Knowledge Based Method to Validate Feature Models

Abdelrahman Osman, Somnuk Phon-Amnuaisuk, Chin Kuan Ho
Center of Artificial Intelligent and Intelligent computing,
Multimedia University, Cyberjaya, Malaysia
abdelrahman.osman.06@mmu.edu.my,somnuk.amnuaisuk@mmu.edu.my,ckho@mmu.edu.my

Abstract

Feature model has been used to support requirements analysis and domain engineering in Software Product Line by representing variability. This paper proposes a knowledge base method to validate feature model. Validation of feature model has been split into two main processes, automated consistency check (by defining rules control the variation selections considering cross-tree constraint dependencies) and the second process is automated error detection(two types of errors, dead feature and inconsistency, was defined and validated). Variability was described among feature model, and then variability was represented as a knowledge base containing predicates and rules. In addition to validation, the proposed method can be used to identified and provide auto support for some operations on the automated analysis of feature models (propagation, cardinality validation, explanation, and optimization).

1 INTRODUCTION

Software Product lines has been defined by Meyer and Lopez as “a set of products that share a common core technology and address a related set of market applications”[1]. *Software product lines* has two main processes; the first process is the domain engineering process which represents problem space and responsible for preparing domain artifacts including *variability*. The second process is the application engineering which aims to consume specific artifact, picking through *variability*, with regard to the desired application specification. One of the useful techniques to represent *variability* is the feature model introduced first in [2]. A particular product-line member is defined by a unique combination of features. The set of all legal feature combinations defines the set of product-line members.

1.1 Feature Models

According to [3], the two most popular definitions of feature models are: i) an end user visible characteristic of a system, and ii) a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept. A *feature model* is a description of the commonalities and differences between the individual software systems in a software product family. In more detail, a *feature model* defines a set of valid feature combinations. Each such valid feature combination can serve as a specification of a software system [4]. Czarnecki defines Cardinality-based feature modeling as integrates a number of extensions to the original FODA notation in [5]. It is very important to producing error-free feature models, including the possibility of providing explanations to the modeler so that errors can be detected and removed; this process is unfeasible to be done manually.

2. Related Work

One of the earlier and famous usages of knowledge based systems in requirement engineering are: "knowledge base software assistant, (KBSA)"[6], and “knowledge Based system for Modeling software system Specifications (KBMS)” [7]. Schlich and Hein proved the needs and benefits of using the knowledge base representation for configuration systems in [8].
Knowledge Acquisition and Sharing for Requirement Engineering (KARE) is a project aims to support Requirement Engineering process with knowledge engineering [9]. A knowledge-based product derivation process [10], [11] is a configuration model that includes three entities of Knowledge Base. The automatic selection provide a solution to complexity of product line variability, but in contrast to our proposed method, the knowledge-based product derivation process does not provide explicit definition of variability notations and for the selection process. In addition to that, knowledge-

based product derivation process is not focused on modeling and validating variability.

Mannion was the first to connect propositional formulas to feature models [12]. Zhang [13] defined a meta-model of feature model using UML core package and he took Mannion's proposal as base and suggested the use of an automated tool support based on the SMV System, Model Checking @CMU. Benavides[14] proposed a method for mapping feature model into constraint solver, his method does not cover dependencies such as require or exclude constraints. Batory [15] proposed a coherent connection between FMs, grammars and propositional formulas; he represented basic FMs used context–free grammars plus propositional logic. Janota [16] used higher-order logic to reasoning about feature models, but there is no real implementation was described. Czarnecki proposed a general template-based approach for mapping feature models in [17]. And he used object- constraint language (OCL) in [18] to validate constraint rules. In contrast to all these models our proposed method defines across-tree constraints between (variation point-variation point), (variation point-variant), and (variant-variant) and can validate cardinality and extended features.

Benavides in [19] presented a survey on the automated analysis of feature models.

Trinidad defined a method to detect dead features based on finding all products and search for features not used in them [20]. He automated error analysis based on theory of diagnosis [21], mapped feature model to diagnose model and used CSP to analyze feature models. Trinidad's method just deals with a dead feature, while our approach deals with three types of errors, inconsistency, dead features, and redundancy. Inconsistency error in feature model was described by Batory [22] as a research challenge. Compare with methods discussed in this literature review this method is first method to deal with inconsistency.

3. Modeling variability using knowledge base rules

A method of modeling variability was proposed to represent domain engineering process as a knowledge base.

First traditional feature model was extended by adding variability notations and modeling domain engineering process as a knowledge base, and then variation point, variants, and constraint dependency rules was represented using predicates. In the next sections we describe extension of feature model using variability notations, how we can automate consistency check using knowledge base rules, and we illustrate how our method can be used to identify and provide auto support for error check and for some operations in feature model analysis process.

3.1. Extending Feature Models by Variability Notations

To maximize the benefits of feature models *variability* notations was characterized from Orthogonal Variability Model (OVM) [23], in feature models as appears in figure 1. Orthogonal Variability Model is one of the useful techniques to represent variability which provides a cross-sectional view of the variability through all software development artifacts. *Variability* - in OVM- is described by variation points, variants and constraint dependencies rules. Constraint dependencies rules as in [24] are: variation point require or exclude variation point, variant require or exclude variant, and variant require or exclude variation point.

Optional and mandatory constraint were defined in figure 1 by original feature model notations [2] ,and constraint dependencies were described using OVM notations.OVM and *feature models* can easily become very complex to model a medium level system, i.e., several thousand of variation points and variants are needed. For this reason we propose an intelligent method to modeling *variability* in *software product line*.

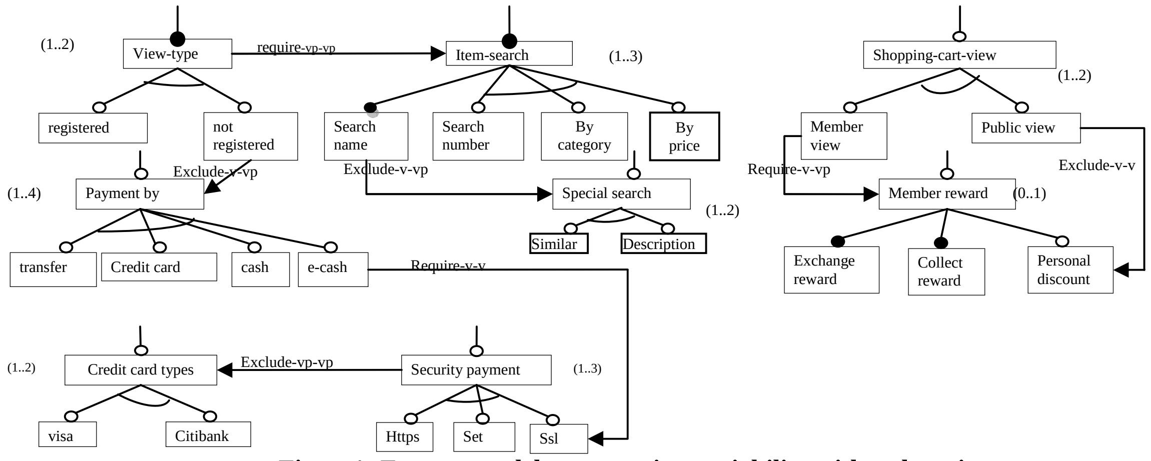

Figure1: Feature model representing variability with e-shopping

Table 1: predicates represent constraint dependency rules in the proposed method.

requires_v_v: Variant requires variant *require_v_v(x,y)| x,y ∈{V}; V= variant*	The selection of a variant *V*1 requires the selection of another variant *V*2 independent of the variation points the variants are associated with. e.g. requires_v_v (ecash, ssl).
excludes_v_v: Variant excludes variant *exclude_v_v(x,y)| x,y ∈{V}; V= variant*	The selection of a variant V1 excludes the selection of the related variant V2 independent of the variation points the variants are associated with .e.g. excludes_v_v(public_view, personal_discount).
requires_v_vp: Variant requires variation point *require_v_vp(x,y)| x,y ∈{V,VP}; V= variant.Vp=variation point*	The selection of a variant V1 requires the consideration of a variation point VP2. *e.g.* requires_v_vp (member_view, member_reword).
excludes_v_vp: Variant excludes variation point *excludes_v_vp(x,y)|x,y ∈{V,VP};V=variant.VP= variation point*	The selection of a variant V1 excludes the consideration of a variation point VP2. *e.g.* excludes_v_vp (not registered, payment_by).
requires_vp_vp: Variation point requires variation point *require_vp_vp(x,y) | x,y ∈{VP}; VP= variation point*	A variation point requires the consideration of another variation point in order to be realized. *e.g.* requires_vp_vp (item_search, view_type).
excludes_vp_vp Variation point excludes variation point *excludes_vp_vp(x,y)|x,y∈{VP}; VP= variation point*	The consideration of a variation point excludes the consideration of another variation point. *.e.g.* excludes_vp_vp (security_payment, credit_card_type).

3.2. Variation points and variants as predicates

In this section variation point and variant were described using predicates: (examples are based on figure 1)

- *type*: Describes the type of feature; variation point or variant. *e.g.*: type (view_type, variationpoint), type (register, variant).
- variant: Identifies the variant of specific variation point. *e.g.*: variant(view_type, not registered)
- *max*: Identifies the maximum number allowed to be selecting of specific variation point. *e.g.* max (payment_by, 4).
- *min*: Identifies the minimum number allowed to be selecting of specific variation point. *e.g.* min (payment_by, 1).

The common variant(s) in a variation point is/are not included in maximum-minimum numbers of selection.

- *common*: describe the commonality of specific feature. *e.g.* common (search_name, yes). If the feature is not common, the second slot in the predicate will become No -as example- common (register, no).

Predicates were used to represent constraint dependency rules between features. Table 1 describes predicates represent constraint dependency rules with examples from figure 1.

Table 2 describe the proposed predicates using Backus–Naur form (BNF).

Table 2: Representation of variant and variation point using Backus–Naur form (BNF).

<type >	::= <variationpoint> |< variant>
<variationpoint>	::=[<name><cardinality><variant> <common>]
<name>	::= <String>
<cardinality>	::= [<min> <max>]
<min>	::= <Digits>
<max>	::= <Digits>
<variant>	::= <variant> <variant>
<variant>	::= [<name> <common>]
<common>	::= <Yes> | <No>

Table 3 shows the representation of *view-type* variation point from figure 1. Table 4 shows the representation of *not registered* variant from figure 1.

Table 3: view-type representation

type (view-type, variationpoint).
variants (view-type, registered).
variants (view-type, not registered).
common (view-type, yes).
min (view-type, 1).
max (view-type, 3).
requires_vp_vp(view-type,search_item).

Table 4: not registered representation

type (not registered, variant).
common(not registered, no).
excludes_v_vp(not registered, payment by).

Table 5: Abstract representation of the main rules in the proposed model

1. $\forall x, y: type(x, variant) \land type(y, variant) \land require\_v\_v(x, y) \land select(x) \Rightarrow select(y)$
2. $\forall x, y: type(x, variant) \land type(y, variant) \land exclude\_v\_v(x ,y) \land select(x) \Rightarrow notselect(y)$
3. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land require\_v\_vp(x, y) \land select(x) \Rightarrow select(y)$
4. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land exclude\_v\_vp(x, y) \land select(x) \Rightarrow notselect(y)$
5. $\forall x, y: type(x, variationpoint) \land type(y, variationpoint) \land require\_vp\_vp(x, y) \land select(x) \Rightarrow select(y)$
6. $\forall x, y: type(x, variationpoint) \land type(y, variationpoint) \land exclude\_vp\_vp(x, y) \land select(x) \Rightarrow notselect(y)$
7. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land select(x) \land variants(y, x) \Rightarrow select(y)$
8. $\exists x\ \forall y: type(x, variant) \land type(y, variationpoint) \land select(y) \land variants(y, x) \Rightarrow select(x)$
9. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land notselect(y) \land variants(y, x) \Rightarrow notselect(x)$
10. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land common(x,yes) \land variants(y, x) \land select(y) \Rightarrow select(x)$
11. $\forall y: type(y, variationpoint) \land common(y,yes) \Rightarrow select(y)$
12. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land variants(y, x) \land select(x) \Rightarrow sum(y,(x)) \leq max(y,z)$
13. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land variants(y, x) \land select(x) \Rightarrow sum(y,(x)) \geq min(y,z)$

3.3 Validation Rules

To validate the selection process, our proposed method triggers rules based on constraint dependencies. With regard to validation process result, the choice is added to knowledge base or rejected, then an explanation of rejection reason is provided and correction actions suggested. At any new solution generated, new fact (select or notselect) added to the knowledge base and backtracking mechanism validates all. At the end of the process all *select* facts represent the product. Table 5 shows the abstract representation of the main rules in the knowledge base. The proposed method contains 13 main rules to validate the selection process based on constraint dependencies.

Rule 1:
For all variant *x* and variant *y*; if *x* requires *y* and *x* is selected, then *y* should be selected.

Rule2:
For all variant *x* and variant *y*; if *x* excludes *y* and *x* is selected, then *y* should not be selected.

Rule 3:
For all variant *x* and variation point *y*; if *x* requires *y* and *x* is selected, then *y* should be selected.
This rule is applicable as well if *y* selected with the same condition:
$\forall x, y: type(x, variant) \land type(y, variationpoint) \land require\_v\_vp(x, y) \land select(y) \Rightarrow select(x)$
For all variant *x* and variation point *y*; if *x* requires *y* and *y* is selected, then *x* should be selected.

Rule 4:
For all variant *x* and variation point *y*; if *x* excludes *y* and *x* is selected, then *y* should not be selected.
This rule is applicable as well if *y* selected with the same condition:
$\forall x, y: type(x, variant) \land type(y, variationpoint) \land exclude\_v\_vp(x, y) \land select(y) \Rightarrow notselect(x)$
For all variant *x* and variation point *y*; if *x* excludes *y* and *y* selected, then *x* should not be selected.

Rule 5:
For all variation point *x* and variation point *y*, if *x* requires *y* and *x* selected, then *y* should be selected.

Rule 6:
For all variation point *x* and variation point *y*, if *x* excludes *y* and *x is* selected, then *y* should not be selected.
Rule 7:
For all variant *x* and variation point *y*, where *x* belongs to *y* and *x* is selected, that means *y* should be selected.
This rule determines the selection of variation point if one of its variants was selected.
Rule 8:
For all variation point *y* there exists of variant *x*, if *y* selected and *x* belongs to *y*, *x* should be selected.
This rule states that if a variation point was selected, then there is variant(s) belong to this variation point should be selected.
Rule 9:
For all variant *x* and variation point *y*; where *x* belongs to *y* and *y* defined by predicate *notselect(y),* then *x* should not be selected.
This rule states that if a variation point was excluded, then none of its variants should be selected.
Rule 10:
For all variant *x* and variation point *y*; where *x* is a common and *x* belongs to *y* and *y* is selected, then *x* should be selected.
This rule states that if a variant is common and its variation point selected then it must be selected.
Rule 11:
For all variation point *y*; if *y* is common, then *y* should be selected.
This rule states that if a variation point is common then it should be selected in any product.
Rule 12:
For all variant *x* and variation point *y*; where x belongs to *y* and *x* is selected, then the summation of *x* should not be less than the maximum number allowed to be selected from *y*.
Rule 13:
For all variant *x* and variation point *y*; where *x* belongs to *y* and *x* is selected, then the summation of *x* should not be greater than the minimum number allowed to be selected from *y*.
Rules 12 and 13 validate the number of variants' selection considering the maximum and minimum conditions in variation point definition. The predicate *sum(y, (x))* return the summation number of selected variants belong to variation point *y.*

From these rules we can define a full common variant, variant included in any product, as:

$$\forall x,y{:}\,type(x,variant) \land type(y,variationpoint) \land variants(y,x) \land common(y,yes) \land common(x,yes) \Longrightarrow full\_common(x)$$

A full common variant is a common variant belongs to common variation point. A common variation point included in any product (rule 11), a common variant belongs to selected variation point should be selected (rule 10).

These rules were implemented using SWI-Prolog [25]. In real implementation we have numerous rules to cover all situations, but all the rules based on the 13 main rules. Definitions and examples were described in the next section (based on figure1) to illustrate the usefulness of our proposed method of modeling variability.

3.4. Operations on the Automated Analysis of Feature Models:

The main operations of automated analysis of feature models defined in [19]. Our proposed method can define and provide auto support for a number of operations (feature Model validation, propagation, explanation, optimization, dead feature detection, inconsistency detection, and cardinality validation).

3.4.1. Propagation. This operation returns a feature model where some features are automatically selected (or deselected).

Definition 1
Selection of variant *n*, *select(n)*, is propagated from selection of variant *x*, *select(x)*, in three cases:

i. $\forall x,y,z,n{:}\,type(x,variant) \land variants(y,x) \land select(x) \land requires\_vp\_vp(y,z) \land type(n,variant) \land variants(z,n) \land common(n,yes) \Longrightarrow select(n).$

If *x* is a variant and *x* belongs to the variation point *y* and *x* is selected, that means *y* is selected (rule 7), and the variation point *y* requires a variation point *z*, that means *z* is selected also (rule 5), and the variant *n* belongs to the variation point *z* and the variant *n* is common that means the variant *n* is selected (rule 10).

ii. $\forall\ x,n{:}\ type(x,variant) \land type(n,variant) \land select(x) \land requires\_v\_v(x,n) \Longrightarrow select(n).$

If the variant *x* is selected and it requires the variant *n*, that means the variant *n* is selected, (rule 1). The selection of variant *n* propagated from the selection of variant *x*.

iii. $\forall x,z,n{:}\,type(x,variant) \land select(x) \land type(z,variationpoint) \land requires\_v\_vp(x,z) \land type(n,variant) \land variants(z,n) \land common(n,yes) \Longrightarrow select(n).$

If the variant *x* is selected and it requires the variation point z that means the variation point *z* is selected (rule 3), and the variant *n* is common and is belongs to the variation point *z* that means the variant *n* is selected (rule 10). The selection of variant *n* propagated from the selection of variant *x*.

Example 1
Suppose the user entered this choice *select (register)*, the system answered *yes* (acceptance of user selection) user announced by selection of the variant *search_name*, as propagated from selection of the variant *register.*

Table 6: example 1

```
?select (view_type.register).
yes
You selected also…. search_name
```

This example illustrates case 1. *view_type* variation point requires *item_serach* variation point and *search_name* is mandatory variant belongs to the variation point *item_search* . The direct selection of variant *register* makes *view_type* variation point selected (rule 7), and the selection of *view_type* variation point makes the *item search* variation point selected (rule 5), then the common variant *search_name(* belongs to *item_search* variation point) should be selected (rule 10). The main result of this example is the additions of two new facts select *(register)* and *select (search_name)* to knowledge base.

3.4.2. Explanation. This operation takes a feature model as an input and returns an explanation in the case when the feature model fails. Definition of error source (the constraint dependency rule(s) that causing the error) is the main aim of explanation in feature models.

Definition 2

Selection of variant *n*, *select (n)*, fails due to selection of variant *x*, *select(x)*, in three cases:

i. $\forall x,y,n{:}type(x,variant)\wedge select(x)\wedge type(y,variationpoint)\wedge variants(y,x)\wedge type(n,variant)\wedge excludes\_v\_vp(n,y)\Longrightarrow notselect(n).$

If the variant *x* is selected, and it belongs to the variation point *y*, which means *y* is selected (rule 7), and the variant *n* excludes the variation point *y*, which means *n* should not be selected (rule 4 is applied also if the variation point is selected).

ii. $\forall x,y,z,n{:}type(x,variant)\wedge select(x)\wedge type(y,variationpoint)\wedge variants(y,x)\wedge variants(z,n)\quad excludes\_vp\_vp(y,z)\Longrightarrow notselect(n).$

If the variant *x* is selected and *x* belongs to the variation point *y*, that means *y* is selected (rule 7), and the variation point *y* excludes the variation point *z*, that means *z* should not be selected (rule 6), and the variant *n* belongs to variation point *z*, that means *n* should not be selected same wise (rule 9).

iii. $\forall x,n\ \ type(x,variant)\ \wedge\ select(x)\wedge type(n,variant)\ \wedge\ excludes\_v\_v(x,n)\Longrightarrow notselect(n).$

If the variant *x* is selected, and it excludes the variant *n*, which means *n* should not be selected(rule 2).

In addition to defining the source of error, these rules can be used to prevent the errors. The predicate *notselect(n)* validate users by prevent selection.

Table 7: example 2

? select (personal_discount).
Reject
You have to deselect public_view first

Example 2

Suppose user selected *public_view* before and entered new selection, and asks to select *personal_discount*, the system rejects his choice and directed him to deselect *public_view* first. Table 7 describes example 2, this example represents case 3. The example illustrates how the model guides users to solve the rejection reason.

In addition to that the proposed method can be used to prevent rejection reasons; example 3 explains this.

Table 8: example 3

? select (Http).
Yes
notselect (credit_card_types) added to knowledge base.

Example 3

User asks to select the variant *https*, system accept his choice and add *notselect(credit_card_types)* to the knowledge base to validate future selections. Table 8 describes example 3.

Selection of the variant *Http* from *security_payment* variation point leads to the selection of *security_payment* (rule 7), and *security_payment* excludes *credit_card_types* variation point, which means *credit_card_types* should not be selected (rule 6).The predicate *notselect(credit_card_types)* prevents the selection of its variants according to rule 9.

3.4.3. Optimization. This operation returns the output, product, according to a given function or predefined criteria.

One of the advantages of our proposed method is that it can handle the extra-functional features proposed in [14], we can use extra-functional feature to optimize the search or to make filtering.

Example 4

Suppose we defined price as extra-functional feature to *security_payment* variation point in figure 1, as a result we have new three facts *price(http,100)*, *price(ssl,200)*, and *price(set,350)*. We want to ask about the feature with price greater than 100 and less than 250(price(X, Y), Y > 100, Y< 250), the system triggers the variant *ssl* with price 200. Table 9 describes example 4.

Table 9: Example 4

? price(X, Y), Y > 100, Y< 250.
X = ssl
Y = 200

3.4.4. Dead Feature Detection. A dead feature is a feature that never appears in any legal product of a feature model, defined [26] as a frequent case of error in feature model.

Definition 3

A variant *x* can be a dead feature in 3 cases:

i. $\forall x,y,z,n{:}type(x,variant)\wedge type(y,variationpoint)\wedge variants(y,x)\wedge type(z,variant)\wedge type(n,variationpoint)\wedge variants(n,z)\wedge common(n,yes)\wedge common(z,yes)\wedge excludes\_v\_vp(z,y)\Longrightarrow dead\_feature(x).$

If variant *x* belongs to the variation point *y*, and the common variant *z* belongs to the common variation point

n, which means the variant z included in any product (rules 11 and 10), the variant z excludes the variation point y that means none of its variants should be selected at all(rule 4).

ii. $\forall x,y,z{:}type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge common(z,yes) \wedge excludes\_vp\_vp(z,y) \Rightarrow dead\_feature(x).$

If the variant x belongs to the variation point y and the common variation point z excludes the variation point y that means y should be excluded (rule 6). Exclusion of the variation point y prevents selection of its variants (rule 9).

iii. $\forall x,y,n{:}type(n,variant) \wedge type(y,variationpoint) \wedge variants(y,n) \wedge common(y,yes) \wedge common(n,yes) \wedge type(x,variant) \wedge excludes\_v\_v(n,x) \Rightarrow dead\_feature(x).$

If the common variant n belongs to the common variation point y that means y should be selected in any product (rule 11), when y is selected n should be selected (rule 10), which means n should be selected in any product. The variant n excludes the variant x that means x should not be selected in any product (rule 2).

Example 5

In figure1 the variant *search_name* is a common feature from a common variation point *item_search* that means it is included in any product(rules 11 and 10), and it excludes the variation point *special_search* which means none of its variants should be selected in any product (rule 9). This means that all variants belonging to the variation point *special_ search* are dead features. Table 10 describes example 5, user inquired about dead features and system triggered the variants *similar* and *description* as dead features, this example represents case 1.

Table 10: example 5

? dead_feature(X).
X = similar
X = Description

Trinidad proposed in [20] a method to detect dead features based on finding all products and search for not used features. Our proposed method to detect dead features is better because it searches only in the above three cases.

3.4.5. Inconsistency. Inconsistency in feature model is a relationship between features that cannot be true at the same time. e.g. (A require B) and (B exclude A)

Definition 4

The inconsistency (error) can be detected in five cases:

i. $\forall x,y{:}type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge excludes\_v\_v(y,x) \Rightarrow error.$

If the variant x requires the variant y that means selection of x leads to selection of y (rule 1). And the variant y excludes the variant x that means if y selected, x should not be selected (rule 2), this is an error.

ii. $\forall x,y{:}type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires\_vp\_vp(x,y) \wedge excludes\_vp\_vp(y,x) \Rightarrow error.$

If the variation point x requires the variation point y that means selection of x leads to selection of y (rule 5), and the variation point y excludes the variation point x means if y, selected x should not be selected (rule 6), this is an error.

iii. $\forall x,y,n,z{:}type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(n,variant) \wedge type(z,variationpoint) \wedge variants(z,n) \wedge requires\_v\_v(x,n) \wedge excludes\_vp\_vp(y,z) \Rightarrow error.$

If the variant x belongs to the variation point y, and the variant n belongs to the variation point z, and x requires n that means if x selected n should be selected (rule1). Selection of the variant x means selection of the variation point y, and selection of variant n means selection of variation point z (rule 7). The variation point y excludes the variation point z that means if one of the variants belongs to y is selected none belongs to z should be selected (rules 6, 7, and 9), this is an error.

iv. $\forall x,y,z{:}type(x,variant) \wedge common(x,yes) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge excludes\_v\_vp(x,z) \wedge requires\_vp\_vp(y,z) \Rightarrow error.$

If the common variant x belongs to the variation point y, and x excludes the variation point z that means if x selected no variant belongs to z should be selected (rules 4, and rule 9), and the variation point y requires the variation point z that means if y is selected z should also be selected(rule 5). Selection of the variation point y means selection of the common variant x (rule 10) but x excludes z, this is an error.

v. $\forall x,y,z{:}type(x,variant) \wedge common(x,yes) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge requires\_v\_vp(x,z) \wedge excludes\_vp\_vp(y,z) \Rightarrow error.$

If the common variant x belongs to the variation point y, selection of x means selection of y (rule 7), and x requires the variation point z that means selection of x leads to selection of z (rule 3); but the variation point y excludes the variation point z which means if y is selected z should not be selected (rule 6), this is an error.

Feature models can contain some other complicated forms of inconsistencies like (A requires B) and (B requires C) and (C excludes A). To avoid this complication the following rules were defined:

i. $\forall x,y,z{:}type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge type(z,variant) \wedge requires\_v\_v(y,z) \Rightarrow requires\_v\_v(x,z).$

If the variant x requires the variant y, and the variant y requires the variant z that means the variant x requires the variant z.

ii. $\forall x,y,z{:}type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires\_vp\_vp(x,y) \wedge type(z,variationpoint) \wedge requires\_vp\_vp(y,z) \Rightarrow requires\_vp\_vp(x,z).$

If the variationpoint x requires the variation point y, and the variation point y requires the variation point z that means x requires z.

iii. $\forall x,y,z{:}type(x,variant) \wedge type(y,variationpoint) \wedge requires\_v\_vp(x,y) \wedge type(z,variationpoint) \wedge requires\_vp\_vp(y,z)$

$\Rightarrow$ *requires_v_vp(x,z).*

If the variant *x* requires the variation point *y*, and *y* requires the variation point *z* that means the variant *x* requires the variation point *z*.

iv. $\forall$*x,y,z:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*requires_v_v(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*requires_v_vp(y,z)*
$\Rightarrow$ *requires_v_vp(x,z).*

If the variant *x* requires the variant *y* and *y* requires the variation point *z* that means the variant *x* requires the variation point *z*.

v. $\forall$*x,y,z:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*excludes_v_vp(x,y)*$\wedge$*type(z,variant)*$\wedge$*excludes_v_v(y,z)* $\Rightarrow$ *excludes_v_v(x,z).*

If the variant *x* excludes the variant *y* and the variant *y* excludes the variant *z* that means the variant *x* excludes the variant z.

vi. $\forall$*x,y,z:type(x,variationpoint)*$\wedge$*type(y,variationpoint)*$\wedge$*excludes_vp_vp(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_vp_vp(y,z)* $\Rightarrow$ *excludes_vp_vp(x,z).*

If the variationpoint *x* excludes the variationpoint *y*, and the variation point *y* excludes the variation point *z* that means *x* excludes *z*.

vii. $\forall$*x,y,z:type(x,variant)*$\wedge$*type(y,variationpoint)*$\wedge$*excludes_v_vp(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_vp_vp(y,z)*
$\Rightarrow$ *excludes_v_vp(x,z).*

If the variant *x* excludes the variationpoint *y*, and the variation point *y* excludes the variation point *z* that means *x* excludes *z*.

viii. $\forall$*x,y,z:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*excludes_v_v(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_v_vp(y,z)*
$\Rightarrow$ *excludes_v_vp(x,z).*

If the variant *x* excludes the variant *y*, and *y* excludes the variation point *z* that means the variant *x* excludes the variation point *z*. Using backtracking mechanism the above rules can solve more complex shapes of inconsistency such as ((A requires B) and (B requires C) and (C requires D) and (D requires F) and (F excludes A)).

Our proposed method to auto detected error in feature model is better than that suggested by Trinidad (P. Trinidad 2008) because it defines two types of error in feature models.

3.4.6. Cardinality. Cardinality of variation point represents the minimum and maximum number of variant selection.

Definition 5

We define cardinality using two predicates:

i. *min(name,num)* represent the minimum cardinality; *name* represents variation point name, and *num* is an integer represents the minimum number allowed to be selected from variation point described in *name*.
ii. *max(name,num)*represent the maximum cardinality; *name* represents variation point name, and *num* is an integer represents the maximum number allowed to be selected from variation point described in *name*. Rule 12, and 13 validate cardinality.

4. Discussions and Comparison with Previous Work

In addition to automated consistency check among constraints during modeling, our proposed method can be considered as a validation and analysis method within modeling. Our proposed method can substantially define and provide an automated mechanism to the following operations:

- Definition of propagation states: identify when the propagation can happen.
- Provide explanation: We identified when the feature model fails to answer users, and we identified all features' relations those lead to feature model failure, i.e. source of error. In addition to that we illustrated how the method can guide users in correction process, in comparison with literature our method is a unique method which can provide correction process; besides all these we defined a validation predicate - *unselect(x)-* this predicate can prevent users from errors. We illustrated how the feature model can be optimized using predicates; the method can deal with the extra features.
- All cases of the dead features were defined using rules: To detect dead features search only for the corresponding defined cases.
- All cases of inconsistency were represented in feature mode using rules: provide a method to detect inconsistency in a feature model is a novel. In addition to that we used rules to define complex types of inconsistency and illustrated that how our method can prevent and solve these types of inconsistencies, which also considered as a core contribution.

5. Conclusion and future work

By modeling variability using knowledge base rules we can get both formalized variability specifications, and support selection process within variability more precisely. Firstly we proposed a method of modeling variability in feature models based product line. We started by used notations of orthogonal variability model to define variability in basic feature model, and then developed knowledge base. Our proposed method of modeling variability aims to improve the effort of producing fully automated product derivation and to deal with the complexity of variability in product line approach (intractability). The proposed knowledge base can be used to configure new product from SPL, analyzing SPL and produce error free feature model. Russell defined five steps to build knowledge [27], as we defined the five steps therefore our work can be considered as knowledge base of variability in domain engineering process.

We are planning to extend our work using constraint handling rules (CHR) to calculate and obtain all products, calculate variability (the ratio between all product and all variants in the model) , and calculate commonality. Also we are planning to develop software tool to support our method. Finally validate our method by applying it to real life case studies from industry.

6. References

[1] Meyer, M. H., and Luis Lopez, *"*Technology Strategy in a Software Products Company*"*. *Product Innovation Management,* Blackwell Publishing, vol *12*, 1995, 294-306.

[2] K. Kang, S. C., J. Hess, W. Novak, and S. Peterson, *"*Feature–Oriented Domain Analysis (FODA) *Feasibility Study*" (Technical Report No. CMU/SEI-90-TR-21) Software Engineering Institute, Carnegie Mellon University,1990.

[3] Krzysztof Czarnecki, U. Eisenecker *, Generative Programming: Methds, Tools, and Applications*, Addison-Wesley, Boston MA, 2000.

[4] Timo Asikainen , T. Männistö, T. Soininen , *"Representing Feature Models of Software Product Families Using a Configuration Ontology"*, Paper presented at the General European conference on artificial intelligence (ECAI)Workshop on Configuration , 2004.

[5] Krzysztof Czarnecki, S. Helsen, U. Eisenecker, *"Staged configuration using feature models",* Paper presented at the Third International Conference of Software Product Lines, SPLC 2004, Boston MA, USA. 2004.

[6] White D. A., *"The Knowledge Base Software Assistant: A program Summary",* Paper presented at the Knowledge-Based Software Engineering Conference, New York USA,1991

[7] kacem Zeroual , P.N. Robillard, *"KBMS: A Knowledge Based System for Modeling Software System Specification"*, IEEE Transactions on Knowledge and Data Engineering*, 4*(3),1992 , 238 – 252.

[8] Michael Schlick, A. Hein, *"Knowledge engineering in software product lines",* Paper presented at the 14th European conference on Artificial Intelligent, Workshop on Knowledge-Based Systems for Model-Based Engineering, 2000.

[9] S. Ratchev , E. Urwin, D. Muller , K.S. Pawar, and I. Moulek,*"Knowledge Based Requirement Engineering for one-of-a-kind Complex Systems*", Knowledge base systems *,*Elsevier*, 16*(1), 2003,1-5.

[10] Lother Hotez , T. Krebs, *"Supporting the Product Derivation Process with a Knowledge Base Approach",* Paper presented at the 25th International Conference on Software Engineering (ICSE2003), 2003.

[11] Lother Hotez , T. Krebs, *"A knowledge based product derivation process and some idea how to integrate product development",* Paper presented at the Software Variability Management Workshop, Groningen The Netherlands, 2003.

[12] M. Mannion , *"Using First-Order Logic for Product Line Model Validation",* Paper presented at the Second Software Product Line Conference (SPLC2), San Diego, CA. , 2002.

[13] Wei Zhang, H. Z., and Hong Mei, *"A Propositional Logic-Based Method for Verification of Feature Models",* Paper presented at the 6th International Conference on Formal Engineering Methods (ICFEM),2004.

[14] David Benavides, P. Trinidad, and A.Ruiz-Cortes, *"Automated Reasoning on Feature Models*", Advanced Information Systems Engineering (Vol. 3520/2005,), Springer, Berlin Heidelberg, 2005, pp. 491-503.

[15] Don Batory,*"Feature Models, Grammars, and Propositional Formulas",* Paper presented at the 9th International Software Product Lines Conference (SPLC05), Rennes, France, 2005.

[16] Mikolas Janota, Joseph Kiniry, *"Reasoning about Feature Models in Higher-Order Logic",* Paper presented at the 11th International Software Product Line Conference (SPLC07), 2007.

[17] Krzysztof Czarnecki, Michal Antkiewicz, *"Mapping features to models: A template approach based on superimposed variants",* Paper presented at the 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, 2005.

[18] Krzysztof Czarnecki, Krzysztof Pietroszek, *"Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints"*, Paper presented at the 5th international conference on Generative programming and component engineering (GPCE'06), 2006.

[19] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura, *"A survey on the automated analyses of feature models", Jornadas de Ingeniería del Software y Bases de Datos(JISBD 2006)*, 2006.

[20] Pablo Trinidad, David Benavides, and Antonio Ruiz-Cort´es, *"Isolated features detection in feature models"*, Paper presented at the Advanced Information Systems Engineering (CAiSE), Luxembour, 2006.

[21] Pablo Trinidad , D. Benavides, A. Dura´n, A. Ruiz-Cortes, and M. Toro, *"Automated error analysis for the agilization of feature modeling"*, systems and software, doi:10.1016/j.jss.2007.10.030, 2008.

[22] Don Batory, David Benavides, Antonio Ruiz-Cortés, *"Automated Analyses of Feature Models: Challenges Ahead"*, Special Issue on Software Product Lines ,Communications of the ACM, December 2006.

[23] Timo Käkölä, Juan C. Dueñas , *Software Product Lines Research Issues in Engineering and Management*, Springer, Verlag Heidelberg Germany, 2006.

[24] Klaus Pohl, Günter Böckle, and Frank van der Linden, *Software Product Line Engineering Foundations Principles and Techniques*, Springer, Verlag Heidelberg Germany, 2005.

[25] Jan Wielemaker ,SWI-Prolog (Version 5.6.36) free software, Amsterdam, University of Amsterdam,2007.

[26] Krzysztof Czarnecki, Chang Hwan Peter Kim, *"Cardinality-based Feature Modeling and Constraints: A Progress Report"*, Paper presented at the International Workshop on Software Factories at (OOPSLA'05), San Diego California, 2005.

[27] Stuart J. Russell , Peter Norvig , *Artificial Intelligence A Modern Approach* , Prentice Hall, New Jersey 07632, 1995.