Efficient Verification of Evolving Software Product Lines

Hamideh Sabouri[1] and
Ramtin Khosravi[1]

[1]School of Electrical and Computer Engineering
University of Tehran, Karegar Ave., Tehran, Iran
sabouri@ece.ut.ac.ir
rkhosravi@ece.ut.ac.ir

Abstract. Software product line engineering represents a promising approach to achieve systematic reuse in development of families of software. Software product lines are intended to be used in a long period of time. As a result, they evolve over time, due to the changes in the requirements. Having several individual products in a software family, verification of the entire software family may take a considerable effort. In this paper, we present an idea for efficient verification of evolving software product lines, by reusing the result of verification and state space of the product family. To this end, we generate the state space of the product family once and verify the desired properties. The result of verification is the set of products satisfying the given properties. When the product line evolves, we may use the result of verification, and the state space to produce new results, and update the existing state space. We show the applicability of our approach by applying it on a small case study.

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 are modeled explicitly [1]. Feature models are usually used to specify the variability of software product lines. A product is defined by a combination of features. The set containing all of the valid feature combinations defines the set of products [2]. A feature model is a tree of features, containing mandatory, and optional features. It may also contain a number of constraints among the features. Feature models may evolve over time due to the changes in the requirements. This evolution may imply addition, removal, or replacement of some features. It also may add or remove some constraints.

Recently, several approaches have been developed for formal modeling and verification of product lines [3–7]. However, these works do not consider the evolution of the product line.

Software product lines are intended to be used in a long period of time. As a result, they evolve over time, due to the changes in the requirements [8]. The evolution may imply addition, removal, or replacement of some features.

Moreover, it may add or remove some constraints. At the model level, adding and removing features result in adding a new behavior and eliminating an existing behavior from the model, and the model should be verified entirely again, after these changes. However, from the product line perspective, adding features to a product line, and removing features from a product line will cause adding a number of new products, and elimination of a number of products. Moreover, removing some of the constraints, and adding some new constraints, will cause a number of previously invalid products to be valid, and invalidation of a number of the valid products. Now, if we look at the problem again, we can see that re-verifying the entire product family after adding or removing features, leads to regeneration of the state spaces of products that are generated before, and re-verification of a number of products that are verified before.

As verification of the product family may take a considerable effort, in this work, we present an idea to verify evolving product lines efficiently, by reusing the state space of the product family and the result of verification. For this purpose, we generate the state space of the product family and verify it against the properties once. The result of verification of the product family against a given property is the set of products satisfying the property. When the product line evolves, it is not necessary to generate the state space of the product family from scratch and verify the entire product family against the set of properties again. The space and the result of verification are used to update the existing state space and produce the new results.

In this paper, we use a product family of coffee machines, as a running example. A coffee machine may serve one type of coffee, and may add fix or variable amount of sugar to the coffee.

This paper is structured as follows. Section 2 introduces product line modeling using PL-CCS. In Section 3, we explain our approach to reuse the existing state space and results of verification, to verify the evolved product line. In Section 4, we present the result of applying the proposed approach on the coffee machine case study, and Section 5 concludes the work.

2 Product Line Modeling

A product line can be modeled as a system including a number of *variation points*. Each variation point has a number of *variants* that are associated to it. Each variant represents a feature, and selection of a variant of a variation point corresponds to including a feature from the feature model, in the final product. Individual products can be obtained from a product line model by deciding about all of the variation points.

The Coffee Machine Example: Feature Model The feature model of the coffee machine example is shown in Figure 1. A coffee machine may serve coffee or coffee with milk. Moreover it may add fix amount or adjustable amount of sugar to the coffee. In our paper, we use PL-CCS notation [6] to model a product

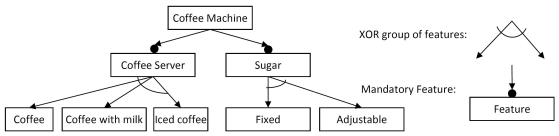

Fig. 1. The feature model of the coffee machine example

line. PL-CCS supports modeling variation in the behavior of product families, by means of *optional* and *variants* operators.

2.1 PL-CCS

The process algebra PL-CCS [6], is an extension of CCS [9], and is used to model the behavior of product families. PL-CCS extends CCS with the variants operator $\oplus$, and the optional operator $\langle\, \rangle$, to define variation points. The variants operator $\oplus$ defines a variation point that one and only one of its variants should be included in the final products. The optional operator $\langle\rangle$ defines a variation point where a variant may be included in the final products, but its inclusion is not mandatory. As the optional operator can be defined using the variants operator ($\langle P\rangle := P \oplus Nil$), we focus on the variants operator only.

In [6], the *configured-transition system* semantics is defined for a PL-CCS program. For a PL-CCS program with n variants, we associate to each variant a unique number, and keep track of inclusion or exclusion made for each variant using a *configuration vector* $\nu \in \{I, E, ?\}^n$, where I and E represent the inclusion and exclusion of the variants respectively, and ? denotes that it is not decided about the inclusion or exclusion of the variant yet. For simplicity, we assume that the i^{th} element of the configuration vector represents the decision that is made for the i^{th} variant.

The Coffee Machine Example: PL-CCS Model The product family of coffee machines can be modeled using PL-CCS as:

$$S \stackrel{def}{=} coin.AddCoffee.AddSugar.S \qquad AddCoffee \stackrel{def}{=} coffee \oplus coffee.milk$$
$$AddSugar \stackrel{def}{=} sugar \oplus (sugar + sugar.sugar)$$

According to the above model, a coffee machine receives a coin, fills the cup with coffee (corresponding to the coffee feature) or coffee and milk (corresponding to the coffee with milk feature), and finally adds sugar. It may add one unit of sugar (corresponding to the fixed sugar feature), or select between one unit or two units of sugar (corresponding to the adjustable sugar feature).

3 Verification of Evolved Product Line

In this section, we describe our approach to reuse the state space and verification result of the product family, in verification of the evolved product line.

The Coffee Machine Example: Evolution We assume that the coffee machine product line evolves by adding the new feature, *iced coffee*, to its feature model. The new model of the product family is:

$S \stackrel{def}{=} coin.AddCoffee.AddSugar.S \quad AddCoffee \stackrel{def}{=} coffee \oplus coffee.milk \oplus coffee.ice$
$AddSugar \stackrel{def}{=} sugar \oplus (sugar + sugar.sugar)$

3.1 Reusing the State Space

In our proposed approach for reuse, the state space of a product family is generated once and is updated as the product line evolves. A trivial way to reuse the state space is generating the state space of each product independently, and update the state space by adding or removing the state space of the new and eliminated products, respectively:

The state space:	$S = S_{p_1} \cup ... \cup S_{p_m}$
Adding a new product p_{m+1}:	$S' = S \cup S_{p_{m+1}}$
Removing a product p_k:	$S' = S_{p_1} \cup ... \cup S_{p_{k-1}} \cup S_{p_{k+1}} \cup ... \cup S_{p_m}$

In this way, we reuse the state space of the products that are verified before. However, we want to reuse state space that is common among products as well. For this purpose, we verify the whole product family, and store the state space. After the evolution of product line, we update the state space based on the evolution, as it is described in the following.

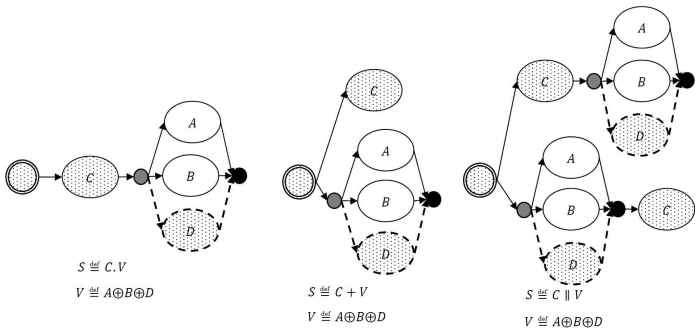

Fig. 2. Updating the existing state space after adding a new variant D, for sequential, non-deterministic, and interleaved execution of two processes

Adding a New Feature To update the state space after adding a new feature, we should mark the states where we are deciding about each variation point in the model. For simplicity, we add two special states to indicate where the behavior varies. In Figure 2, these states are shown using gray and black colors.

A new feature, appears as a new operand of a variant operator. To update the state space, we add the states of the variant to the variant behaviors which are indicated by the special states, for the associated variation point. Figure 2, shows the updated state space after adding a new variant D to the variation point, for sequential, non-deterministic, and interleaved execution of a common behavior C and a variable behavior V. In this figure, only the dashed states are added to the state space and other states are reused.

Removing a Feature To update the state space after eliminating an existing feature, we should keep track of the features that use each state. To update the state space, the states that are used only by the specific feature which is eliminated from the product line, are removed along with their outgoing and incoming transitions.

The Coffee Machine Example: Reusing the State Space Figure 3, shows how the state space of the coffee machine example can be updated by adding the dashed states, after adding a new feature named *iced coffee* to the feature model of the product line.

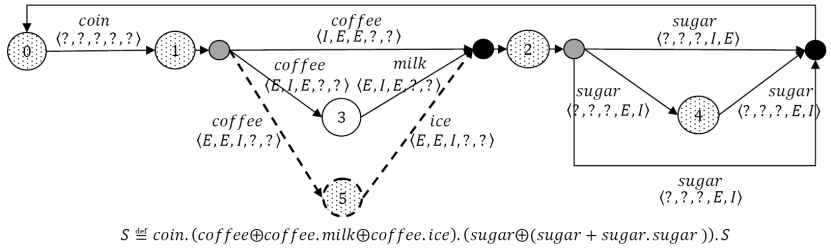

Fig. 3. Updating the existing state space of the coffee machine example after adding the new feature *iced coffee*

3.2 Reusing the Result of Verification

The result of verification of a product family is reused when new products are added to the product line, or some of the products are eliminated. The result of verification of a product family against a property φ, is the set of products that satisfy φ, and are valid according to the feature model:

$$R = \{p_i \mid (p_i \in \mathcal{V}) \wedge (p_i \vDash \varphi)\}$$

Adding a New Feature After adding a new feature to the product line, we intend to only verify the set of new products that are added to the product family consequently. For this purpose, we initialize the configuration vector, to restrict

the model checker to verify only the new products. Adding a new feature leads to adding a new variant in the configuration vector: $\nu \in \{I, E, ?\}^{n+1}$. Considering the fact that we verified all of the products that do not include the new feature before, we should only verify the new products which are the combinations of existing features with the new feature. To this end, the configuration vector is initialized as:

$$\nu = \langle \overbrace{?, ..., ?}^{n}, I \rangle$$

In this way, we only verify the new products against φ. Figure 2 shows that after adding the new variant D to the variation point, only the dotted states, which represent $C.D$, $C + D$, and $C \parallel D$ products, should be investigated for verification.

Eliminating a Feature Elimination of a feature from the product line, causes the elimination of a number of products from the product family. In this case, we only need to update the result set, and there is even no need to re-verify the product family. Therefore, for each product p_k, that is eliminated from the product family, we remove it from the result set.

Adding a New Constraint Adding a new constraint to the feature model leads to invalidation of some of the products that were valid before, and a new set of valid products $\mathcal{V}' \subseteq \mathcal{V}$. To obtain the new result set without re-verifying the product family, we check the validation of the products of the result set, and remove the products that are not valid according to the feature model anymore.

Removing a Constraint Removing an existing constraint from the feature model makes some of the invalid products valid, and leads to a new set of valid products $\mathcal{V}'$ which $\mathcal{V} \subseteq \mathcal{V}'$. In this case, we avoid re-verification of the entire product family by initializing the configuration vector, to restrict the model checker to only verify the new products. To this end, we should consider the effect of each type of constraints on the validation of products, and define the binding constraints based on it. The *requires* constraint from feature f to feature f' ($f \rightarrow f'$) leads to invalidation of products that include f, but exclude f'. The *excludes* constraint between two features f and f' ($f \rightarrow \neg f'$,$f' \rightarrow \neg f$), causes invalidation of products that include both features. We assume that f and f' correspond to the i^{th} and j^{th} ($i < j$) variants of the configuration vector ν, and and initialize the configuration vectors ν_{req} and ν_{excl} to verify only the new products, after removing the *requires* and *excludes* constraints respectively:

$$\nu_{req} = \langle \overbrace{?, ..., ?}^{i-1}, I, \overbrace{?, ..., ?}^{j-1}, E, ?, ..., ? \rangle \qquad \nu_{excl} = \langle \overbrace{?, ..., ?}^{i-1}, I, \overbrace{?, ..., ?}^{j-1}, I, ?, ..., ? \rangle$$

By applying the above initializations, we only verify the new products against φ.

The Coffee Machine Example: Result Reuse After adding the new feature, *iced coffee*, we only should verify the coffee machines that serve iced coffee and add a fixed or adjustable amount of sugar. For this purpose, we initialize the configuration vector to $\langle E, E, I, ?, ? \rangle$. The dotted states in Figure 3 show the states that should be investigated for verification. Therefore, practically we should only verify $\langle E, E, I, I, E \rangle$ and $\langle E, E, I, E, I \rangle$ products, and add them to the previous result set that we have, if they satisfy the property. If we add a new constraint to the feature model stating that "coffee with milk" feature requires the "adjustable sugar" feature, we simply remove the $\langle E, I, E, I, E \rangle$ product from the result set, as it is not valid according to the feature model anymore.

4 Results

We considered the product family of coffee machines as a case study to evaluate the proposed approach. The coffee machine presented in this paper as a running example, is a simplified version of this case study. We modeled the coffee machine using *Promela* and verified it with *Spin* model checker, as PL-CCS does not have tool support. We modeled the configuration vector, using global variables to be able to investigate our approach.

The first model of the coffee machine (M1) serves coffee which is a mandatory feature, and may serve tea, which is an optional feature, and there is one payment method, and a coffee and tea container. In the second model (M2) one more payment method is added as a new feature to the product family. In the next step, the product line of coffee machine evolves, by adding the capability of serving iced coffee (M3) and iced tea (M4). Finally, on the last two models (M5, M6), we can decide about using a small coffee/tea container or a large one, in a coffee machine. Table 1 shows the number of states generated for verification of a product family, the number of states generated for verification of single products, the number of states added to the existing state space after evolution, and the number of states investigated for verification of product family by reusing the existing results.

Table 1. The result of applying our method to the coffee machine case study

Model	Product Family	Single Products	New states	Investigated states
M1	774	854	774	774
M2	14,342	15,533	13,488	13,342
M3	26,179	44,548	10,646	26,179
M4	38,016	79,217	6,533	24,976
M5	96,682	191,493	17,465	52,126
M6	245,006	447,203	53,513	132,460

The result shows the effectiveness of our approach. For example, in the case of model M6, 447,203 states are generated to verify each product independently.

By reusing the common states among products the number of states is reduced to 245,006. However, if we reuse the state space and the result of M5, we only need to add 53,513 new states to it, and investigate 132,460 states to verify the new products.

5 Conclusion

In this paper, we proposed an idea to reuse the state space and the result of verification of a product family, to verify the product line after evolution. Using this approach, we do not need to generate the state space of the product family from scratch, and verify the entire product family against the properties again. The result of applying our proposed approach on a coffee machine example shows the effectiveness of this approach.

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. Kishi, T., Noda, N., Katayama, T.: Design verification for product line development. In: SPLC. (2005) 150–161
4. 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
5. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: ESOP. (2007) 64–79
6. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: FMOODS '08: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems, Berlin, Heidelberg, Springer-Verlag (2008) 113–131
7. Sabouri, H., Khosravi, R.: An effective approach for verifying product lines in presence of variability models. In: FMSPLE: First International Workshop on Formal Methods in Software Product Line Engineering. (2010) 113–120
8. Svahnberg, M., Bosch, J.: Evolution in software product lines: Two cases. Journal of Software Maintenance **11** (November 1999) 391–22
9. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)