J. Software Engineering & Applications, 2010, 3, 614-627
doi:10.4236/jsea.2010.36072 Published Online June 2010 (http://www.SciRP.org/journal/jsea)

An Interactive Method for Validating Stage Configuration

Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, Chin Kuan Ho

Center of Artificial Intelligent and Intelligent Computing, Multimedia University, Cyberjaya, Malaysia.
Email: {abdelrahman.osman.06, somnuk.amnuaisuk, ckho}@mmu.edu.my

Received March 21st, 2010; revised April 6th, 2010; accepted April 8th, 2010.

ABSTRACT

Software product Line (*SPL*) *is an emerging methodology for developing software products. Stage-configuration is one the important processes applying to the SPL. In stage-configuration, different groups and different people make configuration choices in different stages. Therefore, a successful software product is highly dependent on the validity of stage-configuration process. In this paper, a rule-based method is proposed for validating stage-configuration in SPL. A logical representation of variability using First Order Logic* (*FOL*) *is provided. Five operations*: *validation rules, explanation and corrective explanation, propagation and delete-cascade, filtering and cardinality test are studied as proposed operations for validating stage-configuration. The relevant contributions of this paper are*: *implementing automated consistency checking among constraints during stage-configuration process based on three levels* (*Variant-to-variant, variant-to-variation point, and variation point-to-variation point*), *define interactive explanation and corrective explanation, define a filtering operation to guide the user within stage-configuration, and define* (*explicitly*) *delete-cascade validation.*

Keywords: *Software Product Line, Variability, Stage Configuration*

1. Introduction

Software Product Line (SPL) has proved to be an effective strategy to benefit from software reuse [1], allowing many organizations to reduce development costs and duration, meanwhile increase product quality [2]. It is an evolution from software reuse and Commercial Off-The-Shelf (COTS) methodologies.

Feature Model (FM) [3] appeals to many SPL developers as essential abstractions that both customers and developers understand. Customers and engineers usually speak of product characteristics in terms of those features the product has or delivers. Therefore, it is natural and intuitive to express any commonality or variability in terms of features [4]. Orthogonal Variability Model (OVM) is another successful approach proposed to document variability in SPL [5].

The principal objective for SPL is to configure a successful software product from domain-engineering process by managing SPL artifacts (variability modeling). Recently, in [6,7], validation is discussed as important issue in SPL community. Validating SPL intends to produce error-free products including the possibility of providing explanations to the modeler so that errors can be detected and eliminated. Usually, medium SPL contains thousands of features [2]. Therefore, validating SPL represent a challenge because it's a vital process and non-feasible to done manually.

The lack of a formal semantics and reasoning support of FM has hindered the development of validation methods for FM [8]. The automated validation of FM was already identified as a critical task in [9-11]. However, there is still a lack of an automated support for FM validation.

The configuration is a task of selecting a valid and suitable set of features for a single system, and it can become very complicated task [12]. Supporting user's needs in generating a valid and suitable solution is the basic functionality of a configuration system. In SPL, selecting a solution from domain-engineering process to use it in application-engineering process is the meaning of a configuration. As a conclusion, FM represents the configuration space of a SPL. An application-engineer may specify a member of a SPL by selecting the desired features from the FM within the variability constraints defined by the model, e.g., the choice of exactly one feature from a set of alternative features. In stage configuration, different groups and different people make configu-

ration choices in different stages [13].

2. Preliminaries

2.1 Software Product Line

SPL 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 [14]. SPL has two main processes; the first process is the domain-engineering process that represents domain repository and is responsible for preparing domain artifacts including variability. The second process is the application-engineering that aims to consume specific artifact, picking through variability, with regards to the desired application specification. The useful techniques to represent variability are FM and OVM. SPL has various members. A particular product-line member is defined by a unique combination of features (if variability modeled using FM) or a unique combination of variants (if variability modeled using OVM). The set of all legal features or variants combinations defines the set of product line members.

2.2 Feature Model

FM [3] is considered as one of the well-known methods for modeling SPLs [9]. According to Czarnecki and Eisenecker [12] the two most popular definitions of FM are 1) an end user visible characteristic of a system 2) a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholders of the concept. Features in FM represent essential abstractions of the SPL (to both developers and customers). Customers and developers usually speak of product characteristics in terms of those features the product has or delivers. Therefore, it is natural and intuitive to express any commonality or variability in terms of features [4]. A FM is a description of the commonalities and differences between the individual software systems in a SPL. In more detail, a FM defines a set of valid feature combinations. The set of all legal features combinations defines the set of product line members. Each of such valid feature combinations can be served as a specification of a software product [12,15].

A feature model is a hierarchically structure of features and consists of: 1) relationships between a parent feature and its child features, and 2) dependency constraints rules between features, which are inclusion or exclusion. Czarnecki *et al.* [13] defined Cardinality-based feature modeling by integrating a number of extensions to the original FODA notation. According to [13] there are three different versions of FM: basic FM, cardinality FM, and extended FM. **Figure 1** illustrates basic FM. **Figure 1** is borrowed from [10]. In any type of FMs there are there are some common properties such as (examples are based on **Figure 1**):

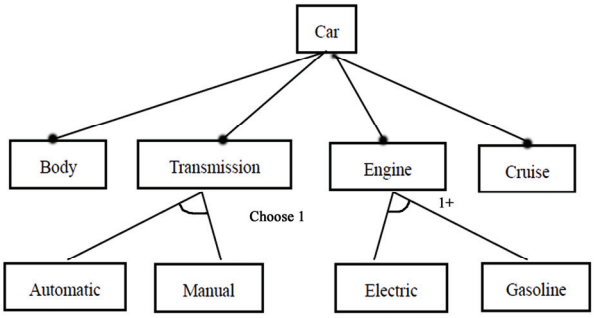

Figure 1. A car software product line represented by basic feature model

Parent Feature: This feature could be described as a decision point, in which some choices or decisions should be taken. A parent feature contains one or more of child feature(s). As example, transmission is a parent feature.

Child Feature: A feature belongs to parent feature is called a child feature, for instance, "Electric" is a child feature belongs to the parent feature "Engine".

Common Feature: In a common relationship (between parent and child features), a child feature follows his parent feature in any product. For instance, "Engine", "Transmission", and "Body" are common features (belongs to the parent feature "Car"), which means it must be including in any product related to car SPL.

Option Feature: Option feature can follow his parent feature or not. For instance, "Cruise" is an optional feature.

Selection process: In selection process, one or more features could be selected from the parent feature. In basic FM, this operation is known as alternative, where one child feature can be included in a product when the parent feature is included. In cardinality FM, the selection process organizes by two numbers represent maximum and minimum numbers allowed to be selected from parent feature.

Constraint dependency: is known also as cross-tree relation. Require and exclude represent the constraint dependency. In the following require and exclude are defined.

Exclude: feature X excludes Y, means that if X is included in a product then Y must not be included and vice versa.

Require: feature X requires Y, means that if X is included in a product then Y must be included.

2.3 Orthogonal Variability Model

Orthogonal Variability Model (OVM) [5] is one of the useful techniques to represent variability, which provides a cross-sectional view of the variability through all software development artifacts. **Figure 2** illustrates OVM for e-shopping system. **Figure 2** borrow from [5]. Variabil-

ity is described (in OVM) by three terms. These terms are:

Variation Point: a variation point is a point that could be select one or more of its variants. For instance, "Member reward" in **Figure 2** is a variation point.

Variant: a variant is a choice belonging to specific variation point. For instance, "https", "SSL", and "SET" are variants belonging to "Secure payment" variation point.

Constraint dependencies rules: These rules describe the dependency relation between variation points and variants. Require and exclude signify the constraint dependency relation in OVM. These relations are: variation point requiring or excluding another variation point, variant requiring or excluding another variant and variant requiring or excluding variation point.

3. Related Work

In this section, we survey the works that related to validation of SPL regardless the method of modeling variability used. Inside these studies, we highlight the validation operations of the configuration. At the end, we justify our contributions.

Schlich and Hein proved the needs and benefits of using the knowledge-base representation for configuration systems in [16]. A knowledge-based product derivation process [17,18] is a configuration model that includes three entities of Knowledge Base. The automatic selection provides a solution for complexity of product line variability. In contrast to the proposed method, the knowledge -based product derivation process does not provide explicit definition of variability notations and for the selection process. In addition, knowledge-based product derivation process is not focused on validation operations.

Mannion [19] was the first who connect propositional formulas to FM. Mannion's model did not concern cross-tree constraints (Require and Exclude constraints). Zhang *et al*. [20] defined a meta-model of FM using Unified Modeling Language (UML) core package and took Mannion's proposal as foundation and suggested the use of an automated tool support. Zhang's model satisfies constraint dependency checking in the basic level (feature-to-feature). By define pre-condition and post-condition for each feature explanation operation was satisfied but there is no mentioned for propagation. Batory in [21] proposed a coherent connection between FM, grammar and propositional formulas. Batory's study represented basic FM using context–free grammars plus propositional logic. This connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfible solvers to debug FM. Batory using solvers satisfied constraint dependency checking and explanation operations only. Sun and Zhang [22] proposed a formal semantics for the FM using first order logic. Sun used Alloy Analyzer (tool for analyzing models written in alloy) to automate constraint dependency checking (feature-to-

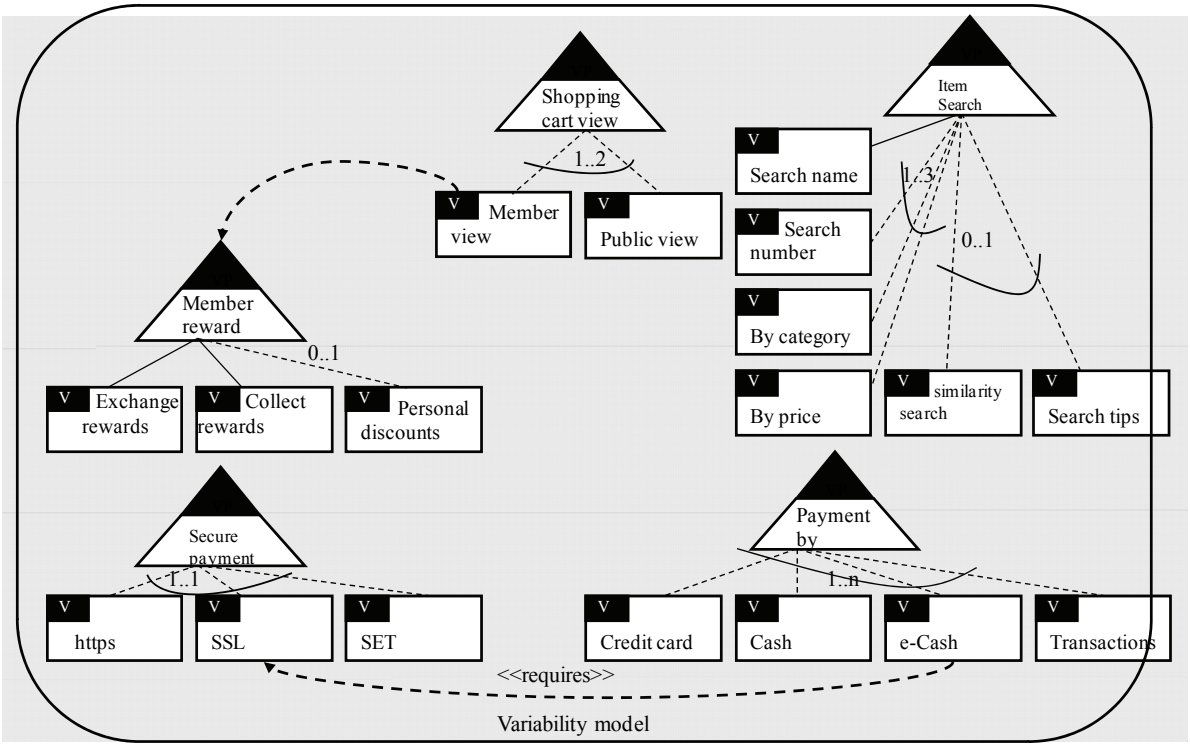

Figure 2. OVM represent variability in E-shopping system

feature level) and explanation operations in the configuration process. Asikainen *et al*. [15] satisfied constraint dependency checking and explanation by translate the model into Weigh Constraint Rule Language (WCRL). WCRL is a general-purpose knowledge representation language developed based on propositional logic.

Wang *et al*. [23] proposed Ontology Web Language (OWL) to Verify FM. Wang used OWL DL ontology to capture the interrelationships among the features in a FM. Wang supported constraint dependency checking and explanation by using FaCT++ (Fast Classification of Terminologies and RACER (Renamed ABox and Concept Expression Reasoner) reasoner tools. Falbo *et al*. [24] formalized domain-engineering process using ontology. Falbo *et al*. mapped the constraint relations in domain engineering to the synonymous primitive in set theory, and used hybrid approach based on pure first-order logic, and set theory for reasoning.

Dedeban [25] used OWL DL and rule based system to support constraint dependency checking. Dedban's method satisfy constraint dependency checking and explanation used FaCT++ reasoner. Shaofeng and Zhang [26] formalized the FM with description logic to reason constraint rules via description logic.

Transforming FM into Unified Modeling Language (UML) notations, representing, and documenting variability in SPL is proposed by different methods in literature. Usability is the main reward for using UML due to UML standardization. Clauss [27] suggested Object Constraint Language (OCL) to satisfy constraint dependency rules. Korherr and List [28] proposed a UML 2 profile to model variability. Korherr's model used OCL to satisfy the three levels constraint dependency rules. Ziadi *et al*. [29,30] used OCL in the form of meta-model level to satisfy constraint dependency rules. The works in [31-37] adopted UML (with different views) as a solution for modeling variability in SPL. These methods implemented OCL to satisfy dependency constraint rules. Czarnecki and Antkiewicz [38] proposed a general template-based approach for mapping FM. Czarnecki and Pietroszek [39] used object-constraint language (OCL) to validate constraint dependency rules.

The use of constraint programming for dealing with constraint dependency checking and explanation are suggested in [40-42] where FM is translated into a Constraint Satisfaction Problem (CSP) for automating FM analysis with a constraint solver. White *et al*. [43] proposed a method for automated diagnosis of product-line configuration errors in FM. White's method starts with transferring the rules of the FM and the current invalid configuration into a CSP. Later, the solver derives a labeling of the diagnostic CSP. Finally, the output of the CSP labeling is transform into a series of recommendations of features to select or deselect to turn the invalid configuration into a valid configuration.

Cao *et al*. [44] developed algorithm to transfer FM into data structures. This algorithm generates complete feature instances from a feature diagram under constraints. Cao used the Generic Modeling Environment (GME) to develop the algorithm. The Cao's algorithm satisfies constraint dependency checking and explanation operations. Deursern and Klint [45] proposed a feature description language to describe FM. Using their system constraint dependency checking and explanation operations are satisfied. Pohjalainen [46] described subset of regular expressions that can be used to express a FM. Pohjalainen presented a compiler for translating a FODA model to a deterministic finite state machine with support for implementing model constraints via post-augmentation of the compiled state machines. This model satisfies constraint dependency checking and explanation operation.

Cechticky *et al*. [47] proposed a feature meta-model and used an XSL-based mechanism to express complex composition rules for the features. Cechticky *et al*. described a compiler that can translate the constraint model expressed as a feature diagram into an XSL program and checks compliance with the constraints at application model level. Jarzabek and Zhang [48] described a variant configuration language that allows to instrument domain models with variation points and record variant dependencies. Implementation based on XML and XMI technologies was also described. Jarzabek's method satisfies constraint dependency checking and explanation.

Janota and Kiniry [46] formalized a FM using HOL. As the best of our knowledge this is the only one work used HOL for reasoning FM. This formalization satisfied constraint dependency checking and explanation operations. Lengyel *et al*. [49] proposed an algorithm (to handle constraints in FM) based on graph rewriting based topological model transformation. Implementation is done based on OCL semantics. Constraint dependency checking is satisfied based on feature-to-feature level.

Comparing with the literature, our propose method (to satisfy constraint dependency rules, explanation, corrective explanation, propagation and delete cascade, and filtering) is characterized by an interactive mechanism which is guide users systematically. In addition to constraint dependency rules (require and exclude), our validation rules are validate the commonality (is the feature is common or not), and the cardinality.

4. Modeling Variability Using First Order Logic

The most popular models for SPL variability modeling are FM and Orthogonal Variability Model (OVM). Therefore, the successful validation notations are those that can validate both FM and OVM. Roos-Frantz [50] illustrated the differences between FM and OVM. To overcome the differences we merge the FM and OVM in the proposed method (benefiting from their advantages),

e.g. a variation point is defined explicitly (mandatory or optional) and hierarchical structure is supported. This modeling is a prerequisite process for using the validation operations. OVM and FM can easily become very complex for validating a medium size system, *i.e.*, several thousands of variation points and variants are needed.

4.1 Upper Layer Representation (FM-OVM)

Figure 3 represents the upper layer of our proposed method. Optional and mandatory constraints are defined in **Figure 3** by original FM notations [3] and constraint dependency rules are described using OVM notations. The FM-OVM has three layers. Member reward is a variation point having a personal discount as a variant. A personal discount is also a variation point for three variants. A personal discount is a variant and variation point at the same time.

4.2 Lower Layer of the Proposed Method

Variation points, variants, and dependency constraint rules are described using predicates as a lower layer of the proposed method: (examples are based on **Figure 3**. Terms starting by capital letters represent variables and terms starting by lower letters represent constants):

4.2.1. Variation Point

The following five predicates are used to describe each variation point:

type: Define the type of feature; variation point, e.g.: type (view_type,variationpoint),

variants: Identifies the variant of specific variation point, e.g.: variant (view_type, not registered).

max: Identifies the maximum number allowed to be selected of specific variation point, e.g. max (payment_by, 4).

min: Identifies the minimum number allowed to be selected of specific variation poin, 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 variation point, e.g. common (item-search, yes). If the variation point is not common, the second slot in the predicate will become No, as example, common (member reward, no).

4.2.2. Variant

The Following two predicates (from the above five predicates) are used to describe variants:

type: Define the type of feature (variant), e.g.: type (register, variant).

common: Describe the commonality of variant, e.g. common(search name, yes). If the variant is not common, the second slot in the predicate will become No -as example-common (by price, no).

4.2.3. Constraint Dependency Rules

The following six predicates are used to describe constraint rules:

requires_v_v: a variant requires another variant, e.g. requires_v_v (ecash, ssl).

excludes_v_v: a variant excludes another variant, e.g. excludes_v_v(by price, member view).

requires_v_vp: a variant requires variation point, e.g. requires_v_vp (member_view, member_reword).

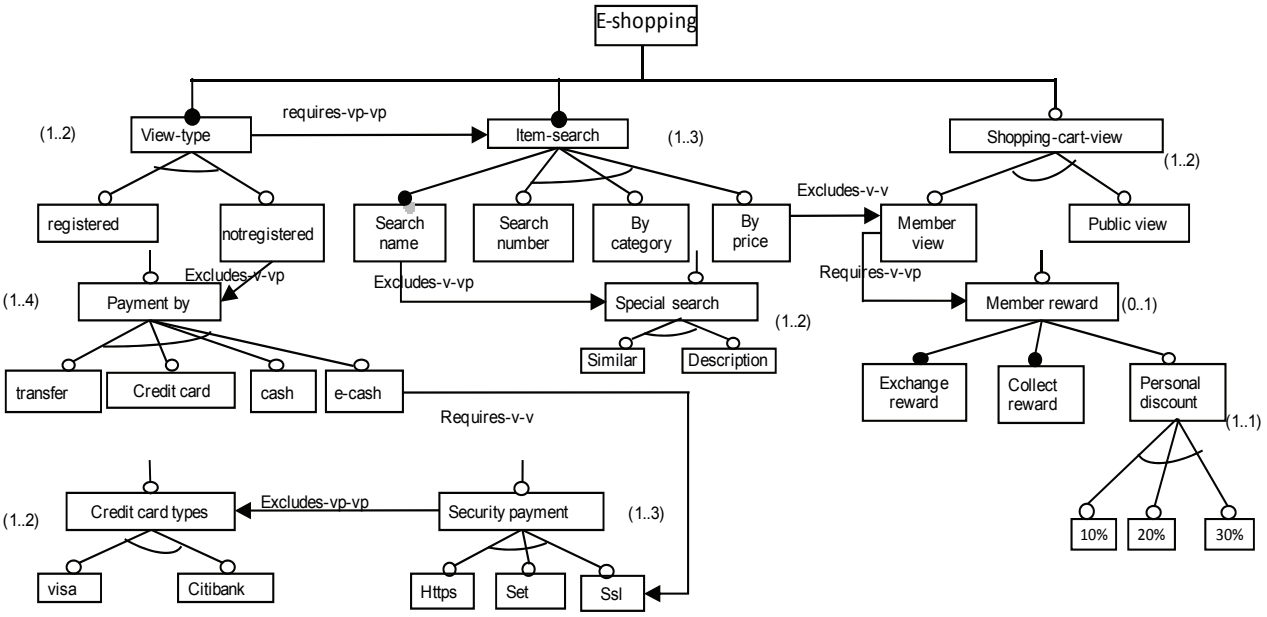

Figure 3. Representation of e-shopping system using the upper layer (FM-OVM)

excludes_v_vp: a variant excludes variation point, e.g. excludes_v_vp (notregistered, payment_by).

requires_vp_vp: a variation point requires another variation point, e.g. requires_vp_vp (item_search, view_ype).

excludes_vp_vp: a variation point excludes another variation point, e.g. excludes_vp_vp (security_payment, credit_card_type).

Table 1 shows the lower layer representation of the variation point *view-type* and the variant *not registered*. The lower layer models variability in one-to-one mapping. The predicate *variants* emphasize this point.

4.3 Generalization

FM-OVM might compose of many levels (variation point can contain one or more variation points) for example form **Figure 3**: variation point *member reward* has a variation point *personal discount* and variation point *personal discount* has a variant **30%**. A mathematical representation of this case is: *member reward* (*personal discount* (30%)). The following facts illustrate the modeling of this case:

type(member_reward,variationpoint)∧type(personal_discount,variation-point)∧type(30%,variant)∧variants(meber_reward,personal_iscount)∧variants(personal_discount,30%).

The following rule (transformation rule) concludes this relation:

$\forall x, y, z$: variants(x, y)∧variants(y, z) $\Longrightarrow$ variants(x, z).

In the next section, we illustrate how the proposed method can be used for validating stage-configuration in SPL.

5. Operations for Validating Stage-Configuration in Software Product Line

5.1 Validation Rules

To validate the configuration process, the 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 are suggested. When a new variant is selected, new predicate (*select* or *notselect*) would be added to the knowledge-base and the backtracking mechanism validates the entire knowledge-base. At the end of the configuration process, *select* and not *notselect* predicates represent the product. **Table 3** shows the abstract representation of the main rules.

Table 1. Snapshot of the lower layer 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_p_p(view-ype, earch_tem). type(not registered, variant). common(not registered, no). excludes_v_vp(not registered, payment by).

Table 2. Predicates represent constraint dependency rules in the proposed method

requires_v_v: Variant requires variant equire_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 *V*1 excludes the selection of the related variant *V*2 independent of the variation points the variants are associated with. e.g. excludes_v_v(By price, member view).
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 *V*1 requires the consideration of a variation point *VP*2. e.g. requires_v_vp (member_view, member_reward).
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 *V*1 excludes the consideration of a variation point *VP*2. 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= var- iation point	The consideration of a variation point excludes the consideration of another variation point. e.g. excludes_vp_vp (security_ payment, credit_card_type).

Rule 1:

For all variant *x* and variant *y*; if *x* requires *y* and *x* is selected, then *y* is selected.

Rule 2:

For all variant *x* and variant *y*; if *x* excludes *y* and *x* is selected, then *y* is assigned by *notselect* predicate.

Rule 3:

For all variant *x* and variation point *y*; if *x* requires *y* and *x* is selected, then *y* is selected. This rule is applicable as well if the variation point is selected first:

$\forall$ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ require_v_vp(x, y) ∧ select(y) $\Longrightarrow$ select(x)

For all variant *x* and variation point *y*; if *x* requires *y* and *y* is selected, then *x* is selected.

Rule 4:

For all variant *x* and variation point *y*; if *x* excludes *y* and *x* is selected, then *y* assigned by *notselected* predicate.

Table 3. Abstract representation of the main rules

1	∀ x, y: type(x, variant) ∧ type(y, variant) ∧ require_v_v(x, y) ∧ select(x) ⟹ select(y)
2	∀ x, y: type(x, variant) ∧ type(y, variant) ∧ exclude_v_v(x ,y) ∧ select(x) ⟹ notselect(y)
3	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ require_v_vp(x, y) ∧ select(x) ⟹ select(y)
4	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ exclude_v_vp(x, y) ∧ select(x) ⟹ notselect(y)
5	∀ x, y: type(x, variationpoint) ∧ type(y, variationpoint) ∧ require_vp_vp(x, y) ∧ select(x) ⟹ select(y)
6	∀ x, y: type(x, variationpoint) ∧ type(y, variationpoint) ∧ exclude_vp_vp(x, y) ∧ select(x) ⟹ notselect(y)
7	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ select(x) ∧ variants(y, x) ⟹ select(y)
8	∃x ∀y:type(x, variant) ∧ type(y, variationpoint) ∧ select(y) ∧ variants(y, x) ⟹ select(x)
9	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ notselect(y) ∧variants(y, x) ⟹ notselect(x)
10	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ common(x,yes) ∧ variants(y, x) ∧ select(y) ⟹ select(x)
11	∀ y: type(y, variationpoint) ∧ common(y,yes) ⟹ select(y)
12	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧variants(y, x) ∧select(x) ⟹ sum(y,(x)) ≤ max(y,z)
13	∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧variants(y, x) ∧select(x) ⟹ sum(y,(x)) ≥ min(y,z)

This rule is applicable as well, if the variation point is selected first:

∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ exclude_v_vp(x, y) ∧ select(y) ⟹ notselect(x)

∀ x, y: type(x, variant) ∧ type(y, variationpoint) ∧ exclude_v_vp(x, y) ∧ select(y) ⟹ notselect(x)

For all variant *x* and variation point *y*; if *x* excludes *y* and y selected, then *x* is assigned by *notselect* predicate.

Rule 5:

For all variation point *x* and variation point *y*, if *x* requires *y* and *x* selected, then *y* is selected.

Rule 6:

For all variation point *x* and variation point *y*, if *x* excludes *y* and *x is* selected, then *y* is assigned by *notselect* predicate.

Rule 7:

For all variant *x* and variation point *y*, where *x* belongs to *y* and *x* is selected, that means *y* is 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* then *x* is selected.

This rule states that if a variation point was selected, then there is variant(s) belong to this variation point must 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* is assigned by *notselect* predicate. This rule states that if a variation point was excluded, then none of its variants must select.

Rule 10:

For all variant *x* and variation point *y*; where *x* is a common, *x* belongs to *y* and *y* is selected, then *x* is selected. This rule states that if a variant is common and its variation point selected then it is selected.

Rule 11:

For all variation point *y*; if *y* is common, then *y* is selected. This rule states that if a variation point is common then it is 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* must 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* must not be greater than the minimum number allowed to be se-lected from *y*.

The *notselect* predicate prevents feature to be selected, e.g. rule 9.

Rules 12 and 13 validate the number of variants' selection considering the maximum and minimum conditions in variation point definition (cardinality definition). The predicate *sum*(*y,* (*x*)) returns the summation number of selected variants belongs to variation point *y*. From these rules, the full common variant (variant included in any product) can be defined as:

∀x,y:type(x,variant)∧type(y,variationpoint)∧variants(y,x) ∧common(y,yes)∧common(x,yes)⟹ 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 is selected (rule 10).

The proposed rules (to validate the configuration) are based on two layers. The upper layer is a variation point layer where each rule applied to variation point reflect into all its variants, e.g. if variant excludes variation point that means this variant excludes all the variants that belong to this variation point, rule (9). The lower layer is a variant layer where each rule is applied for specific variant.

5.2 Explanation and Corrective Explanation

This operation is defined (in this paper) for highlighting the sources of errors within configuration process.
The general pattern that represents failure to select one feature in the configuration process is:

Feature A excludes Feature B and Feature A is selected then Feature B failed to select.

In the proposed method, there are two possibilities for the feature: variation point or variant and three possibilities for the exclusion: variant excludes variant, variant excludes variation point or variation point excludes variation point. Definition 1 describes these possibilities in the form of rules.

Definition 1:

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

∀x,y,n:type(x,variant)∧select(x)∧type(y,variationpoint) ∧variants(y,x)∧type(n,variant)∧excludes_v_vp(n,y)⟹ notselect(n).

If the variant *x* is selected, and it belongs to the variation point *y*, this means *y* is selected (rule 7), and the variant *n* excludes the variation point *y*, this means *n* assigned by *notselect* predicate (rule 4 is applied also if the variation point is selected).

∀x,y,z,n:type(x,variant)∧select(x)∧type(y,variationpoint)∧variants(y,x)∧variants(z,n) excludes_vp_vp(y,z)⟹ notselect(n).

If the variant *x* is selected and *x* belongs to the variation point *y*, that means *y* is selected (rule 7), and if the variation point *y* excludes the variation point *z*, this means *z* is assigned by *notselect* predicate (rule 6), and the variant *n* belongs to variation point *z*, this means *n* is assigned by *notselect* predicate (rule 9).

∀x,n: type(x,variant)∧select(x)∧type(n,variant)∧excludes_v_v(x,n)⟹notselect(n).

If the variant *x* is selected, and *x* excludes the variant *n*, which means *n* is assigned by *notselect* predicate (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 preventing selection.

Example 1

Suppose the user selects *memebr_view* before entering a new selection and request to select *by price*, the system rejects the choice and directs the user to deselect member_*view* first. **Table 4** describes example 1. This example represents rule (3). The example illustrates how the proposed method guides users to solve the rejection reason. In addition to that, it can be used to prevent rejection reasons; example 2 explains this.

Example 2

The user asks to select the variant *https*, the system accepts the choice and adds *notselect*(*credit_card_types*) to the knowledge-base to validate future selections. **Table 5** describes example 2. Selection of the variant *Https* from *security_payment* variation point leads to the selection of *security_payment* (rule 7), and *secu rity_payment* excludes *credit_card_types* variation point, which means *credit_card_types* must not be selected (rule 6). The predicate *notselect*(*credit_card_types*) prevents the selection of its variants according to rule 9.

Table 4. Example 1

? select (by price). You have to deselect memebr_view

Table 5. Example 2

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

The proposed method guides user step by step (in each choice), if the user's choice is invalid immediately reject it and suggest the correct actions (corrective explanation), see example 1. Moreover, *notselect* predicate can be assigned to some features according to user's selection, see example 2. The *notselect* predicate prevents user from future errors.

5.3 Filtering

Filtering operation guides the user to develop his product based on predefined conditions.

Example 3

Suppose *price* was defined as an extra-functional feature to *security_payment* variation point in **Figure 3**. As a result three new predicates (*price*(*https,*100), *price* (*ssl,* 200), and *price*(*set*,350)) were added. 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 6** describes example 3.

5.4 Propagation and Delete-Cascade

In this operation, some features are automatically selected (or deselected).

The general pattern that represents selection of feature based on selection of another feature is:
Feature A requires feature B and feature A is selected then feature B is auto-selected.

In the proposed method, there are two possibilities for the feature: variation point or variant and three possibilities for the requiring: variant requires variant, variant requires variation point or variation point requires variation point. Definition 2 describes these possibilities in the form of rules.

Definition 2:

Selection of variant *n*, *select*(*n*), is propagated fromselection of variant *x*, *select*(*x*), in three cases:

i.∀x,y,z,n:type(x,variant)∧variants(y,x)∧select(x)∧requires_vp_

Table 6. Example 3

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

vp(y,z)∧type(n,variant)∧variants(z,n)∧common(n,yes)⟹ autoselect(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.∀x,n:type(x,variant)∧type(n,variant)∧select(x)∧requires_v_v(x,n) ⟹ autoselect(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.∀x,z,n:type(x,variant)∧select(x)∧type(z,variationpoint)∧requires_v_vp(x,z)∧type(n,variant)∧variants(z,n)∧common(n,yes) ⟹ autoselect(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 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 4

Suppose the user enters 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*. This example illustrates case 1: *view_type* variation point requires *item_search* variation point and *search_name* is common variant belongs to the variation point *item_search*. The direct selection of variant *register* makes v*iew_ 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) is selected (rule 10). The main result of this example is the additions of two new facts *select (register)* and *autoselect (search_name)* to the knowledge base. **Table 7** illustrates example one.

Delete-Cascade Operation:

This operation validates configuration process in the execution time. The following scenario describes the problem: If the variant *x* is selected in time *N* and *x* requires two variants *y* and *k*, then the solution (at time *N*) = {*x*, *y*, *k*}. In time (*N* + 1), the variant *m* is selected, and *m* excludes *x*, then *x* is remove from the solution. The solution at time (*N* + 1) = {*m*, *y*, *k*}. The presence of the variants *y* and *k* is not a real selection. Rule 2 (Subsection 5.1) assign *notselect* predicate (to the excluded variant) as a result from the exclusion process. The following rules added to the knowledge base to implement *delete-cascade*.

i.∀x,y:type(x,variant)∧type(y,variant)∧requires_v_v(y,x)∧autoselect(x) ∧select(y) ∧ notselect(y)⟹ notselect(x).

ii.∀x,y:type(x,variant)∧type(y,variant)∧requires_v_v(y,x)∧autoselect(x) ∧autoselect(y) ∧ notselect(y)⟹ notselect(x).

For all variants *x*, and *y*; if the variant *y* is requires *x,* y is selected or auto selected, *x* is auto selected and *y* assigned by *notselect* predicated, that means *y* is excluded within the configuration process, and *x* was selected according to selection of *y* (propagation) then the presence of *x* after exclusion of *y* is not true. The output for this operation is the assigning of the variant *x* with *notselect* predicate. This assigning permits backtracking mechanism to perform *delete-cascade* operation to verify the products.

5.5 Cardinality Test

Cardinality test operation validates the cardinality of each selected variation point. At the begging of this operation, all auto-selected variants must be assigned by *select* predicate, which constructs all selected variants in the configuration process assign by *select* predicate.
The following rule assiges *select* predicate to all auto-selected variants.

∀x: autoselect(x) ⟹ select(x).

The following rule converts all selected variants belonging to the variation point *y* to a list, *i.e.* the list contains only variants belonging to one variation point:

∀y,x:variants(y,x),select(x) ⟹ list(y,[x]).

The following rules test the maximum and minimum cardinality for each selected variation point.

∀y,x,len,m: length(list(y,[x]),len) ∧ max(y,m) ∧ len > m ⟹ error.

∀y,x,len,m: length(list(y,[x]),len) ∧ min(y,m) ∧ len < m ⟹ error.

In the above two rules, the length of a list compares against cardinality of its variation point and alert messages triggers out in case of error.

6. Implementation and Scalability Testing

In this section, technical details are present and discuss. **Table 8** shows an interactive configuration program. All programs are implemented based on prolog SWI software.

The program in **Table 8** guides user step by step to complete his selections. First, user enters his choice (the variant *X*). Later, two subroutines validate the selection. The first subroutine (interactive_validate_require) works to figure out all variants required by the selected variant *X* based on the three constraints: variant requires variant, variant requires variation point, and variation point requires variation point. The second subroutine (interacttive_ validate_exclude) works to figure out all variants excluded by the selected variant *X* based on the three constraints: variant excludes variant, variant excludes variation point, and variation point excludes variation point.

Table 9 shows a program to generate the maximum product. A maximum product defined as a product contains all variant in SPL considering the constraint dependency rules [42].

The program to generate the maximum product (**Table 9**) contains five subroutines: sel_common, sel_variant, validate_exclude, del_cascade, and make_product. In the following, each subroutine is discussed:

sel_common: this subroutine selects the common variants, *i.e.*, common variant belonging to common variation point.

sel_variant: select all the variants (that are not selected before as common variants) are the mission of this subroutine.

validate_exclude: This subroutine validates exclude constraint. In subroutine, all variants are compared against each other. The excluded variant is assigned by *notselect* predicate.

del_cascade: this subroutine implements the delete-cascade operation that is defined in Subsection 5.4 in this paper.

Table 7. Example 4

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

Table 8. An interactive configuration program

```
sel:-
read(X),
interactive_validate_exclude(X),
interactive_validate_require(X).

interactive_validate_require(X):-
type(N,variant), X \==N,
((requires_v_v(X,N), write(' you have to de select    '), write(N),nl);
( variants(M,N), requires_v_vp(X,M), common(N,yes),write(' you
have to select '), write(N),nl);
(variants(Y,X), variants(M,N), requires_vp_vp(Y,M), common
(N,yes), write (' you have to select '), write(N).

interactive_validate_exclude(X):-
type(N,variant), X \==N,
((excludes_v_v(X,N), write(' you have to de select    '), write(N),nl);
( variants(M,N), excludes_v_vp(X,M),write(' you have to deselect '),
write(N),nl) ;
(variants(Y,X), variants(M,N), excludes_vp_vp(Y,M),write(' you
have to deselect '),write(N).
```

Table 9. A program to generate the maximum product

```
max_product:-
sel_common,
sel_variant,
validate_exclude,
del_cascade,
make_product.

sel_common:-
variants(Y,X),
common(Y,yes),
common(X,yes),
write('select'),write('('), write(X), write(').').

sel_variant:-
type(X, variant),
not(select(X)),
write('select'),write('('), write(X), write(').').

validate_exclude:-
select(N), select(X), X \==N,
((excludes_v_v(X,N), write('notselect'),write('('), write(X), write (').'), nl);
(variants(M,N),excludes_v_vp(X,M),write('notselect'),write('('),
write(X), write(').'),nl) ;
(variants(Y,X), variants(M,N), excludes_vp_vp (Y,M),write ('notse-
lect'), write('('), write(X), write(').'),nl)).

del_cascade:-
select(N),
requires_v_v(M,N),
notselect(M), % that means M was selected and then deleted
write('notselect'),write('('), write(N), write(').').

make_product:-
write('S/wproduct.'),
select(X),not(notselect(X)),
write(X),write(', ').
```

make_product: This subroutine print out the maximum product. The maximum product is represented by all variant assigned by *select* predicate and not assign by *notselect* predicate.

6.1 Scalability Test

Scalability is a key factor in measuring the applicability of the techniques dealing with variability modeling in SPL [51]. The output time is a measurement key for scalability. A system consider scalable for specific problem if it can solve this problem in a reasonable time. In the following, we describe the method of our experiments:

Generate the domain engineering as a data set: Domain engineering is generated in terms of predicates (variation points, and variants). We generated four sets containing 1000, 1500, 3000, and 50000 variants. White *et al.* [43] defines 5000 features as a suitable number to mimic industrial SPL. Variants are defined as numbers represented in sequential order, as example: In the first set (1000 variants) the variants are: 1, 2, 3,…, 1000. In the last set (5000 variants) the variants are: 1, 2, 3, …, 5000. The number of variation point in each set is equal to number of variant divided by five, which means each variation point has five variants. As example in the first

set (1000 variants) number of variation points equal 1000. Each variation point defined as sequence number having the term *vp* as postfix, e.g. vp12.

Define the assumptions: We have three assumptions: 1) each variation point and variant has a unique name, 2) each variation point is orthogonal, and 3) all variation points have the same number of variants.

Set the parameters: The main parameters are the number of variants and the number of variation points. The remaining eight parameters (common variants, common variation points, variant requires variant, variant excludes variant, variation point requires variation point, variation point excludes variation points, variant requires variation point, and variant excludes variation point) are defined as a percentage. The number of the parameters related to variant (such as; common variant, variant requires variant, variant excludes variant, variant requires variation point, and variant excludes variation point) is defined as a percentage of the number of the variants. The number of parameters related to variation point (such as; variation point requires variation point) is defined as a percentage of the number of variation points. We found that the maximum ratio of constraint dependency rules used in literature is 25% [51]. Therefore, we defined the ratio of the parameters in our experiments as 25%. **Table 10** represents snapshots of an experiment's dataset, *i.e.*, the domain engineering in our experiments.

Calculate the output: we tested two programs for each program, we made thirty experiments, and calculated the execution time as average. The experiments were done with the range (1000-50000) variant, and percentage range of 25%.

Experimental platform:

The experiments were performed on a computer with an Intel centrino Duo 1.73GHZ CPU, 2 gigabytes of memory, Windows XP home edition. In the following parts, the results are presented. The results show the execution time compared with number of variants, number of variation points, and the parameters.

6.1.1. Test Scalability of a Program to Validate Product in Interactive Mode

In this subsection, we test the scalability of interactive configuration program (**Table 8**). Instead of read the user's input one by one, we define additional parameter, the predicate select(c), where c is random variant. This predicate simulates the user's selection. Number of select predicate (defined as a percentage of number of variant) is added to the domain engineering (dataset) for each experiment. **Table 11** contains a program to validate the product in interactive mode. This program is modification of the program in **Table 8**. This modification allows us to test the scalability.

Table 12 shows the result of scalability test for a program to validate product in interactive mode.

6.1.2. Test Scalability of a Program to Define the Maximum Product

In this subsection, the scalability test for a program to define the maximum product is discussed. **Table 13** shows the results.

In [51] the execution time for 200-300 features is 20 min after applying atomic sets to enhance the scalability. With compare to the literature, our proposed method is scalable.

7. Conclusions and Future Work

In this paper, a method to validate SPL in stage-con-

Table 10. Snapshot of experiment's dataset

```
type(vp1,variationpoint).type(1,variant).
variants(vp1,1).
common(570,yes).
Common(vp123,yes).
requires_v_v(7552,2517).
requires_vp_vp(vp1572,vp1011).
excludes_vp_vp(vp759,vp134).
excludes_v_v(219,2740).
requires_v_vp(3067,vp46).
excludes_v_vp(5654,vp1673).
```

Table 11. A program to validate product in interactive mode

```
sel:-
interactive_validate_exclude,
interactive_validate_require.

interactive_validate_exclude:-
select(N), select(X), X \==N,
((excludes_v_v(X,N), write(' you have to de select '), write(N),nl);
( variants(M,N), excludes_v_vp(X,M),write(' you have to deselect '),
write(N),nl) ;
(variants(Y,X), variants(M,N), excludes_vp_vp(Y,M),write(' you
have to deselect '), write (N), nl)).

interactive_validate_require:-
type(N,variant), select(X), not(select(N)),
((requires_v_v(X,N), write(' you have to de select '), write(N),nl);
( variants(M,N), requires_v_vp(X,M), common (N,yes), write (' you
have to select '), write(N),nl);
(variants(Y,X), variants(M,N), requires_vp_vp(Y,M), com- mon
(N,yes), write (' you have to select '), write(N),nl)).
```

Table 12. Results of scalability test for a program to validate product in interactive mode

Number of variants	Time (Min)
1000	0.4
1500	1.6
3000	12.8
5000	59.6

Table 13. Results of scalability test for a program to generate the maximum product

Number of variants	Time (Min)
1000	1.98
1500	6.85
3000	54
5000	251

figuration process is presented. Firstly, modeling variability using FOL predicates was proposed. By this modeling, we can get formalized variability specifications, support and validate selection process within variability more precisely. The proposed method provides automated consistency checking among constraints (during configuration process) based on three levels (*i.e.* variant-to-variant, variant-to-variation point, and variation point-to-variation point). The proposed method guides users interactively step-by-step (in each choice). If the user's choice is invalid, immediately is rejected and correction actions are suggested (corrective explanation). Moreover, the proposed method can be used to correct future selections using *notselect* predicate (rule 9). All variants selected directly (by user) assigned by *select* predicate. All variants selected using propagation process assigned by *autoselect* predicate. The delete-cascade operation validates auto-select variants. Before cardinality test, all variants in the configured product converted to assign by *select* predicate. Finally, cardinality test validate the number of selection of each variation point.

Many methods are applying empirical results to test scalability by generating random FMs [43,52-54]. Comparing with literature, our test range (1000–5000 variants) is sufficient to test scalability. The proposed method is limited to work only in certain environment, *i.e.* where constraint dependency rules are well known in all cases.

We plan to complete the proposed method by defining operations for validating SPL in static mode. In addition, we plan to implement our method in real life case from industry.

REFERENCES

[1] J. Bosch, "Maturity and Evolution in Software Product Lines," *Proceedings of the Second International Software Product Line Conference*, Springer LNCS, San Diego, Vol. 2379, 19-22 August 2002, pp. 257-271.

[2] P. Clements and L. Northrop, "Software Product Lines: Practices and Patterns," Addison Wesley, Boston, 2001.

[3] K. Kang, 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.

[4] K. Kang, J. Lee and P. Donohoe, "Feature-Oriented Product Line Engineering, *IEEE Software*, Vol. 19, No. 4, 2002, pp. 58-65.

[5] K. Pohl, G. Böckle and F. van der Linden, "Software Product Line Engineering Foundations Principles and Techniques," Springer, Verlag Heidelberg Germany, 2005.

[6] D. Benavides, A. Ruiz-Cort´es, D. Batory and P. Heymans, *First International Workshop on Analyses of Software Product Lines* (*ASPL*'08), Limerick, Ireland, 2008.

[7] D. Benavides, A. Metzger and U. Eisenecker, "Third International Workshop on Variability Modelling of Software-intensive Systems," ICB-Research Report No. 29, University of Duisburg Essen, Duisburg, 2009.

[8] H. Wang, H. Li, J. Sun, H. Zhang and J. Pan, "Verifying Feature Models Using OWL," *Journal of Web Semantics*: *Science, Services and Agents on the World Wide Web*, Vol. 5, No. 2, 2007, pp. 117-129.

[9] D. Batory, D. Benavides and A. Ruiz-Cortés, "Automated Analyses of Feature Models: Challenges Ahead," *Communications of the ACM* (*Special Section on Software Product Lines*), 2006.

[10] K. Czarnecki and U. Eisenecker, "Generative Programming: Methds, Tools, and Applications," Addison-Wesley, Boston, 2002.

[11] T. Massen and H. Litcher, "Determining the Variation Degree of Feature Models," *Software Product Lines Conference*, *LNCS* 3714, Rennes, 2005, pp. 82-88.

[12] T. Asikainen, T. Männistö and T. Soininen, "Using a Configurator for Modelling and Configuring Software Product Lines Based on Feature Models," *Proceedings of the Workshop on Software Variability Management for Product Derivation, Software Product Line Conference* (*SPLC*3), Boston, 2004.

[13] K. Czarnecki, S. Helsen and U. Eisenecker, "Staged Configuration Using Feature Models," *Proceedings of Third International Conference of Software Product Lines SPLC*2004, Boston, 2004.

[14] H. Meyer and H. Lopez, "Technology Strategy in a Software Products Company, Product Innovation Management," Blackwell Publishing, Vol. 12, No. 4, 1995, pp. 294-306.

[15] T. Asikainen, T. Mnnistand and T. Soininen, "Representing Feature Models of Software Product Families Using a Configuration Ontology," *Proceedings of the General European Conference on Artificial Intelligence* (*ECAI*) *Workshop on Configuration*, Berlin, 2004.

[16] M. Schlick and A. Hein, "Knowledge Engineering in Software Product Lines," *Proceedings of the* 14*th European Conference on Artificial Intelligent Workshop on Knowledge-Based Systems for Model-Based Engineering*, Berlin, 2000.

[17] L. Hotez and T. Krebs, "Supporting the Product Derivation Process with a Knowledge Base Approach," *Proceedings of the* 25*th International Conference on Software Engineering ICSE*2003, Oregon, 2003.

[18] L. Hotez and T. Krebs, "A Knowledge Based Product Derivation Process and Some Idea How to Integrate Product Development," *Proceedings of the Software Variability Management Workshop*, Groningen, The Netherlands, 2003.

[19] M. Mannion, "Using First-Order Logic for Product Line Model Validation," *Proceedings of the Second Software Product Line Conference SPLC*2, San Diego, 2002.

[20] W. Zhang, H. Zhao and H. Mei, "A Propositional Logic-Based Method for Verification of Feature Models," *The*

6*th International Conference on Formal Engineering Methods ICFEM*04, *LNCS* 3308, 2004, pp. 115-130.

[21] D. Batory, "Feature Models, Grammars, and Propositional Formulas," *Proceedings of the* 9*th International Software Product Lines Conference SPLC*05, Rennes France, 2005.

[22] J. Sun and H. Zhang, "Formal Semantics and Verification for Feature Modeling," *Proceedings of the* 10th *IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS*05), Shanghai, 2005.

[23] H. Wang, H. Li, J. Sun, H. Zhang and J. Pan, "A Semantic Web Approach to Feature Modeling and Verification," *Proceedings of Workshop on Semantic Web Enabled Software Engineering* (*SWESE'*05), Galway, 2005.

[24] R. Falbo, G. Guizzardi and K. Duarte, "An Ontological Approach to Domain Engineering," *Proceedings of* 14th *International Conference on Software Engineering and Knowledge Engineering*, Ischia, 2002.

[25] V. Dedeban, "Ontology-Driven and Rules-Based System for Management and Pricing of Family of Product," Master Thesis, Norwegian University of Science and Technology Department of Computer and Information Science, Norway, 2007.

[26] F. Shaofeng and N. Zhang, "Feature Model Based on Description Logics," *Proceedings of* 10*th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems KES*2006, Springer-Verlag Berlin Heidelberg, 2006, pp. 1144-1151.

[27] M. Clauss, "Modeling Variability with UML," *GCSE* 2000 -*Young Researchers Workshop*, 3rd GCSE, Erfurt, 2001.

[28] B. Korherr and B. List, "A UML 2 Profile for Variability Models and their Dependency to Business Processes," 18*th International Workshop on Database and Expert Systems Applications*, IEEE, Regensburg, 2007.

[29] T. Ziadi, J. Jezequel and F. Fondement, "Product Line Derivation with UML," *Software Variability Management Workshop*, Groningen, Netherlands, 2003, pp. 94-102.

[30] T. Ziadi and J. Jézéquel, "Product Line Engineering with the UML: Deriving Products," *Chapter in Software Product Lines*, Springer, 2006, pp. 557-586.

[31] E. Oliveira, I. Gimenes, E. Huzita and J. Maldonado, "A Variability Management Process for Software Product Lines," *The* 2005 *Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Centre for Advanced Studies Conference, Toranto, Ontario, 2005, pp. 225-241.

[32] S. Robak, B. Franczyk and K. Politowicz, "Extending the UML for Modelling Variability for System Families," *International Journal of Applied Mathematics and Computer Science*, Vol.12, No. 2, 2002, pp. 285-298.

[33] A. Schnieders, "Modeling and Implementing Variability in State Machine Based Process Family Architectures for Automotive Systems," *The* 3*rd International Workshop on Software Engineering for Automotive Systems ICSE*06, Shanghai, 2006.

[34] H. Gomaa and E. Shin, "Automated Software Product Line Engineering and Product Derivation," *The* 40*th Annual Hawaii International Conference on System Sciences*, Big Island, Hawaii, 2007.

[35] I. Philippow, M. Riebisch and K. Boell, "The Hyper/UML Approach for Feature Based Software Design," *The* 4*th AOSD Modeling with UML Workshop Collocated* 6*th International Conference on the Unified Modeling Language UML*, San Francisco, 2003.

[36] M. Riebisch, K. B¨ollert, D. Streitferdt and I. Philippow, "Extending Feature Diagrams with UML Multiplicities, 6*th World Conference on Integrated Design & Process Technology* (*IDPT*2002), California, 2002.

[37] D. Streitferdt, M. Riebisch and I. Philippow, "Details of Formalized Relations in Feature Models Using OCL," 10*th IEEE International Conference on Engineering of Computer–Based Systems* (*ECBS* 2003), Huntsville, IEEE Computer Society, 2003, pp. 45-54.

[38] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants, *Proceedings of the* 4*th International Conference on Generative Programming and Component Engineering GPCE*'05, Tallinn, Estonia, 2005.

[39] K. Czarnecki and K. Pietroszek, "Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints," *Proceedings of the* 5*th International Conference on Generative Programming and Component Engineering GPCE*'06, Oregon, 2006.

[40] D. Benavides, A. Ruiz-Cort´es and P. Trinidad, "Automated Reasoning on Feature Models," 17*th International Conference* (*CAiSE*05), Porto, 2005, pp. 491-503.

[41] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cort´es, "Using Java CSP Solvers in the Automated Analyses of Feature Models," *Post-Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering* (*GTTSE*), LNCS 4143, 2006.

[42] D. Benavides, "On the Automated Analysis of Software Product Line Using Feature Models, A Framework for Developing Automated Tool Support," Ph.D. Dissertation, University of Sevilla, Sevilla, 2007.

[43] J. White, D. Schmidt, D. Benvides, P. Trinidad and A. Ruiz-Cortes, "Automated Diagnosis of Product Line Configuration Errors on Feature Models," *Proceedings of* 12*th International Conference of Software Product Line*, Limerick Irland, 2008.

[44] F. Cao, B. Bryant and C. Carol, "Automating Feature-Oriented Domain Analysis," *Proceedings of International Conference on Software Engineering Research and Practice* (*SERP*'03), 2003, pp. 944-949.

[45] A. Deursen and P. Klint, "Domain-Specific Language Design Requires Feature Descriptions," *Journal of Computing and Information Technology*, Vol. 10, No. 1, 2002, pp. 1-17.

[46] M. Janota and J. Kiniry, "Reasoning about Feature Models in Higher-Order Logic," *Proceedings of the* 11*th International Software Product Line Conference* (*SPLC*07), Kyoto, 2007.

[47] V. Cechticky, A. Pasetti, O. Rohlik and W. Schaufelberger, "XML-Based Feature Modeling," *Proceedings of the*

8*th International Conference on Software Reuse* (*ICSR*-8), Madrid, 2004.

[48] S. Jarzabek and H. Zhang, "XML-Based Method and Tool for Handling Variant Requirements in Domain Models," 5*th IEEE International Symposium on Requirements Engineering RE*01, IEEE Press, Toronto, 2001. pp. 116-173.

[49] L. Lengyel, T. Levendovszky and H. Charaf, "Constraint Handling in Feature Models," *Proceedings of* 5*th International Symposium of Hungarian Researchers on Computational Intelligence*, Budapest, 2004.

[50] F. Roos-Frantz, "A Preliminary Comparison of Formal Properties on Orthogonal Variability Model and Feature Models," *Proceedings of the* 3*rd International Workshop on Variability Modeling of Software-Intensive Systems*, Sevilla, 2009.

[51] S. Segura, "Automated Analysis of Feature Models Using Atomic Sets," *The* 1*st International Workshop on Analyses of Software Product Lines* (*ASPL*'08), *Collocated with SPLC*08, Limerick Ireland, 12-15 September 2008.

[52] P. Trinidad, D. Benavides, A. Dura´n, A. Ruiz-Cortes and M. Toro, "Automated Error Analysis for the Agilization of Feature Modeling," *Systems and Software*, Vol. 81, No. 6, 2008, pp. 883-896.

[53] P. Trinidad, D. Benavides, A. Ruiz-Cort´es, S. Segura and A. Jimenez, "FAMA Framework," 12*th Software Product Lines Conference* (*SPLC*), Limerick, 2008.

[54] H. Yan, W. Zhang, H. Zhao and H. Mei, "An Optimization Strategy to Feature Models' Verification by Eliminating Verification-Irrelevant Features and Constraints," *Book Chapter in Formal Foundations of Reuse and Domain Engineering*, Springer Berlin/Heidelberg, 2007, pp. 65-75.

