Variability Modeling for Product Line Viewpoints Integration

Nan Niu
Mississippi State University, USA
niu@cse.msstate.edu

Juha Savolainen
Nokia Research Center, Finland
Juha.Savolainen@nokia.com

Yijun Yu
Open University, UK
y.yu@open.ac.uk

***Abstract*—Modern software product line development uses viewpoints to capture the needs of various stakeholders without resorting to a single complex model. Comparing and integrating different viewpoints help to gain insights into the product line and to derive products. Recent research has proposed conflict resolution rules for handling variability in the integration process. However, one benefit viewpoints bring is to tolerate inconsistency until the rationales about variability are better understood. In this paper, we propose a method for modeling variability when product line viewpoints are consolidated. Our method takes advantage of a lattice ordering to support late binding of variability and stakeholder traceability. We apply our method to viewpoints derived from the mobile phone domain, and show how delayed commitment can support product line evolution and product derivation.**

***Keywords*-Viewpoints; product line; variability; toleration of inconsistency**

I. Introduction

Product line engineering has become the main method for achieving systematic reuse [13]. A software product line succeeds because mass customization is achieved by effective management of domain's commonality and variability. The more commonality there is, the easier the reuse and implementation of the product line assets. The more rigorous the variability is specified, the more efficient the mechanism required to support it.

The product line model contains the variability information of the domain, often expressed in terms of *features* that the products contain [14], [15]. A feature represents a product characteristic that customers view as important in describing and distinguishing members of a software product line [14]. A feature is mandatory if it is essential to the domain, and is optional otherwise.

Traditionally, a single feature model is constructed to capture all the variability and constraints on the variability in the domain. In this context, consistency management requires that only the admissible products can be derived from the feature model and all invalid products are prevented from being derived. In [31], we have described a method that assures consistency for product lines that have a holistic feature model.

In practice, product lines can have thousands of features with hundreds of variation points, complex selection rules, and share assets with other product lines [2], [10]. This makes developing and evolving a single product line model a daunting task. A number of feature model decomposition solutions are therefore proposed to alleviate this problem: Feature dependencies can be separated from the variability model [10], a separate decision model can be created for easier derivation [6], and complex variability decisions can be divided into product sets [28]. Although these models have their advantages, one significant drawback is that it is not easy to trace the choice of a feature's variability type back to its source nor to understand where there might be competing views. Missing traceability information can have serious consequences in industrial product lines by hindering the evolution of variability [30].

In our earlier work [21], we leveraged *viewpoints* [12] to develop the product line model, in which stakeholders are able to maintain their own partial models about the domain and its variability, without being constrained by other stakeholders' views. The key insight is that feature variability is only relative to specific stakeholder view, and this information plays an important role in achieving a compromise among different viewpoints. The approach separates out competing stakeholder priorities of feature variability, and allows variability decisions to be traced back to their sources. To integrate inconsistent viewpoints, some conflict resolution rules are introduced, e.g., if a feature in one viewpoint is "mandatory" but the same feature in another viewpoint is "optional", then the feature becomes "mandatory" in the composite viewpoint [21].

However, a prominent aspect of exploiting viewpoints is toleration of inconsistency [26], since the underlying tenet of viewpoints is that modeling different perspectives is better than building a single coherent model [11]. This suggests that decisions about inconsistency resolution can be delayed until the distinctive rationales are better understood [8]. We believe such delayed commitment is particularly useful for large and evolving product lines, in which multiple competing views are continuously (re-)balanced.

For instance, there is often a tension between marketing and engineering departments about which features must or must not be included in a given new product. Marketing often bases the arguments on perceived customer demand, whereas engineers often refer to current architectural constraints. In addition, as the diversity in a software product line evolves over time, feature variability may change from one type to another. For example, some mandatory features

Table I
FEATURE VARIABILITY

Variability Type	Notation	Semantics
Disallowed	F_i	$\neg F_i$
Mandatory	F_j	$F_j \longleftrightarrow F_p$
Optional	F_k	$F_k \longrightarrow F_p$
Alternative*	F_l F_m	$F_p \longleftrightarrow (F_l \oplus F_m)$

* For more than two alternative features, the operator $\oplus$ evaluates to true if exactly one of the alternatives is true; otherwise to false.

may become obsolete as product characteristics are phased out; some optional features may become mandatory as product characteristics become popular and/or less expensive to implement [32].

In this paper, we propose a method that tolerates variability inconsistency among product line viewpoints. We aim to enhance the evolution of a product line infrastructure by delaying the binding of variability with a mechanism for stakeholder buy-in and traceability. To that end, we explore lattice orderings to express inconsistencies and variability properties as viewpoints are integrated. We apply our method to viewpoints derived from the mobile phone domain, and show how late binding of variability can support product line evolution and product derivation.

The remainder of the paper is organized as follows. Section II motivates our work with a scenario for integrating product line viewpoints. Section III explores lattice orderings for characterizing variability properties. Section IV proposes our method for modeling variability during viewpoints integration. Section V presents an application of our method. Section VI reviews related work. Section VII draws some concluding remarks and outlines future work.

II. MOTIVATING EXAMPLE

Product line features are distinctively identifiable abstractions that must be implemented, tested, delivered, and maintained [15]. A feature model [14] is a tree-like structure, in which the features are related to each other in parent-child relationships. Features that are considered to elaborate on another feature can be made children of this feature. Feature variability is specified by the selection type, which indicates whether a feature should be realized in a specific product. In this paper, a selection type takes one of four values: *disallowed*, *mandatory*, *optional*, or *alternative*.

Table I lists the four variability types and their respective modeling notations. The semantics are given in propositional formulas [1]. By writing a Boolean variable for each feature and by defining a logical operator for each selection constraint relationship between features, a logical expression

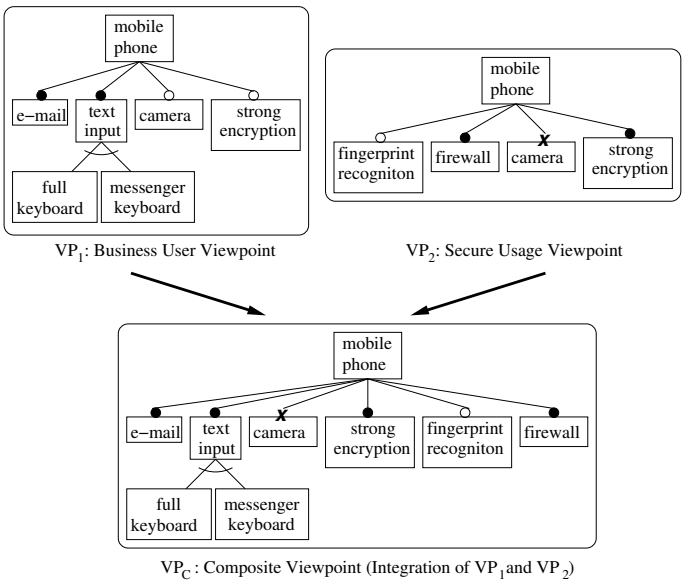


Figure 1. Viewpoints integration example

for a feature model can be developed, and used to verify selection decisions made from the feature model [21].

In Table I, F_p denotes the parent feature of the feature(s) in question. For a feature to be selected in product line members, its corresponding formula must hold. For example, a mandatory feature means that whenever its parent is selected, it must be selected, and vice versa. Alternative features, such as F_l and F_m in Table I, dictate an exclusive-or choice [14].

In modern software product development, tension often exists between competing priorities and constraints, each of which is usually represented by a primary stakeholder. In [21], we introduced *product line viewpoints* to model variability from separate perspectives. In this context, a viewpoint is a subset of a master feature model containing only the features (and their interrelations) that are relevant for a given point of view [21]. In this paper, a viewpoint contains the leaf features and all their parents up to the root feature [4].

Figure 1 shows our running example of integrating viewpoints in the mobile phone domain. VP_1 shows the feature model from the business user's viewpoint. The secure usage viewpoint, VP_2, defines the features for users needing high security in their mobile phones. Secure usage is typically important for some business users but also technology leaders. If we want to create a product for business users that require a secure mobile phone, we must integrate these two viewpoints. The overlapping features, **camera** and **strong encryption**, have different variability types. To resolve such inconsistencies, two rules can be followed [21]:

- Rule # 1(II): When a feature in one viewpoint is *disallowed* but the same feature in another viewpoint is *optional*, then the feature's variability type becomes *disallowed* in the composite viewpoint.

- Rule # 2: When a feature in one viewpoint is *mandatory* but the same feature in another viewpoint is *optional*, then the feature's variability type becomes *mandatory* in the composite viewpoint.

One advantage of applying these rules is obtaining a definitive view so that the result ($\mathrm{VP_C}$ in Figure 1) is consistent overall. However, a number of drawbacks exist. First, there are special cases that a default rule may not be applicable, which requires addition of new rules or exceptions. Second, feature's variability type can change as viewpoints evolve. For example, **camera** is becoming an integral part of a mobile phone nowadays, so business users will soon regard it as a *mandatory* feature. This evolved view, when integrated with $\mathrm{VP_2}$ in Figure 1, leads to an unresolved situation, even by applying rule # 1(I) in [21]. Third, traceability information becomes implicit, e.g., it is difficult for a new product derived from $\mathrm{VP_C}$ to trace the source of the decision about **camera** being *disallowed*, without resorting to all the input views and their evolutions.

We aim to overcome these problems via toleration of inconsistency [26]. We argue that it is beneficial, especially for large and evolving product lines, to delay the commitment to inconsistency resolution until the rationales about feature variability are better understood. While the benefits of tolerating inconsistency are further illustrated in Section V, we now explore formalisms to characterize variability orderings.

III. Variability Orderings

Definition 1 A *partial order* is a reflexive, antisymmetric, and transitive binary relation. A non-empty set with a partial order on it, $(A, \leq_A)$, is called a *partially ordered set* or a *poset*. We use Hasse diagrams [5] to visualize finite posets.

Definition 2 The *Cartesian product* of two posets $(A, \leq_A)$ and $(B, \leq_B)$ is a poset $(A \times B, \leq_\times)$ with elements (a, b), such that the partial order $\leq_\times$ holds between two pairs if and only if it holds for each component separately, i.e., $(a, b) \leq_\times (a', b') \leftrightarrow (a \leq_A a') \wedge (b \leq_B b')$.

Definition 3 Let $(A, \leq_A)$ be a poset and $S \subseteq A$. An element $v \in A$ is an *upper bound* of S if $\forall s \in S : s \leq_A v$. If v is an upper bound of S and $v \leq_A w$ for all upper bounds w of S, then v is called the *least upper bound* or *supremum* of S. Dually, an element $v \in A$ is a *lower bound* of S if $\forall s \in S : v \leq_A s$. If v is a lower bound of S and $w \leq_A v$ for all lower bounds w of S, then v is called the *greatest lower bound* or *infimum* of S. We write $\sqcup_A S$ (respectively $\sqcap_A S$) to denote the supremum (respectively infimum) of $S \subseteq A$, when it exists.

Definition 4 Let $(A, \leq_A)$ be a poset. If both $\sqcup_A\{a, b\}$ and $\sqcap_A\{a, b\}$ exist for any $a, b \in A$, then $(A, \leq_A)$ is called a *lattice*. If both $\sqcup_A S$ and $\sqcap_A S$ exist for any $S \subseteq A$, then $(A, \leq_A)$ is called a *complete lattice*.

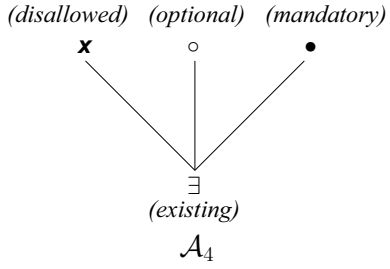


Figure 2. Variability ordering $\mathcal{A}_4$

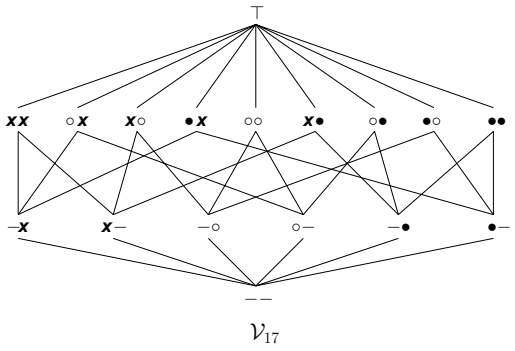


Figure 3. Variability lattice $\mathcal{V}_{17}$

Theorem 1 *(cf. e.g., [5]) Every finite lattice is complete.*

The purpose of introducing the definitions of poset and lattice is to explore ordering among different variability types. Since alternative features represent specializations of a mandatory or an optional feature [4], we consider more primitive values, *disallowed*, *mandatory*, and *optional* in our current work. Figure 2 shows a poset $\mathcal{A}_4$ in Hasse diagram. $\mathcal{A}_4$ presents a variability ordering where an existing feature is *disallowed* or becomes *mandatory* or *optional* during product line evolution. Note that *existing* ($\exists$) has little variability information, so it is at the bottom of the ordering.

If we attempt to integrate two viewpoints, we need the Cartesian product of the poset. Following Definition 2, the product $\mathcal{A}_4 \times \mathcal{A}_4$ contains 16 elements. We modify the value *existing* ($\exists$) to *unknown* ($-$), and add a top element ($\top$) to form a lattice. The resulting ordering, $\mathcal{V}_{17}$, is shown in Figure 3. Following Theorem 1, $\mathcal{V}_{17}$ is a complete lattice.

All the elements in $\mathcal{V}_{17}$, except for the top ($\top$), have two components. We interpret the first (resp. second) component to represent the first (resp. second) viewpoint. For instance, "$\circ-$" means that the feature is specified to be optional ($\circ$) in the first viewpoint, and its variability type is unknown ($-$) in the second viewpoint. The element "$\bullet\times$" denotes that the feature is regarded to be mandatory ($\bullet$) in the first viewpoint, but is disallowed ($\times$) in the second viewpoint. We use a top element ($\top$) in $\mathcal{V}_{17}$ to represent incompatible information.

The lattice $\mathcal{V}_{17}$ and its corresponding variability ordering arise when the input is $\mathcal{A}_4 \times \mathcal{A}_4$. A suitable lattice ordering for the integration of three viewpoints will arise when the input is $\mathcal{A}_4 \times \mathcal{A}_4 \times \mathcal{A}_4$, and so on.

IV. Integrating Product Line Viewpoints

In this section, we present a systematic way to handle and trace variability for viewpoints integration. Our current focus in on syntactic inconsistency rather than structural one [25]. We first introduce the use of a connector for mapping structures (features and their parent-child relationships) and stakeholder vocabularies. Then, we leverage the variability ordering to automate the generation of the composite viewpoint. Finally, we show how inconsistency can be analyzed and tolerated.

A. Mapping Vocabulary and Structure

Software product line engineering considers it crucial to define a set of standard terms used in discussions about and descriptions of the domain, and makes developing a domain dictionary part of the core assets [35]. Domain vocabulary provides the definitions of terms, acronyms, and abbreviations, and identifies synonym classes. However, mismatches in stakeholders' vocabulary often occur [23], as an area of surprising controversy. Experts would occasionally misunderstand one another, because they were using the same words in different ways. In fact, experts would sometimes be in "violent agreement" with one another, all the while expressing the same idea in different terms [24].

This requires a joint effort among domain experts in codifying the glossary and relating their personal constructs to the glossary. Our method employs a connector to capture the vocabulary mapping. Figure 4 shows a scenario of interconnecting viewpoints based on our running example (cf. Figure 1). The business user's viewpoint evolves ($\mathrm{VP}_{1.1}$) by adding a new optional feature, **instant messenger**.

Three connectors are shown in Figure 4 that build pairwise mappings between the viewpoints. Since the evolution mapping ($\mathrm{VP}_1 \rightarrow \mathrm{VP}_{1.1}$) is straightforward, no connector is constructed. Although $\mathrm{VP}_{1.1}$ and VP_3 use different terminologies to express keyboard-related features, C_2 captures their correspondences so that synonyms are not treated as separate features during viewpoints integration. In case of name clashes, i.e., two viewpoints use the same label for different features, the connector helps distinguish the features by assigning each one a distinct label.

The connector unifies not only vocabulary, but also structure of the viewpoints. Structure in this context refers to the features and their parent-child relationships. For example, the two-level parent-child hierarchy presented in C_2 is preserved in both $\mathrm{VP}_{1.1}$ and VP_3. The interconnecting arrows in Figure 4 are formally known as *graph homomorphisms*, and they ensure that we put structures together with nothing essentially new added and nothing left over [25].

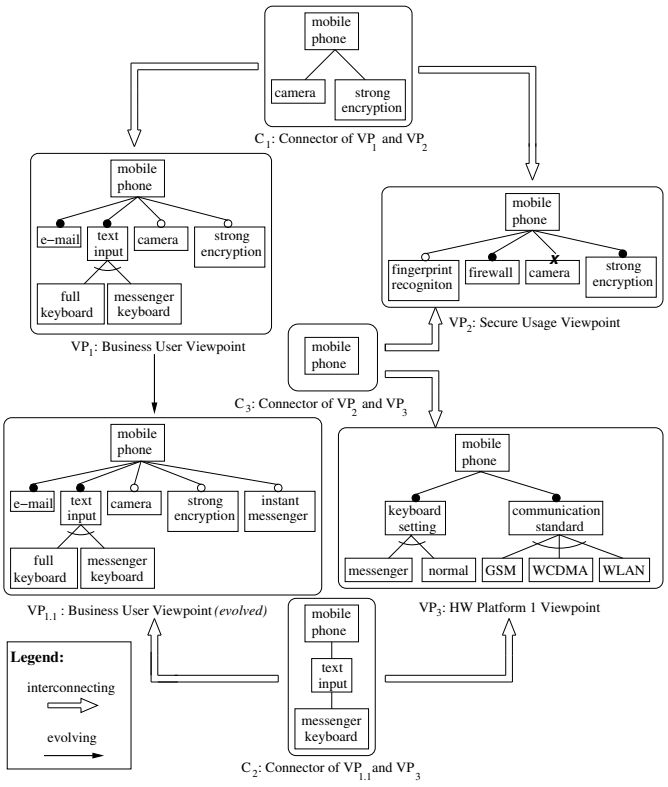


Figure 4. Interconnecting viewpoints

We use a distinct set of connectors to explicitly represent viewpoints' interconnections for a couple of reasons. First, each connector is a hypothesis for standardizing vocabulary and feature hierarchy, so it is an important asset to the product line [35]. Second, encapsulating the interconnection information within a single entity eases modification and evolution, and also reduces the clutter of the diagram. Note that each connector shown in Figure 4 interconnects only two viewpoints, but in general, we can build a connector that unifies any finite set of viewpoints. For instance, the connector of all 4 viewpoints shown in Figure 4 would have only one top-level feature, **mobile phone**, which is equivalent to C_3. We use pairwise connectors to illustrate our method since they are able to express a larger amount of overlap between viewpoints. Although heuristics exist for constructing connectors, manual effort is required for building the connectors [29].

To assign a name to each feature of the composite viewpoint, we should combine the names of all the features in the individual viewpoints that are mapped to it. For example, the composite viewpoint of $\mathrm{VP}_{1.1}$ and VP_3 would have a feature labeled $\{\textbf{text input}_{\mathrm{VP}_{1.1}}, \textbf{keyboard setting}_{\mathrm{VP}_3}\}$, indicating the names used in the two contributing viewpoints. A better way to name the features of the composite viewpoint is assigning naming priorities to the input viewpoints. For example, once a connector is devised, it makes sense to give priority to the feature names in the connector, C_2, and label

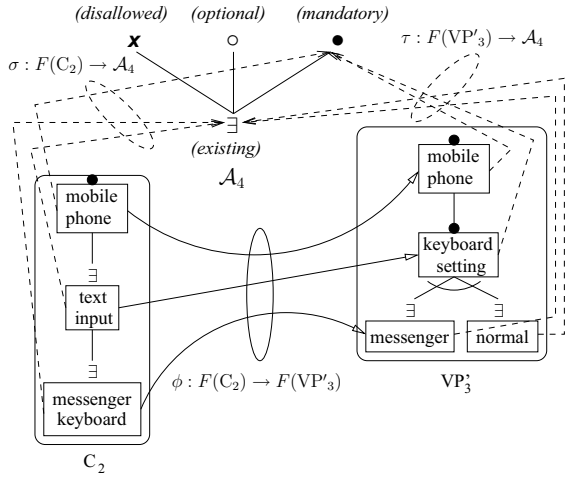


Figure 5. $\mathcal{A}_4$-preserving homomorphism

the above feature **text input** in the composite viewpoint. This naming convention is of no theoretical significance, but it provides a natural solution to the vocabulary mapping problem: in most cases, we would like the choice of names in connector objects, i.e., objects solely used to describe the relationships between other objects, to have precedence in determining the element names in the integrated object. We will use this convention in the rest of this paper.

B. Preserving Variability Ordering

Although stakeholder vocabulary and feature structure are preserved, there is a lack of support for handling variability. We address this problem via the variability orderings defined in Section III. The idea is to attach each feature an annotation denoting the variability type/degree and the way in which variability can evolve (or degree can grow). Due to construction, an upward move in the ordering denotes a growth in the amount of variability information, i.e., an evolution of specificity.

Figure 5 illustrates the idea, in which features of C_2 and VP'_3 (a projection of VP_3) are annotated with the variability ordering $\mathcal{A}_4$. Let F be the function that takes a feature model M, and returns all the features contained in M. In Figure 5, for example, $F(\mathrm{C}_2) = \{$**mobile phone**, **text input**, **messenger keyboard**$\}$. [1] In our work, variability is a property of the node, not the edge, in the feature model. We therefore tease out the node-wise function of a graph homomorphism [25]. In Figure 5, ϕ: $F(\mathrm{C}_2) \to F(\mathrm{VP}'_3)$ denotes such a function, which formalizes the vocabulary and structure mapping discussed in Section IV-A.

Definition 5 Let $\mathcal{A}$ be a variability ordering. A feature model, M, is *$\mathcal{A}$-annotated* by the function σ: $F(M) \to \mathcal{A}$ if each feature of M is annotated with an element drawn from $\mathcal{A}$. For every $m \in F(M)$, the value $\sigma(m)$ is interpreted as the *degree of variability* of m in M.

Definition 6 Let M_1 and M_2 be two $\mathcal{A}$-annotated feature models with annotation functions σ: $F(M_1) \to \mathcal{A}$ and τ: $F(M_2) \to \mathcal{A}$. An *$\mathcal{A}$-preserving homomorphism* is a node-wise function of a graph homomorphism ϕ: $F(M_1) \to F(M_2)$ such that $\sigma \leq_{\mathcal{A}} \tau \circ \phi$, i.e., for every $m \in F(M_1)$, the degree of variability of m in M_1 does not exceed that of $\phi(m)$ in M_2.

The condition in Definition 6 ensures that variability is preserved as we traverse a mapping between annotated viewpoints. For example, if a stakeholder has already specified a feature to be mandatory ($\bullet$ in $\mathcal{A}_4$), it cannot be embedded in another viewpoint such that it is reduced to just existing ($\exists$), or is changed to a value not comparable to mandatory, i.e., optional ($\circ$) or disallowed ($\not$). It can be checked that, in Figure 5, ϕ is an $\mathcal{A}_4$-preserving homomorphism.

The composite viewpoint $\mathrm{VP_C}$ resulting from integrating a finite set of $\mathcal{A}$-annotated feature models, which are interconnected by $\mathcal{A}$-preserving homomorphisms, is built in two steps. First, put feature names and structures together according to the connectors. Second, attach an annotation to every feature f in $\mathrm{VP_C}$ by taking the *least upper bound* (cf. Definition 3) of the annotations of all the features that f represents. Intuitively, the least upper bound of a set of variability degrees $S \subseteq \mathcal{A}$ is the least specific degree that refines (i.e., is more specific than) all the members of S. Note that once the connector is built, the composite viewpoint generation can be fully automated.

As an example, Figure 6 shows the scenario of using the variability lattice $\mathcal{V}_{17}$ to integrate VP_1 and VP_2 with their interconnections expressed in C_1. The annotation procedure is straightforward. [2] First, annotate features in the connector with $--$, indicating variability is unknown yet. Then, annotate features in VP_1 with $v-$ where v is the variability type appeared in VP_1 initially; this indicates that feature variability is specified in the first viewpoint but unknown in the second. Features in VP_2 can be annotated analogously. The above annotation procedure gives rise to $\mathcal{V}_{17}$-preserving homomorphisms, $F(\mathrm{C}_1) \to F(\mathrm{VP}_1)$ and $F(\mathrm{C}_1) \to F(\mathrm{VP}_2)$, so that $\mathrm{VP_C}$ can be generated automatically. To illustrate variability annotation in $\mathrm{VP_C}$, we focus on the **camera** feature. Since its variability in C_1, VP_1, and VP_2 is $--$, $\circ-$, and $-\not$ respectively, the annotation for **camera** in $\mathrm{VP_C}$ is calculated by taking $\sqcup_{\mathcal{V}_{17}}\{--, \circ-, -\not\}$ resulting in the value $\circ\not$. The fact that $\mathcal{V}_{17}$ is a complete lattice ensures $\sqcup_{\mathcal{V}_{17}} S$ exists for every $S \subseteq \mathcal{V}_{17}$.

C. Tolerating Inconsistency

Viewpoints permit building and evolving many partial, overlapping models, without enforcement of consistency

[1] We assume no name clashes occur within any single viewpoint's feature model. Name clashes between different viewpoints are resolved through the connector (cf. Section IV-A).

[2] The root node of every feature model is enforced to be mandatory [14]; no other annotation shall change this value.

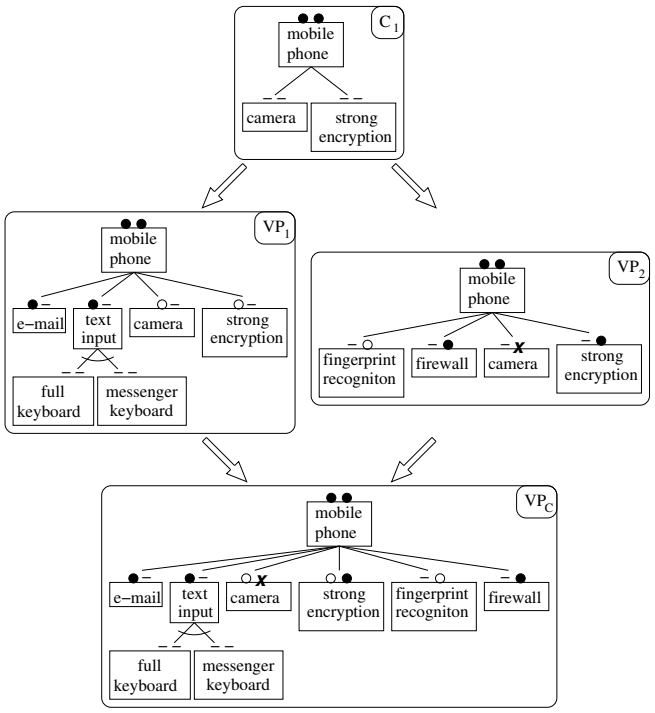


Figure 6. $\mathcal{V}_{17}$-preserving integration

between them [8]. A fundamental problem in comprehending software evolution is the choice of effective visual representations for data that are not inherently physical. The goal is an insightful rather than a faithful depiction of the data [9]. Multiple views are often shown side-by-side in current software evolution tools. This increases developers' cognitive overhead when they are in the process of understanding multiple competing viewpoints. In our framework, the composite viewpoint, e.g., $\mathcal{V}_{17}$-annotated $\mathrm{VP_C}$ in Figure 6, characterizes much variability knowledge and its evolution within one view. This provides engineers with a centralized vision to discover and handle inconsistencies.

The variability lattice $\mathcal{V}_{17}$ offers interpretations of stakeholder viewpoints according to certain semantics. In our framework, a system of interconnected $\mathcal{V}_{17}$-annotated feature models is *syntactically inconsistent* if the composite viewpoint has some feature with an inconsistent annotation.

It is up to the product line management team to designate (in)consistent values of $\mathcal{V}_{17}$. For example, we may regard XX, $\circ\circ$, and $\bullet\bullet$ as consistent and the rest of the values as inconsistent. This is a reasonable choice when the system we are modeling mandates the total agreement of both stakeholders on every aspect. If we are only interested in explicit conflicts and incompatibilities, we can relax this constraint and designate only $\mathit{X}\bullet$, $\bullet\mathit{X}$, and $\top$ as inconsistent. Once we have a measure for how much inconsistency we want to tolerate, the composite viewpoint construction can also serve as a mechanism for determining when an inconsistency amelioration phase [8] is required. When using

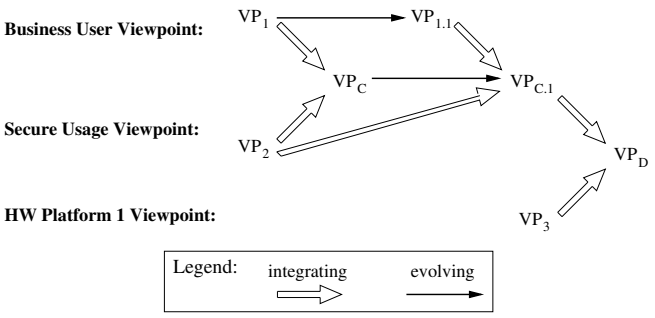


Figure 7. Integration of evolving viewpoints

$\mathcal{V}_{17}$, for example, we may decide to live with all $\mathcal{V}_{17}$ values except for incompatibilities ($\top$).

After an integration is completed, it is desirable to know how the input viewpoints have influenced the result. Particularly, it is important to be able to trace the feature variability of the composite viewpoint back to the originating viewpoint, and to track the rationales behind each variability decision. We call the former notion *stakeholder traceability* and the latter *rationale traceability*. Our method directly supports stakeholder traceability due to the Cartesian product ordering employed. By inspecting merely the annotations of $\mathrm{VP_C}$ in Figure 6, for example, we can easily identify the contributions made by $\mathrm{VP_1}$ and $\mathrm{VP_2}$. This visualization aid also helps to achieve rationale traceability, in that each stakeholder's decision is explicitly recorded. For example, the rationale for eventually setting **camera** (annotated with $\circ\mathit{X}$ in $\mathrm{VP_C}$) to be disallowed could be that the feature is strictly prohibited in $\mathrm{VP_2}$ and considered only as an add-on in $\mathrm{VP_1}$.

V. Worked Example

We use a set of mobile phone viewpoints [21] to illustrate how our method can be applied in practice. The benefits of our method are demonstrated in two stages: integration of evolving viewpoints and derivation of new products from the integrated view. The annotation in our study is based on $\mathcal{V}_{17}$.

Figure 7 presents an overview of the integration scenario. The business user ($\mathrm{VP_1}$) and secure usage ($\mathrm{VP_2}$) viewpoints are first integrated into $\mathrm{VP_C}$. Then, $\mathrm{VP_1}$ evolves to $\mathrm{VP_{1.1}}$, which leads to an evolved composite viewpoint $\mathrm{VP_{C.1}}$ (integration of $\mathrm{VP_{1.1}}$ and $\mathrm{VP_2}$). Finally, the HW Platform 1 viewpoint ($\mathrm{VP_3}$) is considered and merged with $\mathrm{VP_{C.1}}$, resulting in the composite viewpoint $\mathrm{VP_D}$.

The integration of $\mathrm{VP_1}$ and $\mathrm{VP_2}$ is shown in Figure 6. The evolution of the business user's viewpoint is depicted in Figure 4. Compared to $\mathrm{VP_1}$, $\mathrm{VP_{1.1}}$ adds a new optional feature **instant messenger**. This feature, if selected, requires the selection of **messenger keyboard** to be the product's text

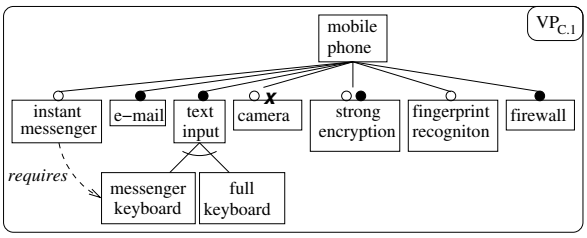


Figure 8. Evolved composite viewpoint

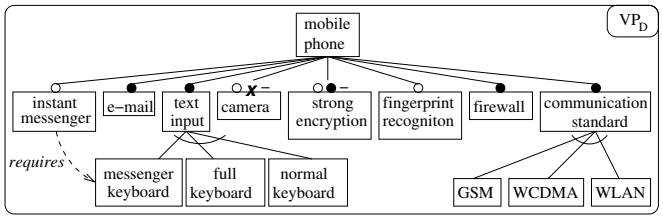


Figure 9. Final composite viewpoint

input. Figure 8 [3] shows the composite viewpoint $\mathrm{VP_{C.1}}$, in which the 'requires' dependency is shown in a dotted arrow. Note that the variability of several features has already been bound. The rationales are as follows:

- The root is mandatory, so annotation is omitted [14].
- The unique features of $\mathrm{VP_{1.1}}$ are annotated with their variability types in $\mathrm{VP_{1.1}}$, e.g., **e-mail** is annotated with $\bullet$ instead of $\bullet-$.
- Similarly, the unique features of $\mathrm{VP_2}$ are annotated with their original variability types.

Although late variability binding is useful, it is crucial to make decisions when appropriate. The extreme case of late binding is to treat all the features as optional, and ask each user to dynamically select her desired feature set at run-time. This is clearly absurd in that every product in the product line would have been built in exactly the same way, i.e., containing all possible features. Therefore, timely binding feature variability and recording the rationale are essential. For instance, the decision of **text input** being *mandatory* can be justified from the business user's point of view.

Figure 9 shows the integrated final view by taking $\mathrm{VP_3}$ (a particular hardware platform) into account. For **text input**, $\mathrm{VP_3}$ allows the selection of a **normal keyboard** representing the traditional number keyboard or a **messenger keyboard** with a foldout two-piece keyboard. When merging this view with $\mathrm{VP_{C.1}}$, we decide to insert **normal keyboard** to the alternative list of **text input**. The rationale stems from the exclusive-or semantics associated with alternative features (cf. Table I). In the composite viewpoint $\mathrm{VP_D}$, the annotations for **camera** and **strong encryption** are expanded, so that stakeholder traceability information is updated. Toleration of inconsistency, one of the main benefits of our method, is sufficiently demonstrated in the above integration scenario, where evolving and competing viewpoints are consolidated. The final integrated feature model $\mathrm{VP_D}$ serves the baseline for not only stakeholder negotiation but also product derivation.

[3] Alternative features are not annotated in this section; we are currently exploring orderings that incorporate alternation.

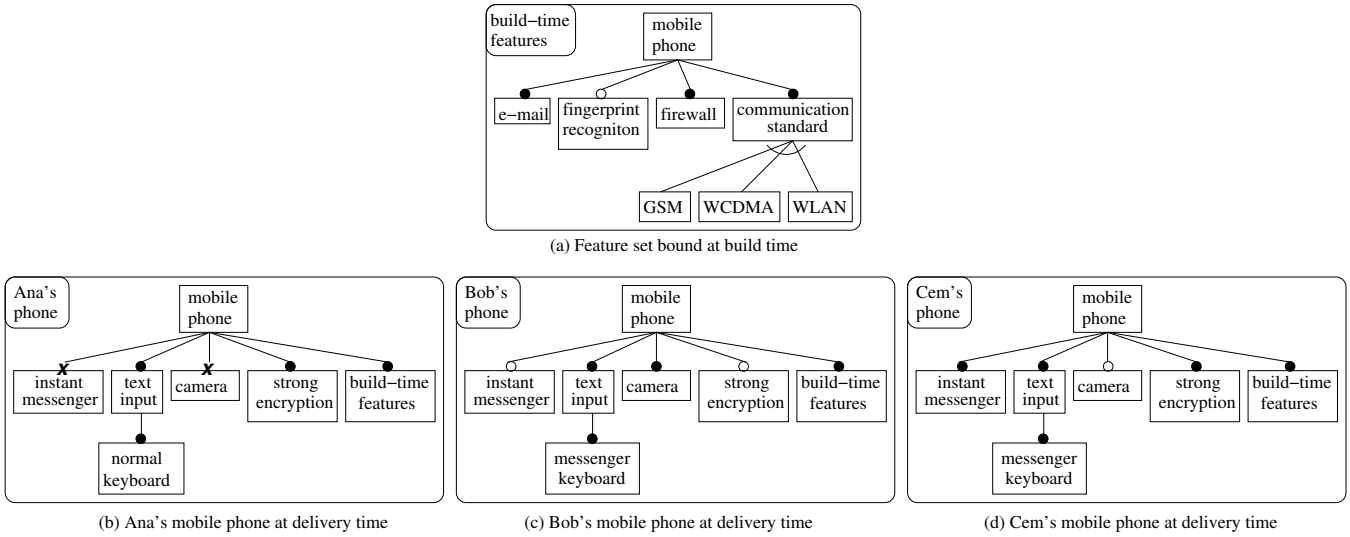
Table II
PRODUCT DERIVATION SCENARIOS

User	Primary Concern	Specific Requirement
Ana	network security	eliminating known vulnerabilities
Bob	social networking	a camera phone & mobile blogging
Cem	news & stock	requiring mobile instant messaging

We consider three product derivation scenarios, as listed in Table II. The three users, Ana, Bob, and Cem, categorize themselves as business users. They all want their mobile phones to be equipped with security features but to various degrees. The product line team working on HW Platform 1 has used our method and generated an integrated feature model, $\mathrm{VP_D}$. Figure 10a shows the feature set bound at build time. Every mobile phone derived in the product family must include these features and must not alter the variability types of these features.

In addition to common views, the three users have their own primary concerns and specific requirements, as listed in Table II. We are able to deliver customized phones to accommodate their needs, thanks to our method's delayed commitment to resolving inconsistencies. Figures 10b, 10c, and 10d show the configured features at delivery (installation) time for Ana, Bob, and Cem respectively.

- Ana has a high security demand, so the secure usage viewpoint ($\mathrm{VP_2}$) takes precedence in customizing her phone, i.e., **camera** is disallowed (X) and **strong encryption** is enforced ($\bullet$). Ana specifically requires any feature with known security vulnerability be excluded from her phone. Since instant messaging has been exploited to deliver virus, trojan, or spyware within an infected file, **instant messenger** is disallowed (X). Ana selects a **normal keyboard** ($\bullet$) due to its simplicity.
- Bob wants a camera phone to do mobile blogging and other social networking activities. This makes **camera** a must ($\bullet$) in his phone. Compared to Ana, he has a lower security demand, so **strong encryption** can be turned on or off ($\circ$) at run-time. Based on his social networking need, **instant messenger** is enabled but not enforced ($\circ$). Even though this feature is optional, **messenger keyboard** must be included in the configuration ($\bullet$) to support Bob's option at run-time.
- Cem often uses his phone to track news and stock information, so he requires **instant messenger** ($\bullet$). This selection makes **messenger keyboard** mandatory ($\bullet$). He is willing to adopt secure usage practice, e.g., **strong encryption** ($\bullet$). To Cem, **camera** is considered a nice-to-have feature ($\circ$).



(a) Feature set bound at build time

(b) Ana's mobile phone at delivery time

(c) Bob's mobile phone at delivery time

(d) Cem's mobile phone at delivery time

Figure 10. Products derived to meet different stakeholders' needs

It is clear from the above derivation scenarios that delayed commitment makes our product line model more flexible to accommodate a broader set of user requirements. Suppose we do not tolerate inconsistency but apply the inconsistency resolution rules [21] to the **camera** and **strong encryption** features, then all the mobile phones derived from the product line would disallow (✗) **camera** and enforce (•) **strong encryption** (cf. $\mathrm{VP_C}$ in Figure 1). Such a configuration works only for Ana, one of the user groups supported by our method.

VI. Related Work

The importance of variability Understanding why a certain type of variability exists in a product line is important. For very simple product lines, variability is just a sum of selections made by products. If at least one product wants a feature that is omitted by other products, then the feature becomes optional. If all the current products want a feature, then it becomes mandatory. In this context, consistency management implies the ability to check that product selections of features are consistent with the variability types [31].

It has long been understood that the future products must also be considered when defining variability. This is needed to prevent defining such variability types that become invalid soon [27]. When the complexity of the product line infrastructure increases, the cost of changing variability increases as well. This makes an incentive to choose such variability types that tend to be correct for the foreseeable future.

As the company grows, the sources for future product requirements tend to increase. In practice, the chosen variability type is a consensus over a large number of stakeholders. During evolution, one must understand the complete rationale on how the interest of each stakeholder is reflected in the decision on the variability type. This information is normally lost. One approach is to attach rationale for each point of variation [34]; however, the rationale tends to be outdated soon. Based on our experience, practitioners often find that the maintenance effort of adding rationale as free text to be excessive in comparison with the benefits.

Consistency checks for variability Recently, a number of researchers have observed the need for consistency checking. Lauenroth and Pohl [17], [18] agree with our view that all product line requirements contain inconsistencies, because they describe multiple products in the same specification. They have proposed a framework for automatic consistency checking prior to derivation [17]. However, they do not check product line requirements against a specific derived product, but against all possible products. In some product lines, this may become infeasible.

A product line model can contain a large number of invalid configurations that are never chosen in practice because the engineers have enough domain knowledge to avoid these invalid configurations. In these cases, consistency checking of all configurations may find a large number of errors that are tolerable in practice. However, in some application domains with critical quality attributes, analyzing the whole product line may be appropriate [20].

Recently, feature diagrams have been used as part of input for model checking [3] and business process configuration management [16]. One difference in our work is that we provide composite feature diagrams that help with inconsistency management of multiple viewpoints. The feature notation used in our work can be extended to handle the general form of feature diagrams [33] which augment mandatory feature with cardinality $(m : n)$.

Stakeholder viewpoints and inconsistency management Orthogonal to the complexity of having multiple contexts of different products, integrating different stakeholder view-

points can further complicate the models for inconsistency management [8], [11], [26].

Using a category-theoretic approach [25], model merging has been applied to various kinds of artifacts belonging to different viewpoints, such as goal models for requirements [29], statecharts for design [22], and call graphs for implementation [25]. Combining this with the dimension of variability management, an algebraic approach has been proposed [7] as a mathematical foundation, which is made more practical in our work to extend feature variability models with more explicit notations for representing viewpoints. This extension helps domain experts to visualize variability and track stakeholder viewpoints directly.

Product line variability elicitation Variability for product line requirements may arise from different sources. Recently, researchers have considered natural language as a rich way to elicit variability [19], [24]. Based on the frame theory of language, various answers to different angles of questioning – e.g., why, what, who, where, when, how, how much, etc. – are a natural way to elicit variability directly from stakeholders. Similarly, variability in feature models can be elicited from the models of early requirements [36], or from the design or implementation of a software system [37].

Recording variability directly on the various viewpoint sources rather than using the isolated texts presents the advantage that one can analyze the different alternatives using a language specific to the viewpoints of the stakeholders. On the other hand, our work complements textual rationale logs: once variability in the feature models has been elicited, we are able to record explicitly the references to viewpoints in the original source. As a result, one can analyze the inconsistencies together with the features, without resorting to the original sources. Furthermore, the results of our inconsistency management can be fed back to the original sources as long as their traceability is maintained.

VII. Conclusions

Managing variability is key to successful product line development. Too much variability reduces the benefits of software reuse, potentially making the product line unprofitable. Too little variability can lead to product lines with a limited life and scope. When product line evolves, the feature selection constraints should be changed to match the current situation. In practice, this has proven to be very difficult.

To make a correct decision on feature variability, one must understand the rationale why a certain feature selection constraint was originally introduced. Since each selection is typically a compromise between multiple business and technology stakeholders, changing selection constraints needs interacting with all the relevant parties. Our earlier work [21] has explored the use of viewpoints [12] to structure evolving product line features. We also defined some default rules to resolve inconsistencies between stakeholder viewpoints.

In this paper, we have investigated how toleration of inconsistency, one of the main benefits of viewpoints-based modeling, can be achieved when multiple competing views are consolidated. We use lattice ordering to characterize variability degrees and evolutionary properties. We then annotate the features in individual viewpoints using the lattice ordering, and show how the composite viewpoint can be systematically generated. During viewpoints integration, some inconsistencies are tolerated, while others are resolved with rationales recorded. The integrated feature model supports stakeholder buy-in and traceability. The study of integrating a set of evolving mobile phone viewpoints shows that toleration of inconsistency effectively supports product line evolution and product derivation.

Our future work includes incorporating alternative features into current variability orderings and exploring new variability dimensions. We also plan to investigate structural merging [29] and edge-based variability types [33]. Moreover, it would be interesting to take into account feature dependencies, such as 'requires' and 'excludes', in viewpoints integration and product customization. Finally, addressing both stakeholder and rationale traceability is in order.

References

[1] D. Batory, Feature Models, Grammars, and Propositional Formulas, in: International Software Product Line Conference (SPLC'05), 2005, pp. 7–20.

[2] J. Bosch, Software Product Families in Nokia, in: International Software Product Line Conference (SPLC'05), 2005, pp. 2–6.

[3] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin, Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines, in: International Conference on Software Engineering (ICSE'10), 2010, (to appear).

[4] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[5] B. A. Davey, H. A. Priestley, Introduction to Lattices and Order *(2nd edition)*, Cambridge University Press, 2002.

[6] J.-M. DeBaud, TrueScope – A Full Life-Cycle Approach to Develop Software Product Lines, in: International Software Product Line Conference (SPLC'00), 2000, tutorial 6.

[7] Z. Diskin, Algebraic Models for Bidirectional Model Synchronization, in: International Conference on Model Driven Engineering Languages and Systems (MoDELS'08), 2008, pp. 21–36.

[8] S. Easterbrook, B. Nuseibeh, Using Viewpoints for Inconsistency Management, Software Engineering Journal 11 (1) (1996) 31–43.

[9] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, P. Schuster, Visualizing Software Changes, IEEE Transactions on Software Engineering 28 (4) (2002) 396–412.

[10] S. Ferber, J. Haag, J. Savolainen, Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line, in: International Software Product Line Conference (SPLC'02), 2005, pp. 235–256.

[11] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, Inconsistency Handling in Multiperspective Specifications, IEEE Transactions on Software Engineering 20 (8) (1994) 569–578.

[12] A. C. W. Finkelstein, I. Sommerville, The Viewpoints FAQ, Software Engineering Journal 11 (1) (1996) 2–4.

[13] W. Frakes, K. Kang, Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering 31 (7) (2005) 529–536.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (1990).

[15] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, Annals of Software Engineering 5 (1) (1998) 143–168.

[16] A. Lapouchnian, Y. Yu, J. Mylopoulos, Requirements-Driven Design and Configuration Management of Business Processes, in: International Conference on Business Process Management (BPM'07), 2007, pp. 246–261.

[17] K. Lauenroth, K. Pohl, Towards Automated Consistency Checks of Product Line Requirements, in: International Conference on Automated Software Engineering (ASE'07), 2007, pp. 373–376.

[18] K. Lauenroth, K. Pohl, Dynamic Consistency Checking of Domain Requirements in Product Line Engineering, in: International Requirements Engineering Conference (RE'08), 2008, pp. 193–202.

[19] S. Liaskos, A. Lapouchnian, Y. Yu, E. S. K. Yu, J. Mylopoulos, On Goal-based Variability Acquisition and Analysis, in: International Requirements Engineering Conference (RE'06), 2006, pp. 76–85.

[20] J. Liu, J. Dehlinger, R. Lutz, Safety Analysis of Software Product Lines Using State-Based Modeling, Journal of Systems and Software 80 (11) (2007) 1879–1892.

[21] M. Mannion, J. Savolainen, T. Asikainen, Viewpoint-Oriented Variability Modeling, in: International Computer Software and Applications Conference (COMPSAC'09), 2009, pp. 67–72.

[22] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave, Matching and Merging of Statecharts Specifications, in: International Conference on Software Engineering (ICSE'07), 2007, pp. 54–64.

[23] N. Niu, S. Easterbrook, So, You Think You Know Others' Goals? A Repertory Grid Study, IEEE Software 24 (2) (2007) 53–61.

[24] N. Niu, S. Easterbrook, Extracting and Modeling Product Line Functional Requirements, in: International Requirements Engineering Conference (RE'08), 2008, pp. 155–164.

[25] N. Niu, S. Easterbrook, M. Sabetzadeh, A Category-Theoretic Approach to Syntactic Software Merging, in: International Conference on Software Maintenance (ICSM'05), 2005, pp. 197–206.

[26] B. Nuseibeh, S. Easterbrook, A. Russo, Making Inconsistency Respectable in Software Development, Journal of Systems and Software 58 (2) (2001) 171–180.

[27] D. L. Parnas, Software Product-Lines: What To Do When Enumeration Won't Work, in: International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'07), 2007, pp. 9–14.

[28] M.-O. Reiser, M. Weber, Using Product Sets to Define Complex Product Decisions, in: International Software Product Line Conference (SPLC'05), 2005, pp. 21–32.

[29] M. Sabetzadeh, S. Easterbrook, View Merging in the Presence of Incompleteness and Inconsistency, Requirements Engineering Journal 11 (3) (2006) 174–193.

[30] J. Savolainen, J. Bosch, J. Kuusela, T. Männistö, Default Values for Improved Product Line Management, in: International Software Product Line Conference (SPLC'09), 2009, pp. 51–60.

[31] J. Savolainen, J. Kuusela, Consistency Management of Product Line Requirements, in: International Symposium on Requirements Engineering (RE'01), 2001, pp. 40–47.

[32] J. Savolainen, I. Oliver, M. Mannion, H. Zuo, Transitioning from Product Line Requirements to Product Line Architecture, in: International Computer Software and Applications Conference (COMPSAC'05), 2005, pp. 186–195.

[33] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Feature Diagrams: A Survey and a Formal Semantics, in: International Requirements Engineering Conference (RE'06), 2006, pp. 136–145.

[34] A. K. Thurimella, B. Bruegge, O. Creighton, Identifying and Exploiting the Similarities between Rationale Management and Variability Management, in: International Software Product Line Conference (SPLC'08), 2008, pp. 99–108.

[35] D. M. Weiss, C. T. R. Lai, Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999.

[36] Y. Yu, J. C. S. do Prado Leite, A. Lapouchnian, J. Mylopoulos, Configuring Features with Stakeholder Goals, in: ACM Symposium on Applied Computing (SAC'08), 2008, pp. 645–649.

[37] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, J. C. S. do Prado Leite, Reverse Engineering Goal Models from Legacy Code, in: International Requirements Engineering Conference (RE'05), 2005, pp. 363–372.