)$	Rule(26)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad \overline{\mathcal{V}}_{TS}(\varphi_1) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(27)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad (TS \models \varphi_2) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(28)
$\overline{\mathcal{V}}_{TS}(\varphi_1 \mathrm{U} \varphi_2) \quad \textbf{\textit{if}} \quad (TS \not\models \varphi_2) \wedge \overline{\mathcal{V}}_{TS}(\varphi_2)$	Rule(29)

Fig. 5. Rules for extracting sufficient conditions of property satisfaction/violation, based on initial values of atomic propositions, and the atomic propositions that do not vary in *TS*

A sufficient condition of property violation for P can be extracted in a similar way:

$$TS \nvDash P \ \ \textbf{\textit{if}} \ \ p \in L(s_0) \wedge q \in L(s_0)$$

4.2 Evaluation of the Extracted Conditions

The initial values of atomic propositions ($p \in L(s_0)$ or $p \notin L(s_0)$) are computed based on initialization statements. For simplicity, we assume that the property is described using boolean variables only. It should be mentioned that we can always rewrite a property such as $\Box(x = y + z)$ in the form $\Box(v = \mathit{true})$, where v is boolean variable representing $x = y + z$. This assumption implies that each atomic proposition is a boolean variable in the Rebeca model, and the value that is assigned to the variable in the initialize message server, determines if $p \in L(s_0)$ or $p \notin L(s_0)$.

The next step is to investigate if the value of the atomic proposition p may vary ($\mathcal{V}_{TS}(p)$). The value of variable v (where v corresponds to p) changes in a product if the product has a reachable statement s that assigns a value to v. According to our model for product families, a tagged statement is executed when its application condition holds in a product. Other statements are executed normally. We assume that $\mathcal{F}(s)$ gives the application condition that is associated to a tagged statement s, and for other ones returns *true*. A statement s is reachable in a product if its associated application condition holds in the product, as well as at least one of the application conditions assigned to those statements on which s is control/activation dependent (possibly indirectly). We compute the reachability condition of the statement s recursively as:

$$RC(s) = \bigvee_{r \rightharpoonup_{c,a} s} (\mathcal{F}(s) \wedge RC(r))$$

In the above computation, $r \rightharpoonup_{c,a} s$ is the set of statements on which s is control or activation dependent. To avoid recursion, we mark each statement r when its condition is extracted, and in $r \rightharpoonup_{c,a} s$ we only consider the unmarked statements. Note that when a behavioral model is inconsistent (e.g. $RC(s)$ contains the conjunction of a feature and its negation), the statement s is not reachable in any of the products.

We assume $\mathit{Def}(v)$ is the set of statements that assign value to the variable v, except the initialization statement which is the one assigning value to v in the initial message server of the Rebeca model. The value of v may change in a product, if at least one of the statements $s \in \mathit{Def}(v)$ are reachable in that product. The atomic proposition p which corresponds to v may vary in transition system TS if:

$$\mathcal{V}_{TS}(p) = \bigvee_{s \in \mathit{Def}(v)} RC(s)$$

Consequently:

$$\overline{\mathcal{V}}_{TS}(p) = \neg(\bigvee_{s \in Def(v)} RC(s))$$

The possibility of variation for p is thus described using application conditions, where each application condition is a propositional logic formula in terms of features itself. Substituting the initial values of atomic propositions and their possibility of variation ($\overline{\mathcal{V}}_{TS}(p)$) in sufficient conditions of property satisfaction/violation, leads to a number of propositional logic formulas. These formulas describe products that we can conclude satisfaction/violation of the given property in them statically. A product satisfies or violates a property if at least one of the sufficient conditions of property satisfaction or violation holds for it, because of the components and alternative behaviors that it includes. The model checker only verifies the products that their satisfaction or violation cannot be concluded from sufficient conditions.

The Vending Machine Example: Evaluation of the Extracted Conditions. We assume that atomic propositions p and q correspond to *addingCoffee* and *addingTea* variables, respectively. According to the initializations in the Rebeca model, we can conclude that $p \notin L(s_0)$ and $q \notin L(s_0)$. The statements 10 and 14 assign value to *addingCoffee* which means that $Def(addingCoffee) = \{s_{14}, s_{19}\}$. Therefore:

$$\overline{\mathcal{V}}_{TS}(p) = \neg(RC(s_{14}) \vee RC(s_{19})) = \neg\,Coffee$$

Because:

$$\begin{aligned}
&RC(s_{14}) = RC(s_{13}) = RC(s_{15''}) = Coffee \wedge RC(s_{14''}) = Coffee \wedge RC(s_{12''}) = \\
&Coffee \wedge [RC(s_{17'}) \vee RC(s_{17''}) \vee RC(s_{21''}) \vee \underbrace{RC(s_{10''})} \vee RC(s_{21})] = \\
&Coffee \wedge [RC(s_{17'}) \vee RC(s_{17''}) \vee RC(s_{21''}) \vee \underbrace{RC(s_{9''})} \vee RC(s_{21})] = \\
&Coffee \wedge [RC(s_{17'}) \vee RC(s_{17''}) \vee RC(s_{21''}) \vee \quad true \quad \vee RC(s_{21})] = Coffee
\end{aligned}$$

and:

$$RC(s_{19}) = RC(s_{18}) = RC(s_{16}) = RC(s_{13}) = Coffee$$

Similarly, we can compute $\overline{\mathcal{V}}_{TS}(q) = \neg\,Tea$. By substitution of $\overline{\mathcal{V}}_{TS}(p)$ and $\overline{\mathcal{V}}_{TS}(q)$ with $\neg\,Coffee$ and $\neg\,Tea$ respectively, the following conditions are achieved which describe the products for which satisfaction/violation of P is inferable without model checking:

$$\begin{aligned}
&TS \vDash P \quad \textbf{\textit{if}} \quad (\neg\,Coffee \wedge \neg\,Tea) \\
&TS \vDash P \quad \textbf{\textit{if}} \quad \neg\,Coffee \\
&TS \vDash P \quad \textbf{\textit{if}} \quad \neg\,Tea
\end{aligned}$$

According to the above conditions, the products that do not have the Coffee feature, and the products that do not have the Tea feature, satisfy P, and there is no need to verify them. This way, the number of the products that should be model check is reduced to $2^2 - 3$, as we can tell that the products $\langle I, E\rangle$, $\langle E, I\rangle$, and $\langle E, E\rangle$ satisfy P (although $\langle E, E\rangle$ is not a valid product).

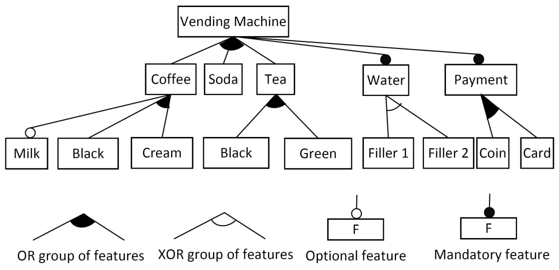

Fig. 6. The feature model of the vending machine case study

5 Results

We applied our proposed approach to a vending machine case study that is much more complex than the running example [3]. The machine includes a controller that handles the requests. Figure 6 shows the feature model of the vending machine. The coffee maker, tea maker, and soda server components are responsible for serving the associated drinks. There is also a milk adder component which adds milk to coffee. There are two coffee container components and two tea container components, containing black coffee, coffee with cream, black tea, and green tea, respectively. The coffee maker and the tea maker components use the proper container to serve the requested drink. They add water through the water component. The water component can be filled using two different mechanisms which are handled by the filler 1 and filler 2 components. Finally, there are two different payment methods for a vending machine: paying by coin, or paying by card. We defined the following six LTL properties to be verified.

– $P_1 = \Box[\neg(\mathit{ServingCoffee} \wedge \mathit{ServingTea} \wedge \mathit{ServingSoda})]$
– $P_2 = \Box(\neg \mathit{empty})$
– $P_3 = \Box(\neg \mathit{overFlow})$
– $P_4 = \Box[\neg(\mathit{addingBlackCoffee} \wedge \mathit{addingCreamCoffee})]$
– $P_5 = \Box[\neg(\mathit{addingBlackTea} \wedge \mathit{addingGreenTea})]$
– $P_6 = \Box\Diamond(\mathit{ServingSoda})$

The first property describes that the vending machine should not be serving three drinks at the same time. The second and third properties check that the

[3] The source code is available at http://ece.ut.ac.ir/rkhosravi/sourcecode

water container should not get empty, or overflow. The forth property describes that the machine should not add black coffee together with coffee and cream to a drink. This fact should be also checked for the tea drink (the fifth property). The last property states that the machine should serve soda infinitely often.

Table 1. Number of states and time of verification (in seconds) before applying the techniques (first column), after applying the slicing technique (second column), and after identifying products that satisfy/violate the property without model checking (third column), for the vending machine case study

	Complete Model		Static Slicing		Slicing and Static Analysis	
	states	time(sec)	states	time(sec)	states	time(sec)
P_1	-	-	49,307,358	24,574	25,590,940	13,849
P_2	-	-	39,169,329	17,156	39,126,321	17,138
P_3	-	-	39,182,632	18,019	19,571,384	9,119
P_4	-	-	43,484,712	19,623	16,037,384	7,517
P_5	-	-	47,317,992	24,084	14,696,264	6,951
P_6	-	-	114,547,805	142,081	63,357,123	75,356

Table 1 shows the number of states and the time of verification (in seconds) for model checking the product family of vending machine case study. The time of applying slicing technique and computing sufficient conditions are negligible comparing to model checking time and are ignored. The complete model can not be model checked against the properties because of state space explosion (first column). After applying the slicing technique and eliminating irrelevant features, the sliced model can be checked against the properties (second column). However, the number of states and time of verification can be reduced even more by extracting sufficient conditions of property satisfaction/violation, and identifying products that satisfy/violate the property without model checking.

6 Conclusion

In this paper we presented two techniques to reduce the number of products of a product line that are model checked against a property. This way, the number of generated states and the required time for verifying product families are reduced. The first technique was to apply static slicing to eliminate the features that do not affect the property. The second technique was to analyze the property and reachability of its variables in different products statically to identify products that satisfy/violate the property without model checking. The results of using these techniques in model checking the vending machine case study show the effectiveness of our approach as the number of generated states and time of verification reduced significantly after applying these techniques. The slicing and static analysis technique are completely automatic, and their cost is negligible comparing to the verification cost which makes using our approach for model checking product families practical.

References

1. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
3. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the 30th international conference on Software engineering. ICSE '08, ACM (2008) 311–320
4. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
5. Kästner, C., Apel, S.: Integrating compositional and annotative approaches for product line engineering. In: Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE), University of Passau (October 2008)
6. Ebert, C., Jones, C.: Embedded software: Facts, figures, and future. Computer **42** (April 2009) 42–52
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
8. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: Proceedings of the 16th European Symposium on Programming. ESOP'07, Springer-Verlag (2007) 64–79
9. Larsen, K.G., Nyman, U., Wasowski, A.: Modeling software product lines using color-blind transition systems. Int. J. Softw. Tools Technol. Transf. **9**(5) (2007) 471–487
10. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Proceedings of the 10th international conference on Formal Methods for Open Object-Based Distributed Systems. FMOODS '08, Springer-Verlag (2008) 113–131
11. Muschevici, R., Clarke, D., Proenca, J.: Feature Petri nets. In: Second Proceedings of the 14th international conference on Software product lines. (2010) 99–106
12. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ICSE '10, ACM (2010) 335–344
13. Sabouri, H., Khosravi, R.: An effective approach for verifying product lines in presence of variability models. In: Second Proceedings of the 14th international conference on Software product lines. (2010) 113–120
14. Liu, J., Basu, S., Lutz, R.R.: Compositional model checking of software product lines using variation point obligations. Automated Software Engg. **18** (March 2011) 39–76
15. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Proceedings of the 2010 international conference on Formal verification of object-oriented software. FoVeOOS'10, Springer-Verlag (2011) 61–75
16. Weiser, M.: Program slicing. In: Proceedings of the 5th international conference on Software engineering. (1981) 439–449
17. Millett, L., Teitelbaum, T.: Issues in slicing Promela and its applications to model checking, protocol understanding, and simulation. Software Tools for Technology Transfer (2000) 343–349

18. Bruckner, I., Wehrheim, H.: Slicing an integrated formal method for verification. In: Proceedings of Seventh International Conference on Formal Engineering Methods. ICFEM'05 (2005) 360–374
19. Rakow, A.: Slicing Petri nets with an application to workflow verification. In: Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science. SOFSEM 2008 (2008) 436–477
20. Sabouri, H., Sirjani, M.: Actor-based slicing techniques for efficient reduction of Rebeca models. Sci. Comput. Program. **75**(10) (October 2010) 811–827
21. Sabouri, H., Sirjani, M.: Slicing-based reductions for Rebeca. In: Electron. Notes Theor. Comput. Sci. Volume 260. (January 2010) 209–224
22. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Wallentine, T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS06, Springer (2006) 73–89
23. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae **63**(4) (December 2004) 385–410
24. Jaghoori, M., Movaghar, A., Sirjani, M.: Modere: The model-checking engine of Rebeca. ACM Symposium on Applied Computing - Software Verification Track (2006) 1810–1815
25. Sirjani, M., de Boer, F., Movaghar, A.: Modular verification of a component-based actor language. Journal of Universal Computer Science **11**(10) (2005) 1695–1717
26. Emerson, E.A.: Temporal and modal logic. Handbook of theoretical computer science (1990) 995–1072
27. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Proceedings of the 9th international conference on Software product lines. SPLC'05 (2005) 7–20
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. DAC '01, ACM (2001) 530–535