Logic-Based Analysis of Product Line Variant Model

Shamim Ripon1*, Sk. Jahir Hossain1, and Touhid Bhuiyan2

1Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh
Email: *dshr@ewubd.edu
2Department of Software Engineering, Daffodil International University, Bangladesh
Email: t.bhuiyan@daffodilvarsity.edu.bd

Abstract—Logical representation of feature diagram of product line and formal verification of requirements gained much interest in recent years. Several proposals have been made to model product line features. This paper presents a software product line (SPL) model consisting of a variant model and a decision model. Such model provides the facilities to customize products from product line model alleviating the very concept of reuse of common features throughout product family. The variant and decision models of SPL are based on tabular mechanism. The tabular approach lacks logically sound formal definition and hence not amenable for formal verification. We present a logical representation of the variant model by using first order logic. The logical representation provides a precise and rigorous formal interpretation of the variant model. The table based decision mechanism can now be verified logically. A case study of Hall Booking System product line is used to illustrate our approach.

Index Terms—Product line, variant model, First Order Logic, Tabular method

I. INTRODUCTION

Designing, developing and maintaining a good software system is a challenge still in this 21st century. Reusing existing good solutions for the development of new application is now one of the central focuses of software engineers. Building software systems from previously developed components saves cost, time of redundant work, and improves the system and its maintainability. A new software development paradigm, software product line [1], is emerging to produce multiple systems by reusing the common assets across the systems in the product line. However, the idea of product line is not new. It dated back in 1976 when Parnas [2] proposed modularization criteria and information hiding for handling product line.

A software product line is a set of software-intensive systems that share a common and managed set of features satisfying the specific needs of a particular market segment or mission and that are developed in a prescribed way [1]. Common assets among the products are the basis for software product line. The common assets often include the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance model, etc. Different product line members may differ in functional and non-functional requirements, design decisions, runtime architecture and interoperability (component structure, component invocation, synchronization, and data communication), platform, etc. Two basic processes are integrated into the product line approach: the abstraction of the commonalities and variabilities of the products considered (development for reuse) and the derivation of product variants from these abstractions (development with reuse) [3].

Presently, product line technology is a way of improving the software development lifecycle and the reuse by providing facilities to reuse the model of the product line. Both industry and academia have shown much
Figure 1. Customized models derived from product line model

interest in handling product line in application domains such as business systems, avionics, command and control systems etc. Today most of the effort in product line development are relating to architecture [4], detail design and code. The main idea of software product line is to explicitly identify all the features which are common to all members of the family as well as those that are different, and then arrange them in a model. This implies a huge model which will help the stakeholders to be able to trace any design choices and variability decisions as well. Finally, the derivation of the product is done by selecting the required variants from the model and configuring them according to product requirements.

Common requirements among all family members are easy to handle as they simply can be integrated into the family architecture and are part of every product of the family. But problem arises from the variant requirements among family members. In a product line, usually variants are modeled using feature diagram, inheritance, templates and other techniques. In comparison to analysis of a single system, modeling variants adds an extra level of complexity to the domain analysis. In any product line model, the same variant has occurrences in different domain model views. Different variants have dependencies on each other. Tracing multiple occurrences in different model views of any variant and understanding the mutual dependencies among variants are major challenges during domain modeling. While each step in modeling variant may be simple but problem arises when the volume of information grows. When the volume of information grows the domain models become difficult to understand. The main problems are the possible explosion of variant combinations, complex dependencies among variants and difficulty in tracing variants from the domain model down to the specification of a particular product. As a result, the impact of variant model becomes ineffective on domain model. Therefore, product customization from the product line model becomes unclear and it undermines the very purpose of domain model.

The objective of this work is to provide an approach to modeling variants in the domain model of a product line. This model will carry all the variant related information like specifications, interdependencies, origins of variants, etc.

In developing product line, the variants are to be managed in domain engineering phase, which scopes the product line and develops the means to rapidly produce the members of the family. It serves two distinct but related purposes, firstly, it can record decisions about the product as a whole including identifying the variants for each member and secondly, it can support application engineering by providing proper information and mechanism for the required variants during product generation (Fig. 1).

We consider a domain model which includes default domain view, a variant model and customization requirements. Default domain views describe typical system in a domain. Default domain views are the starting point for understanding the scope of the product line, i.e., the range of systems in the domain we wish to consider. We draw a model to represent the variants of a product line. The model contains all the variant related information required for customizing any product. After getting the requirements for any particular product of the product line, the product line model collects proper variant information from the variant model. A flexible variant configuration (FVC) tool interprets the variant model and customizes the default domain model by adapting and customizing the default domain according to the particular product requirements. Fig. 2 gives a top level view of the targeted variant model along with its position and activity within product line model.

The left-hand-side of Fig. 2 depicts the product line model comprising of the default model and the variants. A feature diagram can be drawn from the product line model to get an overall picture of the product line.
Figure 2. Variant model within product line model

functionalities. The right-hand-side of the figure mainly depicts the variant model. The variant model is constructed by getting information from the product line. A generic domain model is created by adding the variants to the default model. During this construction, information is collected from the variant model as well as from the product line model. Finally, the required product model is developed by customizing the generic domain model while managing the variants according to the requirements.

We represent the variant model by using a unified tabular approach [5][6]. The table contains all the variant related information along with the commonalities of the domain. A decision table is constructed based on the available variant information. The decision table is used to derive customized product from the product line model. Feature diagram on the other hand, represents the common and variant requirements in a tree-like structure. However, detailed information regarding variants and their dependencies cannot be represented as conveniently as tabular method. Tabular method can be augmented with feature diagram. The table and the associated decision table do not have any logical (or formal) definition and formal verification and automated consistency checking of the variant model cannot be performed. After modeling the variant table and decision table we provide logical definition of the feature diagram and perform various analysis operations.

The rest of the paper is organized as follows. Section II gives a brief overview of the Hall Booking System case study. Section III describes our mechanisms to model variants by using tabular method. We illustrate the steps of how the feature diagram can be modeled by using first-order logic in Section IV. Various logic-based analysis operations and corresponding rules are defined in this section. We illustrate various example configurations and describe how to verify them logically. Finally, Section V concludes the paper and outlines our future plans.

II. HALL BOOKING SYSTEM OVERVIEW

We use Hall Booking System family to illustrate our variability modeling mechanism. The system is used in academic institutions to reserve tutorial rooms and lecture halls, at companies to reserve meeting rooms, and at hotels to reserve rooms and conference facilities, etc. In another sense, the system can be used for either academic or non-academic purposes. Users can manage their own reservation with the system. The main purpose and the core functionality are similar across the Hall Booking System family; however, there are many variants on the basic theme. One of the basic variants is the charging of booking system. Whenever the system is used for academic purposes, no charge is needed for booking halls, whereas there may be a need to charge for booking halls in other areas. In some systems, there are facilities available for seasonal booking as well as multiple bookings. Our Hall Booking System default models include the functional features, (i) Make reservation, (ii) Delete reservation, (iii) Modify reservation, (iv) Search/Retrieve reservation, (v) Add/Delete a resource (Hall), (vi) Modify a resource and (vii) Search/receive a Hall.

By using the extensions shown in [7] of feature diagram [8], a part of the features of Hall Booking System is shown in Fig. 3. Mandatory features appear in all the members of the family whereas variant features appear
in some members of the family. Variant features are also classified as *Optional, Alternative* and *Or* features. An optional feature may or may not be part of a system. An example of optional feature is *Reservation Charge*. An alternative feature describes the selection of one from many features. An example of alternative feature is *Reservation Mode* which can be either Single or Block. An or-feature describes any of many features. For example a Block Reservation can be made by multiple rooms or multiple times or by both. Variants may depend on other variants. Two types of dependencies are illustrated in this paper: *requires* and *exclude*.

### III. MODELLING VARIANTS

Variant modeling is an important part during designing software product line. In product line, variants have several impact and dependencies on other variants. Any particular product of a product line can be customized from the generic model of the product line. On the way of customization, several variants have to be chosen from a list of variants. During selection of variants several queries are raised whose cannot be conveniently represented using FODA [8] and need extra annotation with the model which will sometimes become complex and difficult to handle.

We provide a unified table approach [6] to modeling variants which depicts all the information that FODA describe. This table is a categorization table and helps us selecting variants during application engineering process.

The table is constructed by keeping all the information available in FODA feature diagram. By analyzing the characteristics of variants of a domain we found that all the variants follow certain types of common categorization and all of them have certain behavior. So, if the variants are arranged according to these categorizations and behavior then they can be handled easily in a domain. After identifying the common characteristics and behavior of variants, we arranged the variants in a tabular format. The table contains the following information:

- **Variant** - the name of each variant
- **Variant type** - the type of each variant (Mandatory or Optional)
- **Variant value** - the list of possible values a variant can have
- **Sub domain** - the applicable area of the variant and its values
- **Relation among values** - the relationship among the values of each variant (Alternative, OR)
- **Dependency** - the variant or variant values, on which the variant is dependent.

Usually, there are some complex dependencies among variants which cannot be properly represented by feature diagrams. We used the notations given in [9]. A partial variant table of Hall Booking system is depicted in Fig. 4.

![Hall Booking System feature diagram](image_url)
We derive a decision table from the variant model. Part of the decision table is illustrated in Fig. 5. In the table, each variant is placed in a column, its description, possible choices and traces to the variant are shown in other columns. When inclusion or exclusion of any variant depends on a variant or its values then that variant is placed in the subordinate position showing the values for which it can be chosen. For example, Reservation Mode variant has two possible values either Single or Block. These two values are placed in the subordinate position of the parent variant (Reservation Mode) in the decision table. Similarly, when Block reservation is to be considered, it needs to check its values (Multiple Room and Multiple Time) and these values along with other information will be also be subordinated. For each variant and for each of their values, the decision table can have a corresponding entity.

After collecting requirements from stakeholders, those are checked against the variant model and the decision table. The variability model will guide the application engineer to properly choose the required variants very easily. Therefore, the overall product generation process will be faster and less erroneous.

A. Usage of the table

The decision table guides us to customize variants for a particular product from the domain model. Fig. 6 shows the steps that we follow using the table approach to customize a product from the domain model. By using the table, the customization process becomes very easy. As the table shows all variants and their types, by looking at the table at a glance, it is easy to identify the variants which must be present as well as those which need to be selected. The table shows the applicable sub domain (applicable product) of each variant and their values. Therefore, on the way of customizing any particular product it will be very easy to select only those variants and values which are applicable to that product and those variants which are not applicable can be discarded easily. Along with these, the dependency information also helps in customization steps. If a variant needs to be selected for a product then the dependent variant(s) will also be selected by just looking at the dependency column. Fig. 6 also shows that feature diagram can be used as an alternative approach (or can be augmented) to table based variant model. The decision table can be constructed from the feature diagram. It is mentioned earlier that the tabular representation does not have any logical representation. To derive the decision table in a logically sound and consistent way we use First-Order Logic (FOL) to represent the feature model.
A feature model is a hierarchically arranged set of features. It represents all possible features of a software product line in a single model. The relation between a parent feature (variation point) and its child features (variants) are categorized as Mandatory, Optional, Alternative, Optional Alternative, Or, and Optional Or. The logical representation of feature models facilitates the development of decision table in a formally sound way. We define six types of logical notations [10] to represent all the parts in a feature model using first-order logic. These notations are used to define all possible scenarios of a feature model. The logical representations of these features are defined in Fig. 7.

A. Analysis of Feature Model

From the logical representation of feature model we analyze various scenarios during product customization. We consider a feature model as a graph consists of a set of sub-graphs. Each sub-graph is created separately by defining a relationship between the variation point ($v_p$) and the variants ($v_1, v_2$) by using the expressions shown.
in Fig. 7. A relationship between cross-tree (or cross hierarchy) variants (or variation points) is denoted as a dependency. There are two types of dependencies considered in this paper, inclusion and exclusion: if there is a dependency between $p$ and $q$, then if $p$ is included then $q$ must be included (or excluded). Dependencies are drawn by dotted lines.

**Scenario 1:** If there is a require relation between variants $v_i$ and $v_j$ as shown in Fig. 8(a), then $v_j$ is elected whenever $v_i$ is selected. Adopting the notation in [11] the rule for dependency among variants as well as variation points is defined as follows,

$$
\forall v_i, v_j \cdot \text{type}(v_i, \text{variant}) \land \text{type}(v_j, \text{variant}) \\
\land \text{require}_v(v_i, v_j) \land \text{select}(v_i) \Rightarrow \text{select}(v_j)
$$

**Scenario 2:** From Fig. 8(b), we derive the following rule,

$$
\forall v_i, v_j \cdot \text{type}(x, \text{variant}) \land \text{type}(v_j, \text{variant point}) \\
\land \text{select}(v_j) \land \text{requiresvp}(v_i, v_j) \land \text{select}(x) \Rightarrow \text{select}(v_j)
$$

**Scenario 3:** The following rule is derived from Fig. 8(c)

$$
\forall v_i, v_j, x, y \cdot \text{type}(x, \text{variant}) \land \text{type}(y, \text{variant}) \\
\land \text{variants}(v_i, x) \land \text{variant}(v_j, y) \land \text{common}(y) \\
\land \text{requiresvp}(v_i, v_j) \land \text{select}(x) \Rightarrow \text{select}(y)
$$

**Scenario 4:** When there is an exclude relation between variants (and/or variation point) as shown in Fig. 8(d), only one among them can be selected at a time.

$$
\forall v_i, v_j \cdot \text{type}(v_i, \text{variant}) \land \text{type}(v_j, \text{variant}) \\
\land \text{exclude}_v(v_i, v_j) \land \text{select}(v_i) \Rightarrow \text{notselect}(v_j)
$$
∀v₁, v₂ : type(v₁, variation point) ∧ type(v₂, variation point) ∧ exclude_vp vp(v₁, v₂) ∧ select(v₁) ⇒ notselect(v₂)

Scenario 5: From Fig. 8(e) we derive the following rule,

∀v₁, v₂, x, y : type(x, variant) ∧ type(v₁, variation point) ∧ type(v₂, variation point) ∧ exclude_vp(x, v₂) ∧ select(x) ⇒ notselect(v₂)

Scenario 6: The scenario in Fig 8(f) depicts the following rule,

∀v₁, v₂, x, y : type(x, variant) ∧ type(y, variant) ∧ variant(v₁, x) ∨ variant(v₂, y) ∧ common(y, yes) ∧ requires_vp vp(v₁, v₂) ∧ select(x) ⇒ notselect(y)

Figure 9. (a) Inconsistency checking; (b) False option feature detection and (c) Dead feature detection

B. Analysis Operations

We perform some analysis operations that determine whether the feature model works correctly.

Inconsistency: In Fig. 9(a), v₁, v₂ and v₃ are three variation points where v₁,1 and v₁,2 are variants of v₁ and v₂,1 and v₂,2 are variants of v₂. There exists a require relationship between variant v₁,2 and variation point v₂. As v₁,1 and v₁,2 are mandatory feature whenever v₁ is selected both variants will be selected, and consequently, variation point v₂ will be selected as well due to require relation. However, v₁ and v₂ are alternative features, and both cannot be selected at the same time and it introduces an inconsistency into the feature model.

False optional is a situation where a feature is declared as optional which does not need so. In Fig. 9(b), v₂ is False optional.

Dead feature is a feature that never appears in any legal product from a feature model. As shown in Fig. 9(c) due to exclude relation v₂,1 will never be part of any valid product from the feature model.

C. Analysis Examples

Automatic analysis of variants is already identified as a critical task [8]. Our logical representation can define and validate a number of analysis operations suggested in [12], [13]. In order to construct an instance product from the product line model, TRUE (T) is assigned to the selected features and FALSE (F) to those not selected. These truth values are assigned to the product line model and if TRUE value is evaluated, we call the model as valid otherwise the model is invalid. For convenience, we represent a partial tree of the CAD feature in Fig. 10 which is split into smaller sub-graphs (Fig. 10(b), 10(c) and 10(d)).

Example 1: Suppose the selected variants are v₁, v₁,1, v₂, v₂,1, v₂,2, v₂,3, v₂,4, v₂,5 and v₃,2. We check the validity of the sub-graphs G₁, G₂ and G₃ by substituting the truth values of the variants of the sub-graphs.
As the sub-graphs $G_1, G_2$ and $G_3$ are evaluated to TRUE, the product model is valid. However, variant dependencies are not yet considered in this case. Checking the validity of each sub-graph is not enough for the validity of the whole model. Variant dependencies must also be checked as additional constraints. We evaluate the dependencies of the selected variants and we get,

\[
\text{Dependency: } (v_{2.3.1} \Rightarrow v_{1.1}) \land (v_{2.4} \Rightarrow v_{3.2}) = (T \Rightarrow T) \land (T \Rightarrow T) = T
\]

The truth (T) value of the dependencies ensures the validity of the product instance.

**Example 2:** Suppose the selected variants are, $v_1, v_{1.2}, v_2, v_{2.2}, v_{2.3}, v_{2.4}, v_3, v_{3.1}$. To check whether these input combination builds a valid product we check the validity of the sub-graph $G_1, G_2$ and $G_3$ by substituting the truth values of the variants of the sub-graphs.

\[
G_1 : (v_{1.1} \oplus v_{1.2}) \iff v_1 = T
\]
\[
G_2 : v_2 \iff v_{2.1} \lor v_{2.2} \lor v_{2.3} \lor v_{2.4}
= v_2 \iff v_{2.1} \lor v_{2.2} \lor (v_{2.3.1} \oplus v_{2.3.2}) \iff v_{2.3} \lor v_{2.4}
= T
\]
\[
G_3 : (v_{3.1} \oplus v_{3.2}) \iff v_3 = T
\]

We then evaluate the dependencies of the selected variants,

\[
\text{Dependency: } (v_{2.3.1} \Rightarrow v_{1.1}) \land (v_{2.4} \Rightarrow v_{3.1}) = F
\]

Due to conflict within variant dependencies, the whole graph becomes invalid, which is due to an incorrect selection of input.
V. CONCLUSION

Proper organization and management of product requirements is a key to successful development of software product line. In a domain model, when the volume of information grows due to inclusion of various features, the possible explosion of variant combination becomes inevitable and tracing of proper variant information in the domain model becomes hard. As a result, the impacts on variants on the domain model during customization of any particular product become unclear. A variant model has been presented in this paper which explicitly represents all the variant related information in order to generate any customized product form the domain model. To be able to formally verify the variant configuration and consistency of feature model, we defined six types of variant relations by using first-order logic. Cross tree variants dependencies are defined as well. Such formal definition facilitates the automated decision making during product customization. The various analysis operations suggested in [11],[12] are also addressed here.

To capture domain knowledge and common vocabularies in any field, ontologies have shown itself an acceptable paradigm [14]. It is also necessary to process and exploit knowledge in a computer system. Among the various available approaches for knowledge representation, ontology is a promising solution due to its ability to make the domain knowledge computer readable and processable. Semantic web [15] technology provides a meaningful and shared ontological description of the domain. Web Ontology Language (OWL) [16] is one of the most expressive languages for specifying, publishing and sharing ontologies. We are currently applying semantic web mechanism [17], OWL in particular, to integrate meaningful description and semantic information into SPL feature models.

REFERENCES