An Effective Approach for Verifying Product Lines in Presence of Variability Models

Hamideh Sabouri
School of Electrical and
Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
Email: sabouri@ece.ut.ac.ir

Ramtin Khosravi
School of Electrical and
Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
Email: rkhosravi@ece.ut.ac.ir

***Abstract*—Software product lines are built around the central notion of variability. A variability model specifies possible variations among the members of the product family, constraining combinations of variability options to conform to the real-world requirements. To formally model and analyze a product line, we need to study its behavior in presence of these variability constraints. To this end, we define a method to specify variability models based on propositional logic. In addition, the modeler may specify the binding decisions that are already made for the variability options. These decisions are also described using propositional logic. The resulted propositional formulas can be used by the existing verification methods to determine which product configurations, satisfy a given property. These configurations preserve the binding decisions as well. Furthermore, configurations that are valid according to the variability model constraints, but cannot be derived from the product line behavioral model, are presented to the modeler as the missing configurations. We also distribute the state space of product lines by means of specifying some binding decisions, to avoid the state space explosion. Finally, the result of verification can be used to revise the behavioral model, the variability model, and the binding decisions.**

I. Introduction

Software product line engineering is a paradigm to develop software applications using platforms and mass customization. This employs the concept of managed variability: The commonalities and differences in the applications of the product line should be modeled explicitly. To this end, an *orthogonal variability model* is considered as a central asset in a software product line, to define the variability and relate it to other software development models such as feature models, use case models, design models, component models, and test models [1].

A variability model defines what can vary, and how does it vary, by means of *variation points* and *variants*, respectively [1]. It also consists of a set of constraints on variants, and dependencies among them. Single products can be derived from a product line, by deciding about the inclusion or exclusion of each variant. The moment of variability resolution is called the *binding time* of the variability. The decisions made to derive a product from a product line form a *configuration*.

The product line engineering paradigm, focuses on developing a product family, instead of developing several single products. Accordingly, two extreme approaches may be considered for verifying product lines: The first approach is verifying each product independently. Therefore, the binding decisions for all of the variation points should be made, before the verification. This approach is not practical due to the large number of products that can be derived from a product family. The second approach is verifying the entire product family. In this approach, all of the binding decisions are made during the model checking process: The model checker examines all of the possible decisions that can be made for the variabilities that it encounters. The result of verification of a product family against a property is the set of products (represented by configurations) that satisfy the property. This approach may lead to state space explosion for large product families. The second approach for product line verification is introduced in [2].

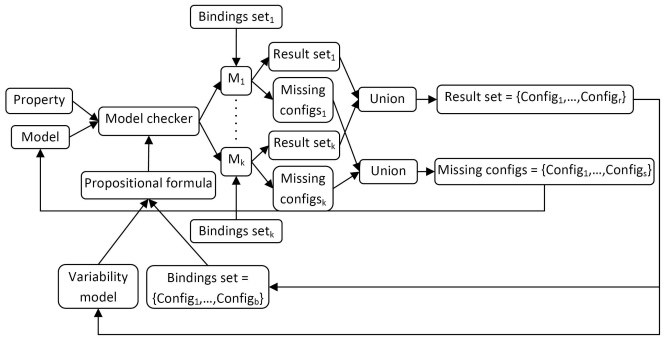


Fig. 1. An overview of the product line verification approach in presence of variability model

In this paper, we present a comprehensive approach for product line verification in presence of variability models, that is shown in Figure 1. The approach is based on the idea of verifying the entire product family [2]. However, we fill the gap between the two extreme verification approaches (making all the binding decisions before verification, and leaving all the bindings undecided until verification), by allowing the modeler to specify the binding decisions that are already made. These decision are described using a set of configurations. Each element of these configurations may denote the decision that is already made for the corresponding variability, or

may indicate that no decision is made for the variability. A propositional logic formula is extracted from these set of configurations. We also transform the variability model to a propositional logic formula. The resulting formulas are used by model checker to restrict the result of verification to the configurations that preserve the specified bindings decisions, and are valid according to the variability model.

Moreover, we try to avoid state space explosion by distributing the state space of a product family over several machines. To this end, we verify a product family on two machines (or more). On one machine we include a variant, and on the other machine we exclude that variant, by specifying these decisions as the binding decisions in each machine.

The final result of verifying a product family against a property is two sets of configurations: The first set is the result set which contains configurations that satisfy that property. These configurations are valid according the variability model, and preserve the specified binding decisions. The second set contains the missing configurations. These configurations are valid according to the variability model, but they cannot be derived from the current behavioral model. Because, as the model checker encounters variabilities and examines different decisions (by making different paths for them), there is not any path that leads to these configurations. Therefore, they wont appear in the result set even if they satisfy a property.

Finally, We may use the result of verification to revise the variability model, the behavioral model, and the binding decisions. The result set can be used to add additional constraints to the variability model (this may consequently affect the software development models), and to specify bindings based on these configurations. The missing configurations may help the modeler to modify the behavioral model so that these configurations can be derived from it.

The contributions of this paper can be summarized as:

- Considering variability models when verifying product lines, to return valid configurations that satisfy a property as the result of verification, and using the verification result to revise the variability model (which is affecting software development models) subsequently.
- Checking if all of the valid configurations can be derived from the behavioral model of the product family.
- Filling the gap between two product line verification approaches (making all the binding decisions before verification, and leaving all the bindings undecided until verification), by allowing the modeler to specify the binding decisions, and using this feature to distribute the state space of a product family over several machines to avoid state space explosion.

We define a simple behavioral model based on transition systems, named variable transition system (VTS), as a basis to describe our approach. A VTS models variabilities in a product family explicitly, and a configuration vector is used to keep the track of the decisions made for each variability in the model.

The Coffee Machine Example: General Description. We use the product family of coffee machines as a running example in the rest of this paper. Figure 2 shows the variability model (based on [1] notations) of the coffee machine example. A coffee machine may serve coffee, tea, and water. More specifically, each coffee machine should serve coffee, and at least one other drink. Moreover, extra milk and sugar, can be added to coffee and tea in some types of coffee machines. In addition, a machine cannot serve water, while having an option for extra milk and sugar, due to the space limitations of a coffee machine. □

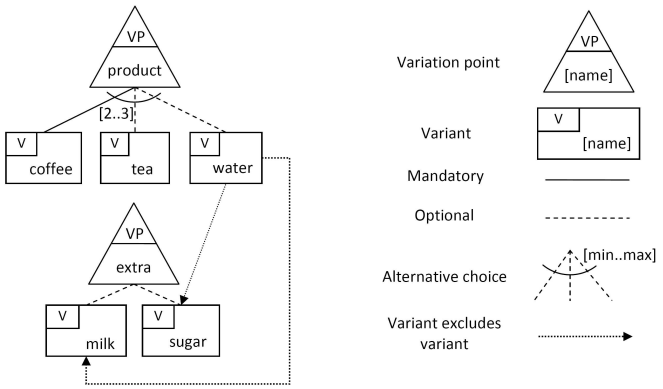


Fig. 2. The variability model of the coffee machine example

This paper is structured as follows. In Section 2, we discuss the related work. In Section 3, we define the variability models, and describe their transformation to propositional logic. Section 4, discusses configurations, their validity, and extraction of a propositional logic formula from a set of configurations. In Section 5, the variable transition system and its semantic are introduced. Section 6 presents the way that the binding decisions, and variability model constraints are used when generating configurations. In Section 7 we describe how the missing configurations can be found. Section 8, explains how the state space of the product family is distributed. In Section 9, it is described that how we can revise the variability model, and the binding decisions based on the results of verifications, and in the last section we conclude the work.

II. Related Work

Recently, several approaches have been developed for formal modeling and verification of product lines. The formal modeling techniques [3], [4] try to model the behavior of a product family. In [3], the authors define modal I/O automata, which is an extension of interface automata [5], and can be used for development of the behavioral model of product lines. In this work, the *may* transitions are used to model variability in the behavior. Each *may* transition can be included in a model or excluded from it, which results in different behaviors. Another work which focuses on product line behavior modeling is [4]. In this work a product family is modeled using input-enabled alternating transition systems. The authors design a general model with all of the functionality and each product is characterized by defining an environment for it. The environment indicates the inputs that the product can

receive, and the outputs of the product that are important for the environment. However, these approaches do not focus on verifying product lines against properties.

The purpose of formal verification techniques [6], [2] is verifying product lines against a number of properties. In [6], the notion of reusable verification models is introduced. A reusable verification model is used to obtain the verification model for a specific test scenario systematically. The reuse of verification models in product line development enables the systematic verification of products. However, in this approach, each product should be verified separately.

In [2], the authors introduce PL-CCS as an extension of CCS, to model the behavior of a product family. In this work, the entire product family is verified against a property and the result is the set of products that satisfy that property. However, this approach does not consider variability models, therefore the result of verification may contain products that are not meaningful or allowed.

III. Variability Model

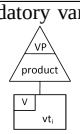
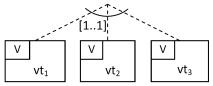
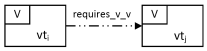
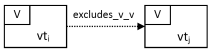
A variability model represents the variability information of a product line family explicitly. There are many works discussing modeling variability, including [1], [7]. In this work, we consider a variability model based on [1] notations, including the basic elements that are required for modeling variability, and are common among the variability models. This variability model consists of a number of variation points, each having a number of optional or mandatory variants associated to them. There may also be constrains on the minimum and maximum number of variants that can be bound to a variation point. In addition, there may be requires/excludes dependencies between two variants.

For simplicity, we assume that all of the variation points are included in a model, and we only decide about the inclusion or exclusion of variants. The exclusion of a variation point can be modeled by exclusion of all of its associated variants. Consequently, "variant excludes variation point" dependency, can be modeled as a set of excludes dependencies from the variant to all of the variants associated to the variation point. We can model, "variation point excludes variant" dependency, and "variation point exclude variation point" dependency, in a similar way. "Variation point requires a variant" dependency can be modeled by a set of requires dependencies from all of the associated variants of the variation point to the variant.

In [8], [9], [10], [11], a number of approaches for transforming feature models to propositional logic are presented. We transform a variability model to a corresponding propositional formula in a similar way. In this work, we consider variability models instead of feature models, as it provides a cross-sectional view of the variability across all software development artefacts [2].

Table I shows the rule set to transform each element of the variability model to propositional logic. For transformation, we consider a boolean variable for each variant (first rule of Table I). The value of these variables indicates if the variant

TABLE I
The rule set for transforming each element of the variability model to propositional logic

	Element of variability model	Propositional formula
1	variant 	Boolean variable vt_{B_i}
2	Mandatory variant 	vt_{B_i}
3	Minimum number of variants 	$[(vt_{B_1}) \vee (vt_{B_2}) \vee (vt_{B_3})]$
4	Maximum number of variants	$\neg[(vt_{B_1} \wedge vt_{B_2}) \vee (vt_{B_1} \wedge vt_{B_3}) \vee (vt_{B_2} \wedge vt_{B_3})]$
5	Variant requires other variant 	$vt_{B_i} \rightarrow vt_{B_j}$
6	Variant excludes other variant 	$vt_{B_i} \rightarrow \neg vt_{B_j}$

is included or excluded. The ultimate propositional formula is constructed as the conjunction of:

- The mandatory variants (rule 2).
- Disjunction of all of the possible conjunction clauses involving k variants from m variants, if we should select at least k variants from m variants (rule 3 where $k = 1$ and $m = 3$).
- Negation of the disjunction of all of the possible conjunction clauses involving $k + 1$ variants from m variants, if we should select at most k variants from m variants (rule 4 where $k = 1$ and $m = 3$).
- Implications from variants to other variants (or their negation), representing the requires and excludes dependencies (rule 5, rule 6).

The Coffee Machine Example: The Propositional Formula. The propositional formula of the variability model of the coffee machine (Figure 2), is the conjunction of: $coffee_B$ as $coffee$ is a mandatory *variant* of $product$, disjunction of the conjunction of the boolean variables of each two *variants* of $product$, as at least two *variants* should be bounded to $product$, and two implications from $water_B$ to $\neg milk_B$ and $\neg sugar_B$ as $water$ excludes these *variants*. Ultimately, the propositional formula of the coffee machine example becomes:

$$
\begin{aligned}
PF_{CM} = \quad & (coffee_B) \wedge \\
& [(coffee_B \wedge tea_B) \vee (coffee_B \wedge water_B) \vee \\
& (tea_B \wedge water_B)] \wedge \\
& (water_B \rightarrow \neg milk_B) \wedge \\
& (water_B \rightarrow \neg sugar_B) \quad \square
\end{aligned}
$$

IV. CONFIGURATIONS

Individual products can be derived from a variability model by bounding a number of variants to each variation point. We use a configuration vector (similar to [2]), $Config \in \{I, E, U\}^m$, to keep track of inclusion or exclusion of the variants (assuming that variability model contains m variants totally). The decision (I: included, E: excluded, U: unknown) that is made for vt_i is represented by $Config_i$. A configuration is *resolved* if it does not include any unknown decisions, otherwise, it is *undecided*.

A configuration vector with m variants, leads to 3^m different configurations. However, all of the possible configurations may not be valid according to the variability model. Having a configuration $Config$, and the propositional formula associated to a variability model VM, we can check if $Config$ is valid according to VM. For this purpose, we assign value to each boolean variable, vt_{B_i}, in the propositional formula according to the $Config$:

$$vt_{B_i} = \begin{cases} true, & Config_{vt_i} = I \\ false, & Config_{vt_i} = E \\ ?, & Config_{vt_i} = U \end{cases}$$

Then, we check if there exists a $true/false$ substitution for variables with value ? which makes the whole propositional formula true, using a SAT-solver such as [12]. The configuration $Config$ is valid according to VM if such substitution exists. We define the function $Validity$ to check the validity of a configuration based on a propositional formula:

$$Validitiy: \ Config \times PF \rightarrow \{true, false\}$$

In addition to the variability model, we may add additional constraints on the validity of a configuration. We may assume that a number of binding decisions are already made (inclusion or exclusion of some of the variants). A resolved/undecided configuration is used to describe these binding decisions. This configuration should be valid according to the variability model. Obviously, in a resolved configuration, all of the binding decisions are made, and it represents a single product. We can use a set of configurations, named the *Bindings set*, to describe different options for binding decisions.

A configuration, $Config$, conforms to the binding decisions that are previously made by the configuration $Config'$, if it refines $Config'$ ($Config \sqsubseteq Config'$):

$$(Config_i = Config'_i) \vee (Config'_i = U) \quad \forall \ i \in \{1, ..., m\}$$

A configuration conforms to the Bindings set, if it conforms to at least one configuration of the set. We can derive a propositional formula, in disjunction normal form (DNF), from the configurations of the Bindings set. The formula is used to check if a configuration conforms to the Bindings set. To this end, each configuration $Config \in \{I, E, U\}^m$ is converted to a conjunction composed of vt_{B_i}s, and their negations. A boolean variable appears in the conjunction clause as vt_{B_i} if $Config_i = I$, and as $\neg vt_{B_i}$, if $Config_j = E$, respectively. The ultimate propositional formula is the disjunction of these conjunction clauses. The conformance of a configuration to the Bindings set can be checked, using the resulted formula, in a way similar to checking the validity of a configuration regarding to a variability model.

The Coffee Machine Example: Configurations. The configuration vector $Config_{CM}$ of the coffee machine may lead to 3^5 different configurations, as there are 5 variants in the variability model. We assume that:

$$vt_1 = coffee \quad vt_2 = tea \quad vt_3 = water$$
$$vt_4 = milk \quad vt_5 = sugar$$

The following configurations are two possible configurations:

$$Config^1 = \{E, I, I, E, E\}$$
$$Config^2 = \{I, U, E, I, E\}$$

$Config^1$ is a resolved configuration. However, $Validity(Config^1, PF_{CM}) = false$, as the $coffee$ which is a mandatory variant is excluded from the model (the $coffee_B$ does not hold in PF_{CM}). $Config^2$ is an undecided configuration as $Config_2 = U$, which results in $tea_B = ?$. In addition, it is a valid configuration, as replacing ? by $true$ (including tea), makes $PF_{CM} = true$.

Let assume that it is already decided to include the tea, and exclude the $milk$, or to include the $milk$ and the $sugar$. Thus, the Bindings set, and its corresponding propositional formula are:

$$Bindingsset = \{\{U, I, U, E, U\}, \{U, U, U, I, I\}\}$$
$$PF_{Bindings} = [(vt_2 \wedge \neg vt_4) \vee (vt_4 \wedge vt_5)]$$

Finally, $Config^1$ conforms to the Binding set, as it conforms to $\{U, I, U, E, U\}$ (although it does not conforms to $\{U, U, U, I, I\}$). In contrast, $Config^2$ does not conform to the Binding set. □

V. VARIABLE TRANSITION SYSTEMS

In this section we introduce the notion of transition system with variability, named variable transition system (VTS) as a basis for describing our approach, by presenting its formal definition, and its semantics.

A. Formal Definition

Variable transition systems are similar to simple transition systems, except that in VTS, we distinguish a special set of actions, named *variant selection* actions, to model variability. Each variant selection action explicitly indicates the inclusion of a variant from a variability model with m variants. A VTS is defined as a five tuple $(S, I, Config, A, A_{vs}, \rightarrow)$ where:

- S is a set of states.
- $I \subseteq S$ is a set of initial states.
- $Config \in \{I, E, U\}^m$ is a configuration vector.
- A is a set of actions.
- A_{vs} is a set of m *variant selection* actions such that $A \cap A_{vs} = \emptyset$ and $A_{all} = A \cup A_{vs}$.

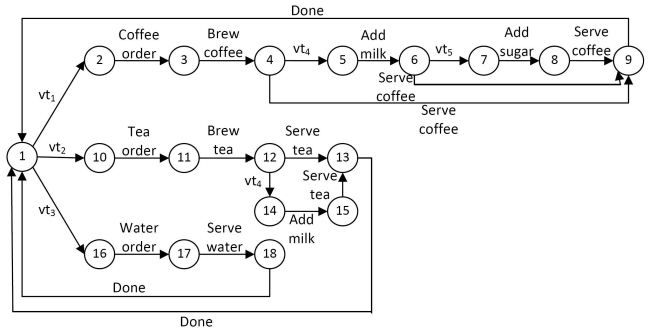


Fig. 3. A VTS modeling the product family of the coffee machine example

- $\rightarrow \subseteq S \times A_{all} \times S$ is a transition relation.

For simplicity, we refer to the transitions: $\rightarrow \subseteq S \times A_{vs} \times S$, as *variant selection transition*, in the rest of the paper.

The Coffee Machine Example: VTS. Figure 3 shows a VTS that models the product family of the coffee machine example (VTS_{CM}). In VTS_{CM}, $A_{vs} = \{vt_1, vt_2, vt_3, vt_4, vt_5\}$. □

B. Semantics

A VTS can be transformed to a single transition system. The resulting transition system is similar to PL-LTS that is introduced in [2]. We define it as a three tuple $TS = (T, Init, Act, \rightarrow')$ where:

- $T = S \times Config$ is a set of states.
- $Init \in T$ is the initial state.
- $Act = A \cup A_{vs}$ is a set of actions.
- $\rightarrow' \subseteq T \times Act \times T$ is a transition relation.

More specifically, we transform a VTS to a TS by deriving the transition relation $\rightarrow'$ from the VTS using a set of rules defined in the following.

Generally, for each state $s_i \in S$ of a VTS there may be a number of possible transitions as $s_i \xrightarrow{a} s_j$ such that $a \in A_{all}$ and $s_j \in S$. When $a \in A$, the transition may be taken, without affecting the configuration vector:

$$\frac{s_i \xrightarrow{a} s_j \quad a \in A}{\langle s_i, Config\rangle \overset{a}{\rightarrow}' \langle s_j, Config\rangle} \tag{1}$$

When $a \in A_{vs}$ (denoted by a_{vs_k}), we may include the variant in the model, if it is not already excluded ($Config_k \neq E$), or exclude it, if it is not already included ($Config_k \neq I$). Then, we should update the configuration vector according to our decision. Replacing the k^{th} element of the configuration vector by I or E is denoted by $Config_k/I$ or $Config_k/E$, respectively. We define the following rule to include a variant, by means of taking a variant selection transition in the model:

$$\frac{s_i \xrightarrow{a_{vs_k}} s_j \quad a_{vs_k} \in A_{vs} \quad Config_k \neq E}{\langle s_i, Config\rangle \overset{a_{vs_k}}{\rightarrow}' \langle s_j, Config_k/I\rangle} \tag{2}$$

When excluding a variant from the model, the VTS remains in the same state (using a τ transition):

$$\frac{s_i \xrightarrow{a_{vs_k}} s_j \quad a_{vs_k} \in A_{vs} \quad Config_k \neq I}{\langle s_i, Config\rangle \overset{\tau}{\rightarrow}' \langle s_j, Config_k/E\rangle} \tag{3}$$

It should be noted that, a variant may be excluded implicitly as well. In this case, the corresponding decision remains unknown in the configuration vector, because the variant selection transition cannot be reached according to the behavioral model. Therefore, practically, the variable behavior is excluded from the model.

Using the above rules, a TS can be derived from a VTS and it can be used to verify the behavioral model against properties. The result of verification is the set of configurations that satisfy a property.

The Coffee Machine Example: Semantics. By applying the defined semantic rules on the VTS of the coffee machine example, the following sample path can be obtained:

$$\begin{aligned}
&\langle 1, \{U,U,U,U,U\}\rangle \xrightarrow{vt_1} \langle 2, \{I,U,U,U,U\}\rangle \xrightarrow{Coffeeorder} \\
&\langle 3, \{I,U,U,U,U\}\rangle \xrightarrow{Brewcoffee} \langle 4, \{I,U,U,U,U\}\rangle \xrightarrow{\tau} \\
&\langle 4, \{I,U,U,E,U\}\rangle \xrightarrow{Servecoffee} \langle 9, \{I,U,U,E,U\}\rangle \xrightarrow{Done} \\
&\langle 1, \{I,U,U,E,U\}\rangle \xrightarrow{vt_2} \langle 10, \{I,I,U,E,U\}\rangle \xrightarrow{Teaorder} \\
&\langle 11, \{I,I,U,E,U\}\rangle \xrightarrow{Brewtea} \langle 12, \{I,I,U,E,U\}\rangle \xrightarrow{Servetea} \\
&\langle 13, \{I,I,U,E,U\}\rangle \xrightarrow{Done} \langle 1, \{I,I,U,E,U\}\rangle \xrightarrow{vt_3} \\
&\langle 16, \{I,I,I,E,U\}\rangle \xrightarrow{Waterorder} \langle 17, \{I,I,I,E,U\}\rangle \xrightarrow{Servewater} \\
&\langle 18, \{I,I,I,E,U\}\rangle \xrightarrow{Done} \langle 1, \{I,I,I,E,U\}\rangle
\end{aligned}$$

The configuration vector won't change from $\langle 1, \{I,I,I,E,U\}\rangle$, further in the above path. Therefore, vt_5 is implicitly excluded from the model, in this sample path. We highlight two inconsistencies that exist in this path:

1) This path results in a coffee machine which can serve coffee, tea, and water. Moreover, it serves water as well as extra milk. However, such coffee machine is not allowed.
2) This VTS cannot model a coffee machine that has not the option of extra milk, but is capable of adding extra sugar. However, such coffee machine is allowed according to the description of the coffee machine families. □

VI. Applying Variability Model and Bindings Constraints

In this section, we describe our approach for applying the constraints that are implied to resulted configurations because of the variability model and the Bindings set. A trivial idea to apply these constraints is verifying the product family, and then checking the validity of each configuration that is appeared in the result set. However, in this way the model checker generates many paths that are not useful as they lead to invalid configurations. Therefore, we involve these constraints in the semantics of the VTS, to avoid generating useless paths.

A. Modifying The Semantics of VTS

We add a condition to the defined rules to check if including or excluding a variant can lead to a valid configuration which also conforms to the Bindings set. To this end, the validity and the conformance of the configuration vector should be checked. For simplicity, we define a general propositional formula, named GPF, as $GPF = PF_{VM} \wedge PF_{Bindings}$, where PF_{VM} and $PF_{Bindings}$ are the propositional formulas of the variability model and the Bindings set, respectively. Finally, we add the condition, $Validity(Config, GPF) = true$, to the rule set which results in the following new rules:

$$\frac{s_i \overset{a_{vs_k}}{\rightarrow} s_j \;\; a_{vs_k} \in A_{vs} \;\; Config_k \neq E \;\; Validity(Config_k/I, GPF) = true}{\langle s_i, Config\rangle \overset{a_{vs_k}}{\rightarrow'} \langle s_j, Config_k/I\rangle}$$

$$\frac{s_i \overset{a_{vs_k}}{\rightarrow} s_j \;\; a_{vs_k} \in A_{vs} \;\; Config_k \neq I \;\; Validity(Config_k/E, GPF) = true}{\langle s_i, Config\rangle \overset{\tau}{\rightarrow'} \langle s_j, Config_k/E\rangle}$$

Using the new rules, the behavioral model generates only valid resolved configurations. However, the undecided configurations may be still invalid. The reason is implicit exclusion of some variants. An undecided configuration is valid, if we can substitute its unknown values with inclusion and exclusion, such that it makes the propositional formula true. However, there may not be any path in the VTS that leads to the configuration with the substituted values. Therefore, such configurations appear in the result set, because there is some substitution for them that makes them valid, although there is not any path in VTS which realizes the substitution. To solve this problem, we should replace the unknown values with exclusion, in these configurations (as they are implicitly excluded), and check their validity and conformance again.

The Coffee Machine Example: New semantics. By applying the new semantic rules on the VTS of the coffee machine example, $\{I, I, I, E, U\}$ cannot be derived by the previous sample path, as it is not valid according to the variability model: After reaching to $\langle 1, \{I, I, U, E, U\}\rangle$, $water$ is excluded by taking a τ transition, leading to $\langle 1, \{I, I, E, E, U\}\rangle$. Moreover, the resulted configuration conforms to the Bindings set, $Bindings\ set = \{\{U, I, U, E, U\}, \{U, U, U, I, I\}\}$, as it conforms to $\{U, I, U, E, U\}$. □

VII. Finding the Missing Configurations

As mentioned in the previous section, a behavioral model may not generate all of the configurations that are valid according to the variability model. This may cause further problems in the verification step, because some of the valid configurations that satisfy a property, may not appear in the result set.

As mentioned earlier, an unknown value that appears in a configuration of the result set, $Config_k = U$, represents the implicit exclusion of the variant, vt_k, from the model. We can conclude that vt_k cannot be included in the $Config$, therefore, $Config_k/I$ is missed from the model practically. Based on this idea, we find the missing configuration. To this end, first we extract the ultimate undecided configurations that can be derived from the behavioral model, and cannot be refined anymore:

$$\begin{aligned}&Config \in Undecided\ Configs \;\; if\\ &(\exists\ k \mid Config_k = U)\ and\\ &(\exists\ maximal\ path\ \pi = \langle s_1, Config^1\rangle, \langle s_2, Config^2\rangle, ...\\ &such\ that\ (\exists\ i > 0 \mid Config^i = Config)\ and\\ &(\nexists\ j > i \mid Config^j \sqsubseteq Config^i))\end{aligned}$$

The missing configurations are extracted from the undecided configurations, by substituting the undecided values in these configurations with included values, and checking their validity according to the variability model. A configuration $Config$, with l undecided values, leads to $2^l - 1$ possible missing configurations: Each of the undecided values may be substituted by the included or excluded value to produce all of the possible combinations, which results in 2^l possible configurations. However, we should ignore the configuration in which all of the undecided values are replaced by excluded. Because this configuration is practically derived from the model, as we replace all of the undecided values with the excluded values in the result set. Therefore, $2^l - 1$, possible missing configurations are extracted from $Config$. Finally, each of these $2^l - 1$ configurations should be validated according to the variability model, and the valid ones are added to the missing configurations set.

The missing configurations are presented to the modeler, and they can be used to modify the behavioral model so that it can generate these configurations.

The Coffee Machine Example: Finding the missing configurations. The configuration $\{I, I, E, E, I\}$, cannot be derived from VTS_{CM} as by excluding extra milk variant from the model, the extra sugar variants will be excluded from the model implicitly. Considering our approach for finding the missing configurations, $\{I, I, E, E, U\}$, is an undecided configuration that cannot be refined further according to the behavioral model. This configuration has 1 undecided value, leading to $2^1 - 1 = 1$ possible missing configuration (Config = $\{I, I, E, E, I\}$). As $Config$ is valid according to the variability model, it is categorized as a missing configuration.□

VIII. Distributing the State Space of Product Families

We take advantage of the Bindings set to reduce the required time and space for model checking, and consequently avoid the state space explosion.

To this end, the model checking process of the entire product family is distributed based on including or excluding variants, over several machines. The state space of a product family with m variants, can be distributed over 2 machines, by including a variant vt_i on one machine, and excluding that variant on another machine, using the Bindings sets. If we assume that the Bindings set of the product family is BS, then each configuration of the result set should conform to the BS, as

well as the Bindings set of the first and second machines which are defined as:

$$BS_{M_1} = \{Config\}$$
$$BS_{M_2} = \{Config'\}$$

where

$$Config_k = \begin{cases} \text{I}, & k = i \\ \text{U}, & k \neq i \end{cases} \qquad Config'_k = \begin{cases} \text{E}, & k = i \\ \text{U}, & k \neq i \end{cases}$$

It should be noted that, $Config$ and $Config'$, should be valid according to the variability model, and should conform to the Bindings set of the product family. In this way, we can distribute the model checking process over 2^k machines, by considering the combinations of inclusion and exclusion of k variants. Selecting no variant ($k = 0$) is equivalent to verifying the entire product family, and selecting all of the variants ($k = m$) is equivalent to verifying each product separately. The final result of model checking is the union of the result sets of each machine.

In this way, Each machine results in the set of configurations that satisfy the property, and the set of missing configurations. The ultimate result of verification is the union of the results of each machine.

The Coffee Machine Example: Distributing the state space. In the coffee machine example, we may consider tea to distribute the state space. If we assume that the Bindings set is empty in the first place, then the Bindings set for the first and second machines are $BS_{M_1} = \{\{U, I, U, U, U\}\}$, and $BS_{M_2} = \{\{U, E, U, U, U\}\}$, respectively.

IX. Revising the Variability Model and Binding Decisions Based on Verification Results

A. Revising the Variability Model

We can enrich the variability model based on the result set. For this purpose, we convert the configurations of the result set to a propositional logic formula (as described in Section 4) and use the formula to extract additional constrains for the variability model.

The propositional logic formula is in the disjunction normal form. Therefore, we can extract variability constraints from it using a straightforward method: We add a variation point to the variability model which its number of variants is equal to the number of conjunction clauses in the formula, and at least one variant should be bound to it (minimum number of variants is 1). Each variant of this variation point corresponds to one of the conjunction clauses, and requires the variants that appear in the its corresponding conjunction as vt_{B_i}, and excludes the variants that appear as $\neg vt_{B_i}$.

In this way, the modeler may add a variation point to the variability model, for a property that has been verified, when the focus is developing products that satisfy that property.

The Coffee Machine Example: Revising the variability model. If we consider the property "the milk is added to the coffee every so often", the result of verification would be the configurations $\{I, I, E, I, E\}$ and $\{I, I, E, I, I\}$. The corresponding propositional formula of these configurations is $[(vt_{B_1} \wedge vt_{B_2} \wedge \neg vt_{B_3} \wedge vt_{B_4} \wedge \neg vt_{B_5}) \vee (vt_{B_1} \wedge vt_{B_2} \wedge \neg vt_{B_3} \wedge vt_{B_4} \wedge vt_{B_5})]$. If we want to focus on developing products in the product line, that satisfy this property, we should add a variation point as shown in Figure 4, to the variability model of the coffee machine. □

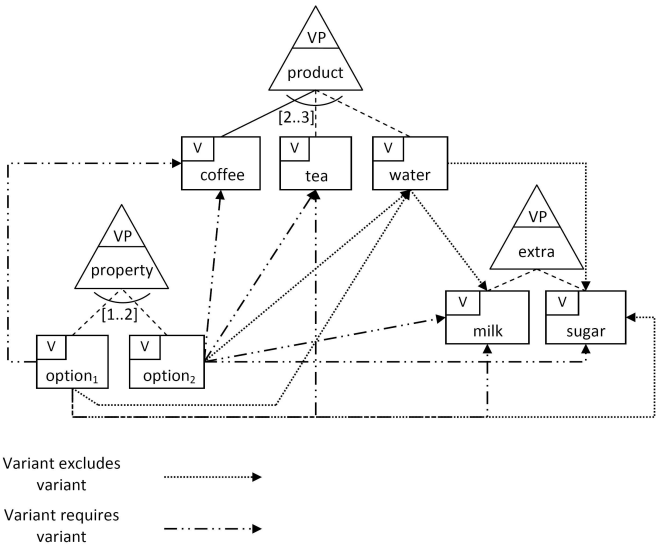


Fig. 4. The variability model after modification

B. Revising the Bindings Set

The configurations that appeared in the result set of verification of one property can be used in the Bindings set when verifying another property.

The result of verifying a product family against a property, is the set of configurations that satisfy that property. In other word, the result is the different binding decisions that can be made, so that the resulted product satisfies the property. Therefore, this set can be used in the Bindings set for verifying of a second property. Consequently, the result of verification of the second property, satisfy the first property as well as the second property.

The Coffee Machine Example: Revising the binding sets. A property for the coffee machine is that "the water is ordered every so often, and when ordered it is finally served". A second property is that "the coffee is ordered every so often, and when ordered it is finally served". We can use the result of verifying the first property as the Bindings set when verifying the second property. In this case, the configurations including the extra milk and extra sugar wont appear in the result, as they do not satisfy the first property (although they satisfy the second property). □

X. Conclusion

In this paper we presented an approach for product line verification in presence of variability models. Moreover, we allowed the modeler to specify the binding decisions that are already made, for the model checker. In this way, we filled the gap between the two product line verifications approaches.

The result of verification of the second approach is the set of products that satisfy a property.

To involve the variability model and the bindings in verification, we transformed them to propositional formulas, which are used by the model checker. In this way, the result of verification of a product family behavioral model against a property, is the set of products that satisfy the property, are valid according the variability model constraints, and preserve the binding decisions. In addition, we presented the missing products to the modeler, which are products that cannot be derived from the behavioral model of the product family. Therefore, Although they may satisfy the property, they wont appear in the verification result.

Finally, the result of verification can be used to revise the behavioral model, variability model, and the bindings: The modeler can use the missing products for modifying the product line behavioral model such that it can generate the missing products as well. The set of product that satisfy a property can be used to add additional constrains to the variability model, and to specify the binding decisions.

In this work, we focused on the theoretical aspect of verification of product lines in presence of variability models. For the future work, we are planning on investigating the cost of the approach, and reducing it using efficient algorithms. In addition, we will apply the product line verification techniques, including the verification approach, on higher level modeling languages.

REFERENCES

[1] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[2] A. Gruler, M. Leucker, and K. Scheidemann, "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, pp. 113–131.

[3] K. G. Larsen, U. Nyman, and A. Wasowski, "Modal I/O automata for interface and product line theories," in *ESOP*, 2007, pp. 64–79.

[4] ——, "Modeling software product lines using color-blind transition systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, pp. 471–487, 2007.

[5] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, pp. 109–120.

[6] T. Kishi, N. Noda, and T. Katayama, "Design verification for product line development," in *SPLC*, 2005, pp. 150–161.

[7] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A framework for modeling variability in software product families," in *SPLC*, 2004, pp. 197–213.

[8] M. Mannion, "Using first-order logic for product line model validation," in *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*. London, UK: Springer-Verlag, 2002, pp. 176–187.

[9] W. Zhang, H. Zhao, and H. Mei, "A propositional logic-based method for verification of feature models," in *ICFEM*, 2004, pp. 115–130.

[10] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*, 2005, pp. 7–20.

[11] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC '07: Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC '01: Proceedings of the 38th annual Design Automation Conference*. New York, NY, USA: ACM, 2001, pp. 530–535.