International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 670803, 24 pages
doi:10.5402/2012/670803

Review Article

Model-Driven Engineering for Software Product Lines

Jean-Marc Jézéquel

Institut de Recherche en Informatique et Systèmes Aléatoire (IRISA), University of Rennes 1, 35042 Rennes, France

Correspondence should be addressed to Jean-Marc Jézéquel, jezequel@irisa.fr

Received 24 September 2012; Accepted 14 October 2012

Academic Editors: J. Brooke and H. Okamura

Modeling variability in the context of software product-lines has been around for about 25 years in the research community. It started with Feature Modeling and soon enough was extended to handle many different concerns. Beyond being used for a mere description and documentation of variability, variability models are more and more leveraged to produce other artifacts, such as configurators, code, or test cases. This paper overviews several classification dimensions of variability modeling and explores how do they fit with such artifact production purposes.

1. Introduction

Software is nowadays a critical asset for many organizations: many aspects of our daily lives indeed depend on complex software-intensive systems, from banking and communications to transportation and medicine. Constant market evolution triggered an exponential growth in the complexity and variability of modern software solutions. Due to the increasing demand of highly customized products and services, software organizations now have to produce many complex variants accounting not only for differences in software functionalities but also for differences in hardware (e.g., graphic cards, display capacities, and input devices), operating systems, localization, user preferences, look and feel, and so forth. Of course, since they do not want to develop each variant from scratch and independently, they have a strong motivation to investigate new ways of reusing common parts to create new software systems from existing software assets.

Software Product Lines (SPL) [1], or *software product families* [2, 3], are emerging as a paradigm shift towards modeling and developing software system families rather than individual systems. SPL engineering embraces the ideas of mass customization and software reuse. It focuses on the means of efficiently producing and maintaining multiple related software products (such as cellular phones [4]), exploiting what they have in common and managing what varies among them [5].

Several definitions of the *software product line* concept can be found in the research literature. Northrop defines it as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way*" [6]. Bosch provides a different definition [7]: "*A SPL consists of a product line architecture and a set of reusable components designed for incorporation into the product line architecture. In addition, the PL consists of the software products developed using the mentioned reusable assets*". In spite of the similarities, these definitions provide different perspectives of the concept: *market-driven*, as seen by Northrop, and *technology-oriented* for Bosch.

SPL engineering is a process focusing on capturing the *commonalities* (assumptions true for each family member) and *variability* (assumptions about how individual family members differ) between several software products [8]. Instead of describing a single software system, a SPL model describes a set of products in the same domain. This is accomplished by distinguishing between elements common to all SPL members, and those that may vary from one product to another. Reuse of core assets, which form the basis of the product line, is key to productivity and quality gains. These core assets extend beyond simple code reuse and may include the architecture, software components, domain models, requirements statements, documentation, test plans, or test cases.

The SPL engineering process consists of two major steps.

(1) *Domain Engineering*, or *development for reuse*, focuses on core assets development.

(2) *Application Engineering*, or *development with reuse*, addresses the development of the final products using core assets and following customer requirements.

Figure 1 graphically represents the general SPL engineering process, as it can be found in the research literature [6]. As illustrated, the two phases are intertwined: application engineering consumes assets produced during domain engineering, while feedback from it facilitates the construction of new assets or improvement of existing ones.

Domain Engineering consists of collecting, organizing, and storing past experiences in building systems in the form of reusable assets and providing an adequate means for reusing them for building new systems [9]. It starts with a *domain analysis* phase to identify commonalities and variability among SPL members. During *domain design*, the product line architecture is defined in terms of software components and is implemented during the last phase.

Application Engineering, also known as *product derivation*, consists of building the actual systems from the core assets.

Central to both processes is the management of variability across the product line [3]. In common language use, the term *variability* refers to *the ability or the tendency to change*. Variability management is thus seen as the key feature that distinguishes SPL engineering from other software development approaches [10]. Variability management is thus growingly seen as the cornerstone of SPL development, covering the entire development life cycle, from requirements elicitation [11] to product derivation [12] to product testing [13, 14].

A traditional way used by scientists to master the increasing complexity and variability of real-world phenomena is to resort to *modeling*. Modeling is not just about expressing a solution at a higher abstraction level than code [15]. This limited view on modeling has been useful in the past (assembly languages abstracting away from machine code, 3GL abstracting over assembly languages, etc.) and it is still useful today to get, for example, a holistic view on a large C++ program. But modeling goes well beyond that. In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough. These models may be expressed with a general purpose modeling language such as the UML [16], or with Domain Specific Languages when it is more appropriate. Each of these models can be seen as the abstraction of an aspect of reality for handling a given concern. The provision of effective means for handling such concerns makes it possible to effectively manage variability in product-lines.

Modeling variability allows a company to capture and select which version of which variant of any particular aspect is wanted in the system [10]. To do it cheaply, quickly, and safely, redoing by hand the tedious weaving of every aspect is not an option; some form of automation is needed to leverage the modeling of variability [17, 18]. Model-Driven Engineering (MDE) makes it possible to automate this weaving process [19]. This requires that models are no longer informal, and that the weaving process is itself described as a program (which is as a matter of facts an executable metamodel [20]) manipulating these models to produce, for example, a detailed design that can ultimately be transformed to code, or to test suites [21], or other software artifacts.

MDE has started to be used by organizations to effectively manage software product lines. An entire SPL can be 2 and created from a single configurable model. Models can explicitly show both the common and varying parts of the SPL design. Using MDE technology, SPLs can be planned, specified, processed, and maintained on a higher abstraction level.

In recent years, several variability modeling techniques have been developed, aiming to explicitly and effectively represent SPL variability, and to leverage these models for a variety of engineering purposes. The purpose of this paper is to survey several classification dimensions of variability modeling, and explore how do they fit with other artifact production purposes.

The remainder of the paper is organized as follows. Section 2 gives an historical perspective on the emergence of variability modeling. In Section 3, we define several dimensions of variability modeling and then illustrate them with an overview of representative variability modeling methods. Note that this section does not have any goal of exhaustivity; for a systematic literature review, we refer the reader to [22]. Here we only subjectively select a set of approaches that we feel are representative of the possible ways of modeling variability in SPL. Going from contemplative to productive, in Section 4 we present some MDE tools leveraging variability models for a range of product line engineering activities. Section 5 concludes the paper and discusses further readings in the field.

2. The Emergence of Variability Modeling

2.1. Definitions of Variability. The basic vision underlying SPL can probably be traced back to Parnas seminal article [23] on the Design and Development of Program Families. Central to the SPL paradigm is the *modeling and management of variability*, the commonalities and differences in the applications in terms of requirements, architecture, components, and test artifacts [24]. Software variation management is often split into two dimensions [24].

Variability in time, referring to the existence of different versions of an artifact that are valid at different times;

Variability in space, referring to the existence of an artifact in different shapes at the same time.

Variability in time is primarily concerned with managing program variation over time and includes revision control system and the larger field of software configuration management [25]. The goal of SPL engineering is mainly to deal with variability in space [26, 27].

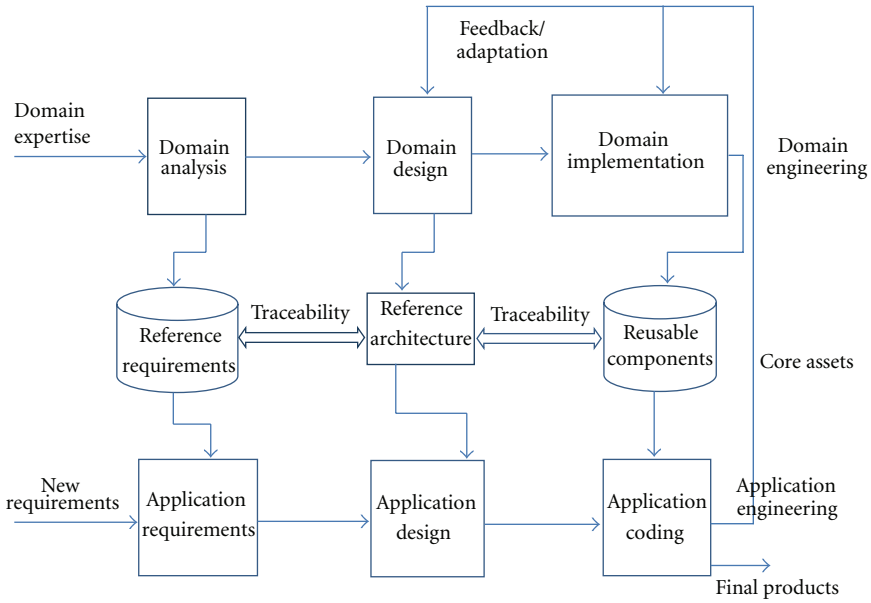

Figure 1: Product line engineering process [6].

Weiss and Lai [28] defined variability in SPL as "*an assumption about how members of a family may differ from each other*". From a software perspective [29], variability can be seen as the "*the ability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context*".

Variability has also been defined in many other different ways in a product line context. According to Bachmann and Clements [30] "*variability means the ability of a core asset to adapt to usages in different product contexts that are within the product line scope*". For Pohl et al. [24] it is the "*variability that is modeled to enable the development of customized applications by reusing predefined, adjustable artifacts*". A more goal-oriented definition of variability is also given by Bachmann and Clements [30] as a way to "*maximize the return on investment for building and maintaining products over a specified period of time or number of products*".

2.2. Classifications of Variability. Several possible classifications have been proposed. Halmans and Pohl [3] distinguish between *essential* and *technical* variability, especially at requirements level. Essential variability corresponds to the customer's viewpoint, defining what to implement, while technical variability relates to product family engineering, defining how to implement it. A classification based on the dimensions of variability is proposed by Pohl et al. [24]. Beyond variability in time and variability in space as discussed above, Pohl et al. claim that variability is important to different stakeholders and thus has different levels of visibility: external variability is visible to the customers while internal variability, that of domain artifacts, is hidden from them. Other classification proposals come from Meekel et al. [31] (feature, hardware platform, performances, and attributes variability) or Bachmann and Bass [32] who discuss about variability at the architectural level.

The management of variability can also be described through a process oriented point of view [29]:

(i) *identification of variability* determines where variability is needed in the product line (list the features that may vary between products),

(ii) *constraining variability* provides just enough flexibility for current and future system needs,

(iii) *implementing variability* selects a suitable variability realization technique based on the previously determined constraints,

(iv) *managing variability* requires constant feature maintenance and repopulation of variant features.

2.3. Modeling Variability. Central to the modeling of variability is the notion of *feature*, originally defined by Kang et al. as "*a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems*" [33]. Customers and engineers refer to product characteristics in terms of what features a product has or delivers, so it is natural to express any commonality and variability between products also in terms of features [34]. Hence Czarnecki and Eisenecker adapted this definition to the SPL domain as "*a system property relevant to some stakeholder used to capture commonalities or discriminate among systems in a family*" [35].

A feature can play different roles during the SPL engineering process. During domain engineering, they are *units of evolution* that adapt the system family to optional user requirements [36]. During application engineering, "*the product is defined by selecting a group of features, for which a carefully coordinated and complicated mixture of parts of different components are involved*" [37].

Based on this notion of *feature*, Kang et al. proposed to use a *feature model* [33] to model the variability in a SPL. A feature model consists of a *feature diagram* and

other associated information: *constraints* and *dependency rules*. Feature diagrams provide a *graphical tree-like notation depicting the hierarchical organization of high-level product functionalities* represented as features. The root of the tree refers to the complete system and is progressively decomposed into more refined features (tree nodes). Relations between nodes (features) are materialized by *decomposition edges* and *textual constraints*. Variability can be expressed in several ways. Presence or absence of a feature from a product is modeled using *mandatory* or *optional features*. Features are graphically represented as rectangles while some graphical elements (e.g., unfilled circle) are used to describe the variability (e.g., a feature may be optional).

Features can be organized into *feature groups*. Boolean operators *exclusive alternative (XOR)*, *inclusive alternative (OR)*, or *inclusive (AND)* are used to select one, several, or all the features from a feature group. Dependencies between features can be modeled using *textual constraints*: *requires* (presence of a feature imposes the presence of another), *mutex* (presence of a feature automatically excludes another).

For the last 25 years, there have been a lot of contributions from research and industry in this area. The initial proposal of Kang et al. was part of the Feature Oriented Domain Analysis (FODA) methodology [33]. Its main purpose was to capture commonalities and variabilities at requirements level. Feature Diagrams proved themselves very useful as a concise way to describe allowed variabilities between products of the same family, to represent feature dependencies, to guide feature selection as to allow the construction of specific products [38].

This notation has the advantage of being clear and easy to understand. However it lacks expressiveness to model relations between variants or to explicitly represent variation points. Consequently, several extensions were added to Kang et al.'s original notation, in particular for people wanting to extend Feature Diagrams beyond the requirement level.

2.4. Managing Variability into the Code. Several authors have also focused on proposing mechanisms to implement and manage variability especially at design or code level. Jacobson et al. [39] and Bachmann and Clements [30] propose to use mechanisms, like, inheritance, extensions and extension points, parameterization, templates and macros, configuration and module interconnection languages, generation of derived components, and compiler directives for this purpose.

For example, the contribution of [40] is to propose a method to reify the *variants* of an object-oriented software system into language-level objects; and to show that newly available compilation technology makes this proposal attractive with respect to performance (memory footprint and execution time) by inferring which classes are needed for a specific configuration and optimizing the generated code accordingly. This approach opens the possibility of leveraging the good modeling capabilities of object-oriented languages to deal with fully dynamic software configuration, while being able to produce space and time efficient executable when the program contains enough static configuration information.

Creational Design Patterns [41] are used to provide the necessary flexibility for describing and selecting relevant configurations within an object-oriented implementation, and thus benefitting from a better security implied by static typing, that is checked by the compiler. With this design framework, the actual configuration management can be programmed *within* the target language; it boils down to only create the class instances relevant to a given configuration [42]. However some care has to be taken for programming the creation of these objects to ensure that the design is flexible enough. In simple cases, an *Abstract Factory* is used to define an interface for creating variants. The factory features one *Factory Method* (encapsulating the procedure for creating an object) for each of the variability dimensions. The Factory Methods are parameterized to let them create various kinds of products (i.e., variants of a type), depending on the dynamic configuration selected at runtime. These Factory Methods are abstractly defined in the abstract factory, and given concrete implementations in its subclasses, called *concrete factories*.

In this approach, product derivation can be seen as an application of ideas circulating in the "Partial Evaluation" community for years. Actually, it can be seen as taking benefit of the fact that the type of configurable parts have bounded static variations (i.e., the sets of possible types are known at compile time). Thus the Partial Evaluation community trick known as *The Trick* (see [42]) can be applied to specialize the general program at compile time, and thus obtain specific products only embedding the code they really need.

More generally along these lines, Svahnberg et al. [29] present a taxonomy of different ways to implement variation points, which they refer to as "variability realization techniques".

2.5. Bridging the Gap between Requirement and Code: Feature Modeling Extensions. To bridge the gap between variability modeling at the level of requirements as originally found in Feature Models, and variability realization techniques as surveyed by Svahnberg et al. [29], many researchers proposed to extend Feature Models to encompass a wider spectrum of assets.

A first extension of FODA is the Feature Oriented Reuse Method (FORM) [43] developed by Kang et al. in 1998. It proposes a four-layer decomposition structure, corresponding to different stakeholder viewpoints. There are small differences in the notation compared to FODA: feature names appear in boxes, three new types of feature relations introduced (*composed-of*, *generalization/specialization*, *implemented-by*).

Griss et al. propose FeatuRSEB [44], a combination of FODA and the Reuse-Driven Software Engineering Business (RSEB) method. The novelties proposed are *introduction of UML-like notational constructs* for creating Feature Diagrams, *explicit representation of variation points, and variants* (white and black diamonds), explicit graphical representation for feature constraints and dependencies. Van Gurp et al. [45] slightly extend FeatuRSEB by introducing *binding times* (annotation to indicate when features can be selected) and *external features* (capabilities offered by the target platform).

Riebisch proposed to explicitly represent *cardinalities* in Feature Diagram and thus extends them with *UML multiplicities* [46]. *Group cardinalities* are introduced and denote the minimum and maximum number of features that can be selected from a feature group. There are two other changes: first a feature is allowed to have multiple parents and second, it is no longer features that are optional or mandatory, but edges.

Czarnecki and Eisenecker adapted Feature Diagrams in the context of Generative Programming [35] by adding an *OR feature decomposition* and defining a graphical representation of features dependencies. More recently, this notation was extended with new concepts: *staged configuration* (used for product derivation) and *group and feature cardinalities* [47].

Product Line Use Case modeling for System and Software engineering (PLUSS) [48] is another approach based on FeatuRSEB [44] that combines Feature Diagrams and Use Cases. The originality of this approach is that the decomposition operator is made explicit to compose feature: two new types of nodes are introduced: *single adapters* (represent XOR-decomposition) and *multiple adapters* (OR decomposition).

Many other extensions to feature modeling have been proposed to increase their expressiveness. Deursen and Klint in [49] defines an abstract syntax and semantics for Feature Diagrams. Batory [50] introduces propositional constraints defined between features. Ferber et al. [51] defines a separate view to represent feature dependencies and interactions.

Figure 2 from [52] provides a synthesis of the concepts used to capture variability and how they are graphically represented by the feature modeling languages described in this section. The figure shows what each feature modeling dialect is able to represent, as well as its limitations.

Despite their popularity and widespread use, all these Feature Models variants only provide a hierarchical structuring of high-level product functionalities [53], with very little connection with the actual software products. Since there is no indication of what the concrete representations of the features are, Feature Models only allow the SPL engineer to make a simple configuration of products through a *feature selection*. It is thus clear that more than a simple Feature Model is required for performing product derivation.

These limitations generated the need for other more expressive mechanisms for representing variability and linking it to the base assets.

3. An Overview on Representative Variability Modeling Methods

3.1. Introduction. Since SPLs revolve around the ideas of capturing commonalities and variations, a SPL can be fully modeled as

> *an assests model* that models a set of core assets, that is, reusable components used for the development of new products;
>
> *a variability model* that represent the commonality and variability between product line members.

Since standard languages are generally not developed to explicitly represent all types of variability, SPL models are frequently expressed by extending or annotating standard languages (models). The annotated models are unions of all specific models in a model family and contain all necessary variability concepts.

In MDE, the structure of a domain is explicitly captured in a *metamodel*. So it is clear that we need two different metamodels (i.e., two different sets of concepts) to handle both aspects of SPLs, but we also need to somehow *connect* these two sets of concepts to manipulate models of SPLs. Consequently, as noted by [54], there are two categories of techniques to handle variability: *amalgamated* and *separated*. The amalgamated approach proposes to connect the asset model and the variability model at metamodel level, that is, to augment the asset metamodel with variability concepts, while the separated approach proposes to connect them at model level, that is, the two modeling languages remain independent and the connection is made across model elements of either metamodels.

In turn, each of these two main styles of approaches decomposes into several threads that we are going to overview in the next subsections, taking into account issues such as addressing behavioral variability or handling variability at several stages in the software lifecycle (requirements time, design time, deployment time, and runtime).

Table 1 summarizes the classification proposed in [52]. It outlines what does happen at metamodel and model level for the identified classes and subclasses of variability modeling techniques.

3.2. Amalgamated Approach. Techniques using an amalgamated approach extend a language or a general purpose metamodel with specific concepts that allow designers to describe variability. Their core characteristic is the mix of variability and product line assets concepts into a unique model. Concepts regarding variability and those that describe the assets metamodel are combined into a new language, that may either have a new, mixed syntax, or one based on that of the base model extended by the syntax of the variability language. This applies at both metamodel and model level. We further distinguish 3 subcategories: ad hoc extensions to existing languages, generic extensions that can be woven into any language, and finally ad hoc languages.

3.2.1. Annotate a Base Model by Means of Extensions. Clauss [55, 56], first proposed to apply variability extensions to UML Class Diagrams, leveraging the UML of extension mechanism that allows designers to describe generic models. Clauss uses such generic models in which he explicitly defines variability at particular points called *hot spots*. The extensions proposed are based on the notions of *variation points* and *variants*: variation points help locate variability; each variant denotes a concrete way to realize that variability.

Variation points and variants are explicitly marked with the stereotypes $\langle\langle variation\,Point\rangle\rangle$, respectively $\langle\langle variant\rangle\rangle$, specified for the Generalizable Element UML metaclass. Therefore the variation point notion can be applied to

	FODA	FORM	FeatuRSEB	Van Gurp and Bosch	Riebish	Generative programming	PLUSS
Mandatory feature	F	F	F	F	F	F	F
Optional feature	F	F	F	F	F	F	F
And decomposition	F, F F	F, F F	F, F F	F, F F	F, F F	F, F F	F, F F
OR decomposition	—	—	F, F F	F, F F	F, 1..2, F F	F, F F	F, M M, F F
XOR decomposition	F, F F	F, F F	F, F F	F, F F	F, 1..1, F F	F, F F	F, S S, F F
Dependencies between features (Textual)	"requires" "exclude"	"requires" "exclude"	"requires" "exclude"	—	"requires" "exclude"	"requires" "exclude"	"requires" "exclude"
Dependencies between features (Graphical)	—	—	"requires" F → F "exclude" F → F	—	"requires" F → F "exclude" F → F	"requires" F → F "exclude" F → F	"requires" F → F "exclude" F → F
Explicit marking of variation points (VP) and variants (V)	—	—	F →VP, F F, V	—	—	—	—
Other special notational elements	—	Generalization/ specialization, Implemented by	—	F External feature, F Runtime, F F	F, * F F, Where * 0..1 1 0..*n* 1..*n* *p*..1 *p*..*n* 0..*	—	—

Figure 2: Feature Diagram Dialects—synthesis of variability modeling concepts.

Table 1: Classification of variability modeling techniques.

Technique name	Metamodel level		Model level	
Unique model (combined) for product line assets and PL variability				
Annotating the base model by means of extensions	AMM + V		PLM (conforms to AMM + V)	
Combine a general, reusable variability metamodel with base metamodels	AMM	VMM	PLM (conforms to (AMM o VMM))	
Separate (distinct) assets model and variability model				
Connect Feature Diagrams to model fragments	AMM	VMM	AM	VM (FDM)
Orthogonal Variability Modeling (OVM)	AMM	VMM	AM	VM (OVM)
ConIPF Variability Modeling Framework (COVAMOF)	AMM	VMM (CVV)	AM	VM (CVV)
Decision model based approaches	AMM	VMM (DMM)	AM	VM (DM)
Combine a common variability language with different base modeling languages	AMM	VMM (CVL)	AM	VM (CVL)

AMM: assets metamodel, AM: assets model, VMM: variability meta-model, VM: variability model, AMM + V: assets metamodel with variability, PLM: product line model, CVL: common variability language, FDM: feature diagram model, and DMM: decision metamodel, DM: decision model.

classes, components, packages, collaborations, and associations. The ⟨⟨*variation Point*⟩⟩ stereotype can be used together with several *tagged values* to specify the *binding time* (development, installation, or runtime) and the *multiplicity* of variants associated to a variation point. A variation point is connected to its variants through generalization/parameterization relations. It also has a unique name used to clearly associate it to its variants. Concerning variants, it is possible to capture dependencies between them using the ⟨⟨*mutex*⟩⟩ or ⟨⟨*require*⟩⟩ stereotypes. It is also possible to specify evolutionary constraints between elements using the ⟨⟨*evolution*⟩⟩ stereotype. Optional model elements can be identified with the ⟨⟨*optional*⟩⟩ stereotype. Presence conditions, similar with those used for variants, and tagged values for the binding time, can also be used with optional elements.

A second approach belonging to this category proposes to extend UML to specify product line variability for UML class and sequence diagrams [16, 57]. It defines a set of *stereotypes*, *tagged values*, and *structural constraints* and gather them in a UML profile for product lines [12].

Class diagrams are first extended with the concept of *optionality*. The ⟨⟨*optional*⟩⟩ stereotype marks model elements that can be omitted in some products. It is applied to the Classifier, Package, and Feature metaclasses from UML. As for the previous approach, the *variation point* concept is used. It is are modeled using UML inheritance and stereotypes: a variation point is defined by an abstract class and a set of subclasses which represent its variants. The abstract class is tagged with the ⟨⟨*variation*⟩⟩ stereotype while the subclasses with ⟨⟨*variant*⟩⟩. The UML profile also contains *constraints* which specify structural rules applicable to all models tagged with a specific stereotype.

For sequence diagrams, variability is introduced through three concepts: *optionality*, *variation,* and *virtuality*. The ⟨⟨*optional Lifeline*⟩⟩ and ⟨⟨*optional Interaction*⟩⟩ stereotypes identify optional objects and interactions in a sequence diagram. *Variation* refers to the existence of several possible interaction variants and is specified using the ⟨⟨*variation*⟩⟩ and ⟨⟨*variant*⟩⟩ stereotypes, which extend the Interaction metaclass. Finally, *virtuality* means that parts of a sequence diagram can be redefined for individual products by another sequence diagram. The ⟨⟨*virtual*⟩⟩ stereotype is used for this.

The work of Gomaa and Shin [58, 59] on multiple view product line modeling using UML also falls into this category. It promotes the use of different perspectives for a better understanding of the product line. The used views are *use case model view* for functional SPL requirements; *static model view* for static structural SPL aspects; *collaboration model view* to capture the sequence of messages passed between objects; *state chart model view* to address dynamic SPL aspects.

A multiple-view model is modified at specific locations, different for each view. To represent SPL variability in the Use Case model view, Use Cases are stereotyped as either *kernel*, *optional* or *variant*, while *extend* and *include* relations allow a Use Case to extend or include another Use Case at a variation point. In the class diagrams, *abstract classes* and *hot spots* provide ways for variation points. For the collaboration and state chart models, concepts from single-system development such as *alternative branches*, *message sequences,* and *state transitions* are used. Of course, when a view is modified at a variation point, the other views also need to be modified in order to maintain consistency.

Initially defined for Use Cases, the stereotypes mentioned above were also applied to other views. For the static model view, additional stereotypes are introduced: ⟨⟨*control*⟩⟩ (provide overall coordination), ⟨⟨*algorithm*⟩⟩ (details of application logic), ⟨⟨*entity*⟩⟩ (encapsulate data), and ⟨⟨*interface*⟩⟩ (interface to external environment). Variation is also expressed using classical UML concepts like abstract classes and inheritance.

Finally, de Oliveira Jr. et al. [60] present a UML-based process for variability management that allows the identification, representation, and delimitation of variabilities. In the variability identification and definition phases of the process, a series of stereotypes can be used to explicitly identify variation points, mark mandatory and optional elements, and identify exclusive or inclusive variants of a variation point. Dependency relations between variants are also supported. These extensions are applied to UML class and Use Case diagrams. For each stereotype, the set of UML relations on which it can be applied is provided.

In addition to stereotypes, *UML notes* are used to support variability representation. For a variation point, they define the *type of relationship with its variants* ({} for optional, {*or*} and {*xor*} for alternative and exclusive alternative); its *name* and *multiplicity* (minimum number of variants to be chosen for it); the *binding time* (design, implementation, compiling, linking or runtime); whether or not it supports new variants to be added. This information is established during the variability delimitation process.

Table 2 (from [52]) recalls the stereotypes and extensions introduced by each method discussed above, while Table 3 presents which type of UML diagrams are supported.

3.2.2. Combine a General, Reusable Variability Metamodel with Different Domain Metamodels. In the previous approaches, authors extended the UML metamodel for modeling variability in multiple UML diagrams like Class or Sequence diagrams. Morin et al. [61, 62] propose a more generic solution that can be applied to any kind of metamodel and that is fully supported by a tool. They propose a reusable variability metamodel describing variability concepts and their relations independently from any domain metamodel. Using Aspect-Oriented Modeling (AOM) techniques, variability can be woven into a given base metamodel, allowing its integration into a wide variety of metamodels in a semiautomatic way.

A key point of this method is the definition of a general variability metamodel, based on the work of Schobbens et al. [36, 38] on feature modeling. The abstract syntax proposed in [38] serves as the basis for the variability metamodel defined by Morin et al. [61]. In this meta-model, the central metaclass *PointOfVariability* can be woven with any base metamodel element on which variants are needed. *VariabilityOfElement* is a subclass of the *PointOfVariability* metaclass that allows actual domain concepts to vary.

Table 2: Annotating UML with stereotypes—synthesis.

Variability	Clauss	Ziadi and Jézéquel	Gomaa	Oliveira et al.
Variation point	⟨⟨*variation Point*⟩⟩	⟨⟨*variation*⟩⟩		⟨⟨*variation Point*⟩⟩
Variant	⟨⟨*variant*⟩⟩	⟨⟨*variant*⟩⟩	⟨⟨*variant*⟩⟩	⟨⟨*alternative—OR*⟩⟩ ⟨⟨*alternative—XOR*⟩⟩
Dependencies	⟨⟨*mutex*⟩⟩ ⟨⟨*require*⟩⟩			⟨⟨*mutex*⟩⟩ ⟨⟨*require*⟩⟩
Optional elements	⟨⟨*optional*⟩⟩	⟨⟨*optional*⟩⟩ ⟨⟨*optionalInteraction*⟩⟩ ⟨⟨*optionalLifeline*⟩⟩	⟨⟨*optional*⟩⟩	⟨⟨*optional*⟩⟩
Mandatory elements			⟨⟨*kernel*⟩⟩	⟨⟨*mandatory*⟩⟩
Other concepts	⟨⟨*evolution*⟩⟩	⟨⟨*virtual*⟩⟩ Generic/specific constraints	⟨⟨*control*⟩⟩ ⟨⟨*algorithm*⟩⟩ ⟨⟨*entity*⟩⟩ ⟨⟨*interface*⟩⟩	UML notes

Table 3: Annotating UML with stereotypes—supported diagrams.

Method name	Class	Use cases	Sequence	State charts
Clauss	Yes	No	No	No
Ziadi and Jézéquel	Yes	No	Yes	No
Gomaa	Yes	Yes	Yes	Yes
Oliveira et al.	Yes	Yes	No	No

Boolean operators inspired from feature diagrams are used to actually represent variability: *and*, *xor*, *or*, *opt*. The cardinality operator *Vp(i,j)* provides a greater degree of flexibility. Operators can either be *homogeneous* (apply only to elements of the same type) or *heterogeneous* (apply to elements of different types).

The process of creating the new metamodel that integrates concepts from both variability and base metamodels is easy: new metaclasses are created to connect the base metamodel with the variability aspect. The base metamodel is just extended, none of its elements are removed. This allows an easy translation of models encoded in the variability-woven metamodel into the original one and the reuse of already developed tools such as model editors or checkers. Once the weaving of variability is done to obtain an extended metamodel, product line models can be created. These are models with variability, conforming to the variability extended metamodel.

3.2.3. Ad Hoc Language Supporting Variability: Clafer. Contrary to previous approaches, Clafer [63] (Class, Feature, and Reference) is a standalone, lightweight modeling language with first-class support for feature modeling. More precisely, the language integrates feature modeling (i.e., a formalism to model variability, see below) into class modeling, so that variability can be naturally expressed in class models.

Clafer has a minimalistic syntax and semantics (very few underlying concepts), trying to unify existing notations (such as feature, class, and metamodels), both syntactically and semantically. It supports the most common constructions from domain modeling, modeling requirements, and structural modeling. Clafer models are expressive, yet analyzable with state-of-the-art tools, such as SAT/SMT-solvers and Alloy Analyzer. Currently, Clafer relies on Alloy (which uses SAT-solvers) to do analyses.

3.3. Separated Approaches. Techniques in this category have separate representations for the variability and for the assets model. Elements from the variability model relate to asset model elements by referencing them one way or another. The key characteristic of such methods is the clear separation of concerns they provide. This separation applies at both metamodel and model level, with the following advantages: each asset model may have more than one variability model; designers can focus on the product line itself and not on its variability, which is addressed separately. It also opens the way for a standardized Common Variability Language (CVL) as discussed below.

Istoan [52] further identifies three subcategories of methods which share the same principle but differ in the type of variability model they use.

3.3.1. Connect Feature Diagrams to Model Fragments. Since Feature Diagrams only concentrate on a specific aspect of SPL modeling, there is a need to combine them with other product representations, that is, to associate model fragments to features. Various model fragment types can be associated to features. The feature diagram defines the product line variability, with each feature having an associated implementation. Concerning our classification, we make a clear distinction between assets and variability related concepts at metamodel level. This situation extends to

model level: separate assets and variability models do coexist. For this category of methods, the assets model typically consists of a set of software artefact/asset fragments, while the variability model is a Feature Diagram.

Perrouin et al. [64] address specific and unforeseen customer requirements in product derivation by combining automated model composition with flexible product derivation approaches [65]. Their contribution are two metamodels defined to support the approach: a *generic feature metamodel* that supports a wide variety of existing FD dialects (used in the preconfiguration process), and a subset of UML used to define the assets metamodel (transformed during the customization step).

Their generic feature metamodel leverages Schobbens et al.'s pivot abstract syntax [36] that subsumes many of the existent FD dialects. In their proposal, *Feature Diagram* is the central metaclass and contains a list of features (*Feature* metaclass), with is a special one that is considered as the *root node*. Variability is represented using boolean operators. All classical feature diagram operators are provided: *or, and, xor, opt,* and *card*. They are subtypes of the abstract *Operator* metaclass. *Decomposition edges* represent relations between features. Feature dependencies like *mutex* or *require* can also be represented.

In the feature diagram metamodel, the *Feature* metaclass is connected using a composite association to a class called *Model* that defines the core assets involved in feature realization. This relation specifies that a feature may be implemented by several model fragments. Initially exploited with class diagrams [65], the metamodel allows any kind of assets to be associated with features.

Czarnecki and Antkiewicz [66] proposes a *general template-based* approach for mapping feature models to concrete representations using structural or behavioural models. They use a model representing a superimposition of all variants, whose elements relate to corresponding features through annotations. The approach is general and works for any model whose metamodel is expressed in MOF.

The idea promoted by Czarnecki and Antkiewicz is to separate the representation of a model family (product line model) into a *feature model* (defines feature hierarchies, constraints, possible configurations) and a *model template* (contains the union of model elements in all valid template instances). Elements of a model template can be annotated. These annotations are defined in terms of features from the feature model, and can be evaluated according to a particular feature configuration. Possible annotations are *presence conditions* (PCs) and *metaexpressions* (MEs). PCs are attached to a model element to indicate whether it should be present in a template instance. Typical PCs are boolean formulas over a set of variables, each variable corresponding to a feature from the FD. Complex PCs can be expressed using XPath. MEs are used to compute attributes of model elements. When a PC is not explicitly assigned to an element of a model template, an implicit presence condition (IPC) is assumed. IPCs reduce the necessary annotation effort for the user. Czarnecki and Antkiewicz define a set of choices of IPCs that can be used for UML class and activity diagrams, based on the element's type.

General guidelines for applying this method for a particular target notation are provided. They require to decide first the form of PCs and MEs, attach IPCs to model elements not explicitly annotated, decide on the type of the annotation mechanism used (e.g., UML stereotypes), and on how to render the annotations (labels, icons, or colouring).

There exist other methods belonging to this category, which we briefly mention below. Laguna and González-Baixauli [67] separate SPL variability aspects using goal models and UML diagrams, while keeping features at the core of the representation. They combine previous approaches with the *UML package merge* implementation to provide a set of mapping rules from features to class diagram fragments. Apel et al. [68] introduce *superimposition* as a technique to merge code fragments belonging to different features. They extend the approach and analyse whether UML class, state, and sequence diagrams can be decomposed into features and then recomposed using superimposition to create complete models corresponding to SPL products.

3.3.2. Orthogonal Variability Modeling. With Orthogonal Variability Modeling, the assets model and the variability model are kept separate. The variability model relates to different parts of the assets model using *artifact dependencies*. The differentiating factor from the previous category is the type of variability model used: an orthogonal variability model (OVM). There is also a difference regarding the assets model which is a compact software development artifact and no longer a set of model fragments.

Pohl et al. [24] propose the OVM concept, that is, later refined in [69]; a model that defines the variability of a SPL separately and then relates it to other development artifacts like Use Cases, component, and test models. OVM provides a view on variability across all development artifacts.

The central concepts used in OVM are *variation points (VP)* and *variants (V)*. A VP documents a variable item and a V its possible instances. Both VPs and Vs can be either *optional* or *mandatory*. Optional variants of the same VP are grouped together by an alternative choice. An optional variant may be part of at most one alternative group. To determine how many Vs may be chosen in an alternative choice, the *cardinality notation [min..max]* is used. OVM also supports the documentation of Vs belonging to different VPs. Simple constraints between nodes (mutex or require) can be graphically represented and can be applied to relations between Vs, but also to VP-V and VP-VP relations.

Modeling VPs, Vs and how they are connected is just a first step of the OVM process. The variability model can be related to software artifacts specified by other models. Pohl et al. document these relations using traceability links between the variability model and the other development artifacts. A special type of relationship called *artifact dependency* [24] which relates a V or a VP to a development artifact serves this purpose. A synthesis of OVM concepts together with their graphical representation is shown in Figure 3.

These concepts are captured and formalized slightly differently in [69]; while Pohl et al. [24] group them into a metamodel which defines what a well-formed OVM diagram is, Metzger et al. [69] formalize OVM's abstract syntax using

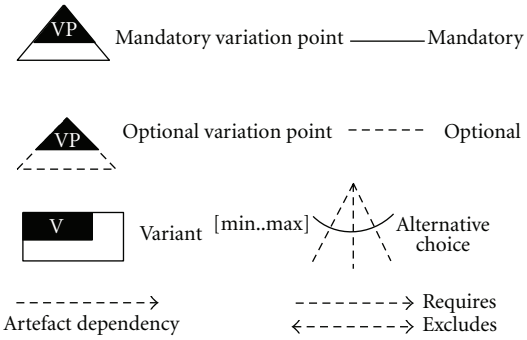

Figure 3: OVM method: synthesis of concepts.

a mathematical notation: an OVM is defined as a tuple of the form *(VP, V, VG, Parent, Min, Max, Opt, Req, and Excl)*.

OVM is a general approach that can be used to document variability in several software artifacts. Requirements variability is handled by relating the OVM to textual requirements or Use Cases. Architectural variability is documented in the development view by relating OVM to component, class, or object diagrams; in the process view by connecting OVM models to state machines, activity, or sequence diagrams; and in the code view by relating OVM to deployment diagrams.

3.3.3. ConIPF Variability Modeling Framework (COVAMOF). The COVAMOF method [70, 71] is yet another orthogonal variability method that differs in the type of variability model, that is, used. Sinnema et al. [70] identify four requirements which they considered essential for a variability modeling technique, and that they wanted to support in COVAMOF:

(1) uniform and first class representation of variation points at all abstraction levels;

(2) hierarchical organization of variability representation;

(3) first-class representation of dependencies, even complex ones;

(4) explicit modeling of interactions between dependencies.

COVAMOF was hence designed to uniformly model variability in all abstraction layers of a SPL. The COVAMOF framework addresses variability in a dedicated view called COVAMOF Variability View (CVV). Variability is represented as *variation points* and *dependencies* and provides means to explicitly model simple and complex relations between dependencies.

Variation points in the CVV reflect the variation points of the product family and are associated with product family artifacts. Five types of variation points are defined in CVV: *optional*, *alternative*, *optional variant*, *variant*, and *value*. A variation point also contains a *description*, some information about its *state*, the *rationale* of the binding, the *realization mechanism,* and the associated *binding time*. The *state* can be either *opened* (new variants can be added) or *closed* (not possible to add new variants). If a variation point does not have a realization mechanism associated in the PL artifacts, it will be realized by a variation point on a lower level. This is done using a *realization relation* which defines how variation points in one abstraction layer realize variation points in a higher layer.

Dependencies are associated with one or more variation points in the CVV and are used to restrict the selection of associated *variants*. A dependency has several properties: *type, description, validation time, and type of associations to variation points*. These associations to variation points are classified according to the impact that variant selection has on the validity of the dependency: *predictable* (impact can be determined before the actual variant binding), *directional* (specifies whether variants selection should have positive or negatively effect), and *unknown*. Three different types of dependencies are possible: *logical*, *numerical,* and *nominal*, which express the validity of the dependency in different ways. CVV also explicitly defines *dependency interactions* which specify how two or more dependencies mutually interact.

The main entities of the CVV metamodel are presented in Figure 4, while Figure 5 (from [52]) summarize the main concepts introduced by COVAMOF and their graphical representation.

3.3.4. Decision Model Based Approaches. This class of approaches keeps the same general characteristics as all other in this category. They differ in using *decision models* as variability model. Decision-oriented approaches were designed to guide the product derivation process based on *decision models*. Research literature offers several definitions of the term. Weiss and Lai [28] define it as "*the document defining the decisions that must be made to specify a member of a domain*". For Bayer et al. [72] it is a model that "*captures variability in a product line in terms of open decisions and possible resolutions*". A decision model is basically a table where each row represents a decision and each column a property of a decision.

Decision modeling in SPL was initially introduced as a part of the Synthesis Project by Campbell et al. [73]. Decisions were defined as "*actions which can be taken by application engineers to resolve the variations for a work product of a system*". Decision-oriented approaches treat *decisions* as first-class citizens for modeling variability. They describe the variation points in a PL and define the set of choices available at a certain point in time when deriving a product.

A representative approach in this category is DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) from Dhungana et al. [74]. It aims at supporting the modeling of variability for industrial SPL with a focus on automating the derivation of customer-specific products. It is a flexible and extensible decision-oriented variability modeling language. DOPLER was designed to support the modeling of both problem space variability (stakeholder needs) using decision models, and solution space variability (architecture and components of technical solution) using asset models and also to assure traceability between them.

A core concept used by DOPLER is the *decision*. It is specified by a *unique name* and has a *type*, which defines the

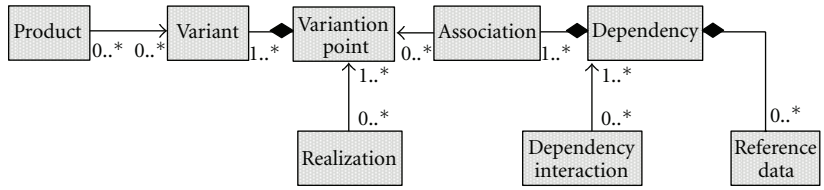

Figure 4: The COVAMOF meta-model.

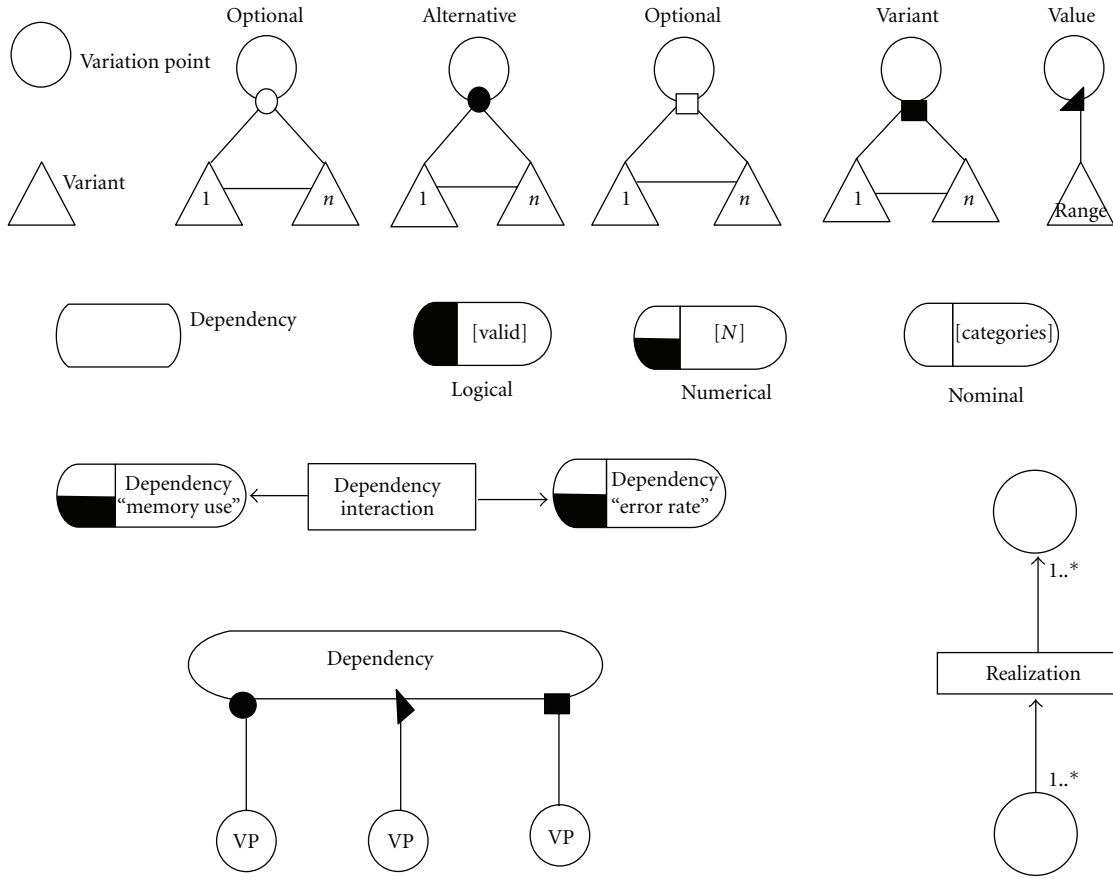

Figure 5: Summary of COVAMOF concepts.

range of values which can be assigned to a decision. Available decision types in DOPLER are *boolean, string, number, and enumeration*. Decisions can be annotated using *decision attributes* to capture further information (description, question) for the modeler or user. As the value range determined by the basic decision type is often too broad, *validity conditions* can be used to restrict it. *Visibility conditions* specify when a particular decision becomes relevant to the user and thus define hierarchical dependencies between decisions. Decisions are usually not independent of each other and cannot be made in isolation. Dependencies between them can be specified using *decision effects*. In DOPLER decisions are directly connected to *assets*, which represent the available artifacts of the product line. A collection of assets is defined in an *asset model*. Assets can have a *type* (defined for specific domains) and several *attributes*. Relations between assets are specified using *asset dependencies*. Assets are linked to decisions using *inclusion conditions* which describe the context in which a particular asset is required in a particular product. One inclusion condition can refer to multiple decisions.

These concepts are gathered and formalized by Dhungana et al. in a specific metamodel (see Figure 6). It is generic and can be adapted to different domains by defining concrete asset types, asset type attributes, and relationships between assets.

Along the same lines as DOPLER, Xabier Mansell and Sellier [75] propose a decision modeling process based on which the European Software Engineering Institute Spain and IKV++ Technologies AG Germany developed the VManage method [76]. It offers an XML-based solution to formally specify a PL decision model and use it for automatic product derivation. Each decision from the decision model has a set of properties: *name, description, type, default value, validity, and dependencies*. The different types of decisions are specified using an XML schema. There are two possible kinds of decisions: *unrestricted* and *restricted*. Furthermore,

VManage supports *collections of decisions* (instances of a decision or set of decisions). A metamodel that supports the approach and defines the general elements that form a decision model is available in [75].

Another proposal comes from Schmid and John [77] and is an extension of the original Synthesis approach. It adds binding times, set-typed relations, selector types, mapping selector types to specific notations, using multiplicity to allow the selection of subsets of possible resolutions, clear separation of constraints on the presence and value of decisions. The KobrA approach [78] integrates product line engineering and component-based software design. KobrA decision models are described using a tabular notation. A clear distinction is made between simple and high level decisions. For simple decision, references are given to the involved assets and variation points. In the case of high-level decisions, references are given to the other decisions that are affected by resolving it. The decision model thus forms a hierarchy.

3.3.5. Combine a Common Variability Language with Different Base Languages. Methods in this category propose a generic language or model that subsumes variability related concepts. The same general variability model can be combined with different base models, extending them with variability. Regarding our classification, at metamodel level there is a separate generic variability metamodel and an assets metamodel (AMM). The AMM is actually the metamodel of the base language on which the *common variability language* is applied. At model level, variability model elements relate to assets model elements by referencing and using substitutions.

We discuss in more detail the *Common Variability Language* (CVL) as proposed for standardization at the OMG. It is based on several previous works, notably by Haugen et al. [54, 79]. CVL models specify both variabilities and their resolution. By executing a CVL model, a base SPL model is transformed into a specific product model as illustrated in Figure 7.

The Variability Model and the Resolution Models are defined in CVL while the Base Model and Resolved Models can be defined in any MOF-defined language (see Figure 8).

> *The Base Model* represents an instance of an arbitrary MOF metamodel, such as UML, on which variability is specified using CVL. From the standpoint of CVL the base model is just a collection of objects and links between them.
>
> *The Foundation Layer* comprises means to define abstract variability with proper constraints, how to resolve the variability to define products, and how to realize the products to produce products defined in the base language.
>
> *The Compositional Layer* on top of the foundation layer includes ways to combine models in the foundation layer such that variability definitions can be reused, resolutions may have cascading effects, and several different base models (defined in different base languages) can be described. The configurable unit module provides the constructs that are needed for the modularization and encapsulation of variability as configurable units, that is, component-like structures that may be configured through an exposed variability interface.

Let us now detail the Foundation Layer, which is made of the *variability realization model*, the *variability abstraction model*, and *Constraints* and *Resolutions*, as shown in Figure 8.

Variability Realization. The *variability realization model* provides constructs for specifying variation points on the base model. A variation point is an item that defines one step in the process of how the base model is modified to reach the specified product. This module is the part of CVL that impacts the base model. The variation points refer to base model elements via base model handles.

The realization layer makes it possible to derive the products from the CVL description by transforming a base model in some MOF defined language to another product model in that same language. Every construct of the realization layer defines a Variation Point of the base model representing a small modification of the base model into the product model. There are several kinds of variation points.

(i) *Existence* is an indication that the existence of a particular object or link in the base model is in question.

(ii) *Substitution* is an indication that a single object, an entire model fragment, or the object at the end of a link, may be substituted for another. Object substitution involves two objects and means redirecting all links in which one is involved to the other and then deleting the former. Fragment substitution involves identifying a placement fragment in the base model via boundary element, thereby creating a conceptual "hole" to be filled by a replacement fragment of a compatible type.

(iii) *Value assignment* is an indication that a value may be assigned to a particular slot of some base model object.

(iv) *Opaque variation point* is an indication that a domain specific (user defined) variability is associated with the object(s) where the semantic of domain specific variability is specified explicitly using a suitable transformation language.

Variability Abstraction. The variability abstraction module provides constructs for specifying and resolving variability in an abstract level, that is, without specifying the exact nature of the variability w.r.t. the base model. It isolates the logical component of CVL from the parts that manipulate the base model. The central concept in this module is that of a variability specification (abbreviated as VSpec), which is an indication of variability in the base model. VSpec are similar to features in feature modeling, to the point that the concrete syntax of the variability abstraction is similar to a feature diagram where the variability specifications are shown as trees.

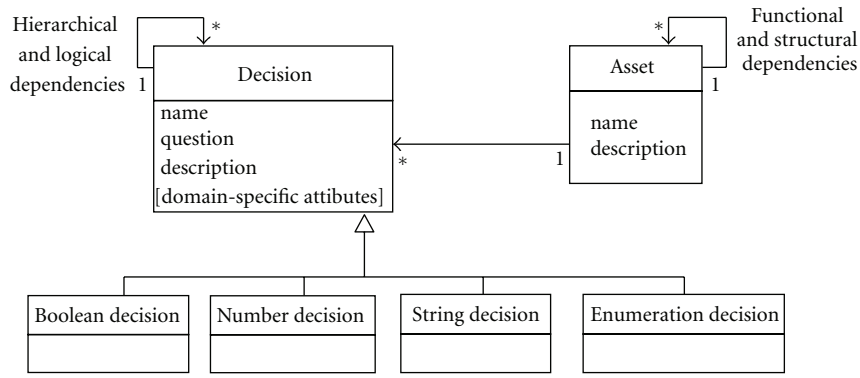

Figure 6: DOLPER meta-model.

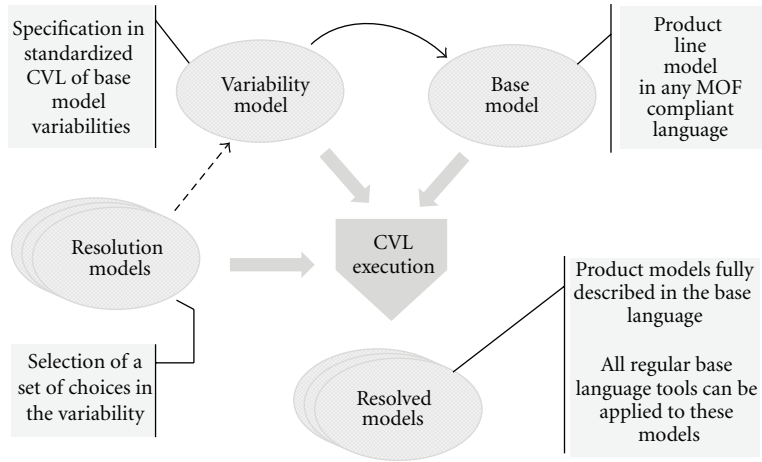

Figure 7: Using CVL.

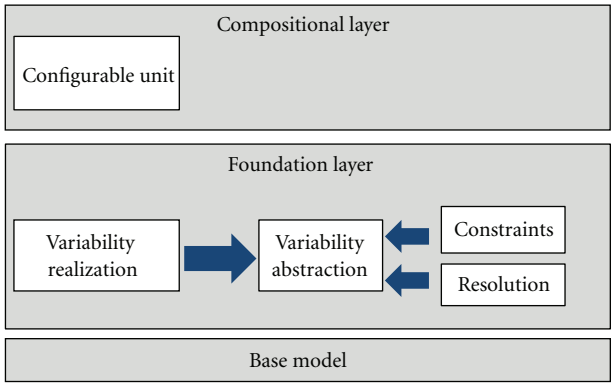

Figure 8: The Structure of CVL.

The specifics of the variability, that is, what base model elements are involved and how they are affected, is not specified, which is what makes VSpecs abstract. The effect on the base model may be indicated by binding VSpecs to variation points which refer to the base model. VSpecs may be arranged as trees, where the parent-child relationship organizes the resolution space by imposing structure and logic on permissible resolutions.

There are three kinds of VSpecs provided in the base layer: choices, variables, and variability classifiers.

(1) A choice is a VSpec whose resolution requires a yes/no decision. Nothing is known about the nature of the choice in the level of a VSpec tree, beyond what is suggested by its name. For example, the fact that there is a choice X in the tree indicates that in the resolution process there will be a need to decide *yes* or *no* about X, and that this decision may have some effect on the base model, the nature of which is unknown. It could decide for instance whether or not a given element will be deleted, a given substitution will be performed, a link will be redirected, and so forth.

(2) A variable is a kind of VSpec whose resolution involves providing a value of a specified type. This value is meant to be used in the base model, but similar to choices, it is unknown in this level exactly where and how.

(3) A variability classifier (abbreviated as VClassifier) is a kind of VSpec whose resolution means creating instances and then providing per-instance resolutions for the VSpecs in its subtree. Like choices and variables, it is unknown at this level what the effect of each instance will be. Each VClassifier has an instance multiplicity which indicates how many instances

of it may be created under its parent in permissible resolutions.

VSpecs are organized in a Tree Structure. The subtree under a node represents subordinate VSpecs in the sense that the resolution of a node imposes certain constraints on the resolutions of the nodes in its subtree:

(i) A negative resolution implies a negative resolution for its subchoices and no resolutions for all other child VSpecs.

(ii) Each choice has a field *isImpliedByParent* which, when True, indicates that if its parent is resolved positively then it must be decided positively. A resolution for a nonchoice VSpec is always considered positive for this definition. The general rule is as follows: if a parent is resolved positively, that is, it is either a positive choice decision or any variable resolution or any instance, then its subchoices with *isImpliedByParent* = *True* must be resolved positively, its subvariables must be resolved, that is, given a value, and its subclassifiers must be instantiated according to their instance multiplicity.

Each VSpec may also have a group multiplicity indicating how many total positive resolutions there may be under it in case it is resolved positively, where positive resolution means the same as above, that is, a positive choice decision or any variable value assignment or any instance of a VClassifier.

Constraints. Additional constraints can be used to express intricate relationships between VSpecs that cannot be directly captured by hierarchical relations in a VSpec tree. To this end CVL introduces a basic constraint language, a restricted subset of The Object Constraint Language (OCL), that is, amenable to formal processing and practical constraint solving.

Resolutions. VSpecs are resolved by VSpec resolutions, thus three kinds of VSpec resolutions mirror the three kinds of VSpecs. Choice resolutions resolve choices, variable value assignments resolve variables, and VInstances resolve VClassifiers. Each VSPpec resolution resolves exactly one VSpec of the appropriate kind. In the absence of classifiers each VSpec is resolved by at most one VSpec resolution.

Compositional Layer. The abstraction and realization modules of the foundation layer provide constructs for specifying logically organized variation points on a base model but do not provide means for grouping such specifications into units configurable as wholes. Base models on which variability will be specified with CVL may exhibit complex structures of modularization, composition, and encapsulation. For example a UML design for a real system will typically contain many packages, components, and classes organized in hierarchies, possibly deep ones. For scalability purposes, CVL must therefore itself accommodate such structures so that product line designs, that is, base models plus CVL variability defined against them, may continue to exhibit the same structures supported by the base models. Variability Encapsulation is helpful to the following.

(i) Accommodate the modular specification of large, complex systems with variability.

(ii) Accommodate the construction of libraries of reusable assets (components with variability in them) which are used in multiple projects of a given organization after configuring their variability. In this mode of work the system overall architecture does not contain variability but is constructed from configured components taken from a pool of reusable assets with variability.

(iii) Accommodate configuration dependencies between units over base models of different domains, that is, different metamodels. For example, the configuration of a unit of requirements may trigger the configuration of a UML unit designed to fulfill those requirements.

4. Exploitation of Variability Models

4.1. Using MDE to Process Variability Models. Models have been used for long as *descriptive* artifacts, and proved themselves very helpful for formalizing, sharing, and communicating ideas. Modeling variability in SPL is thus already very useful by itself, as highlighted by the popularity of feature modeling languages and their supporting tools (Pure::Variants [80], RequiLine, Gears [81], etc.).

In many cases we can however go beyond that, that is, we want to be able to perform computations on Variability Models, for a variety of purposes, such as validation of the consistency of models, automatic composition or decomposition of variability models, production of new artifacts (e.g., tests), and of course concrete product derivation. These usages of variability models require that they are no longer informal, and that the language used to describe them has a well-defined abstract syntax (i.e., metamodel) and semantics, as it is the case for the Variability Modeling Languages surveyed in the previous section.

From the methods that extend UML, those of Ziadi and Jézéquel [12], Gomaa and Shin [59], and de Oliveira Jr. et al. [60] provide formalizations of their approaches in the form of metamodel extensions. Conversely, Pohl et al. [24] use an explicit ad hoc metamodel, as well as Sinnema et al. who regroup the concepts used by the COVAMOF framework in a well defined metamodel described in [70, 71]. The concepts introduced in DOPLER are also gathered in a specific metamodel detailed in [74]. Morin et al. [62] also propose an explicit variability metamodel, to be woven into other metamodels as discussed before.

Once variability is actually modeled, based on a well-defined metamodel, standard Language Engineering tools can be leveraged. This tools fall into two categories:

(i) endomorphic tools, processing variability models on their own, for either validation (self consistency) or composition/decomposition

(ii) exomorphic tools, generating other artifacts from variability models, such as concrete software products or test cases.

On the concrete side, one can rely on, for example, well-tooled Eclipse standards such as E-MOF to describe these metamodels, and then readily benefit from a set of tools such as reflexive editors, or XML serialization of models, and also from a standard way of accessing models from Java, that is used in for example, [54].

This toolset can easily be complemented with operational semantics tools such as Kermeta [15, 20], a Kernel Metamodeling language and environment, to obtain a complete environment for such Variability Modeling languages, including checkers, interpreters, compilers, and so forth. Kermeta has indeed been designed to easily extend metamodels with many different concerns (such as static semantics, dynamic semantics, model transformations, connection to concrete syntax, etc.) expressed in heterogeneous languages, using an aspect-oriented paradigm. Kermeta is used for example, to support Perrouin et al.'s approach [64], to support product derivation in [12], and to weave variability into metamodels [62].

Another example is *fmp2rsm*, a tool supporting Czarnecki and Antkiewicz template based approach [66]. It is delivered as an Eclipse plug-in integrating the Feature Modeling Plug-in (FMP) with Rational Software Modeler (RSM), a UML modeling tool from IBM.

The use of such an MDE environment thus makes it quite straightforward to build a wide range of tools able to process Variability Models in several different ways that are described in the following sections.

4.2. Automated Analysis of Feature Models. Feature models may have interesting properties that can be automatically extracted by automated techniques and reported to an SPL engineer [82]. In particular, a feature model might represent no valid configuration, typically due to the presence of incompatible cross-tree constraints, or a feature model might have *dead* features, that is, features not present in any valid configuration.

The automatic analysis of feature models is thus an active area of research that is concerned with extracting information from feature models using automated mechanisms. Since the introduction of feature models, the literature has contributed with a number of algorithms to support the analysis process. Mannion [83] was the first to identify the use of propositional logic techniques to reason about properties of a feature model. Several other proposals [84–88] have been made to formalize Feature Models, but the most complete proposal, called Free Feature Diagrams, probably comes from Schobbens et al. [36].

The following steps are typically performed to encode a feature model as a propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature:

(1) each feature of the feature model corresponds to a variable of the propositional formula,

(2) each relationship of the model is mapped into one or more formulas depending on the type of relationship (Xor- and Or-groups),

(3) the resulting formula is the conjunction of all the resulting formulas of step 2,

(4) plus additional propositional constraints.

Batory [50] established the relationship that exists between feature model, propositional logic, and grammar. Batory also suggested to use logic truth maintenance system (LTMS) to infer some choices during the configuration process. Schobbens et al. [36] have formalized some operations and their complexity. Benavides et al. [89] presented a structured literature review of the existing proposals for the automated analysis of feature models. Example analyses include consistency check or dead feature detections [90], interactive guidance during configuration [91, 92], or fixing models and configurations [93–95]. It should be noted that most of the reasoning operations (e.g., satisfiability) are difficult computational problem and are NP-complete [36].

On the technical side, various kinds of automated support have been proposed:

(i) propositional logic: SAT (for satisfiability) solvers or Binary Decision Diagram (BDD) take a propositional formula as input and allow reasoning about the formula (validity, models, etc.),

(ii) constraint programming: a constraint satisfaction problem (CSP) consists of a set of variables, a set of finite domains for those variables, and a set of constraints restricting the values of the variables. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. In contrast to propositional formulas, CSP solvers can deal not only with binary values (true or false) but also with numerical values such as integers or intervals,

(iii) description logic (DL): DLs are a family of knowledge representation languages enabling the reasoning within knowledge domains by using specific logic reasoners. A problem described in terms of description logic is usually composed by a set of concepts (i.e., classes), a set of roles (e.g., properties or relationships), and set of individuals (i.e., instances). A description logic reasoner is a software package that takes as input a problem described in DL and provides facilities for consistency and correctness checking and other reasoning operations.

Benavides et al. [89] report that CSP solvers or DL solvers are mostly used for extensions of feature models (e.g., feature models with feature attributes), whereas propositional logic quite well fits basic feature models, as well as the core of the OMG's CVL proposal.

4.3. Multiviews and Compositional Approaches. At the code level, when features are implemented separately in distinct modules (files, classes, packages, plug-ins, etc.), they can easily be composed in different combinations to generate variants. Voelter and Groher [96] call this kind of variability *positive variability*, since variable elements are added together. Many techniques have been proposed to realize compositional approaches (frameworks, mixin layers,

aspects [97], stepwise refinement [98], etc.). In model-based SPL engineering, the idea is that multiple models or fragments, each corresponding to a feature, are composed to obtain an integrated model from a feature model configuration. Aspect-oriented modeling techniques have been applied in the context of SPL engineering [62, 99, 100]. Apel et al. [68] propose to revisit superimposition technique and analyze its feasibility as a model composition technique. Perrouin et al. propose a flexible, tool-supported derivation process in which a product model is generated by merging UML class diagram fragments [64].

Acher et al. [101] point out that quite often however, there is a need to compose and decompose variability models at the abstract modeling level, because variability exists across very different concerns of an SPL [59]: from functionality (e.g., particular function may only exist in some services or can be highly parameterized), deployment technology (e.g., operating system, hardware, libraries required, and dependency on middleware), specificities of data format (e.g., image format), to nonfunctional property (like security, performance or adaptability), and so forth.

Acher et al. [102] coined the term *multiple feature models* to characterize a set of feature models, possibly interrelated, that are combined together to model the variability of a system. These multiple feature models can either come from the problem domain (e.g., variability of independent services that are by nature modular entities, when independent suppliers describe the variability of their different products, etc.) or as an engineering artifact to modularize the variability description of a large system into different criteria (or concerns).

More specifically, Acher et al. [103] have identified some important issues when dealing with multiple feature models, for example, for representing SPL based on Service-Oriented Architectures.

> *Consistency Checking of Multiple*: Feature Models variability must be dealt with both within and across services. Within services, there are complex relationships between services' concerns. Across services, interactions between services (e.g., a feature of one service may exclude another feature of another service) have to be managed when services are combined to form workflows.
>
> *Grouping Feature Models*: For each category of activity to be performed in the workflow, there are several candidate services provided by different suppliers. Grouping similar services helps in finding the relevant service and in maintaining the service directory.
>
> *Updating Feature Models*: When concerns are interrelated within a service by constraints, some features of some concerns may become dead or mandatory. Hence for each concern of service the variability information needs to be updated so that each feature model is a correct representation of the set of configurations.
>
> *Reasoning Locally about Some Specific Parts of the Feature Models*: need to reason about some specific parts of the two services.
>
> *Multiple Perspectives Support*: Ideally, different experts should focus on different, specific dimension (e.g., security) and the details that are out of the scope of their expertise should be hidden. Dedicated decomposition facilities should be applied to feature models.
>
> *Multistage and Multistep Process*: The design of a product within a SPL is usually not realized in one step or by a unique stakeholder. For example, the specific requirements of an application are obtained by configuring the feature model, that is, by gradually removing the variability until only those features that are part of the final product remain. At each step of the process, it should be possible to reiterate the reasoning tasks mentioned above (consistency checking of FMs, update of FMs, etc).

On these issues, the contributions of [102] can be summarized in two main points:

(i) a set of composition and decomposition operators to support Separation of Concerns in feature modeling. The operators are formally defined (using propositional logic), fully automated, guaranteeing properties in terms of sets of configurations, and can be combined together or with other operators, for example, to realize complex reasoning tasks;

(ii) a domain-specific, textual language, called FAMILIAR, that provides an operational solution for using the different operators and managing multiple feature models on a large scale.

Instead of merging multiple feature models as in [102], another approach is to reference those multiple feature models, as proposed by Hartmann et al. [104]. This approach introduces the Supplier Independent Feature Model (SIFM) in order to select products among the set of products described by several Supplier Specific Feature Models (SSFM).

The overall idea is that any feature of SIFM, say feature F, is then related to the features F of SSFMs using cross-tree constraints. Additional constraints between SSFMs are expressed so that features F of SSFMs cannot be selected at the same time. By defining such constraints between SIFM and SSFMs, Hartmann et al. [104] allow users to build a multiple SPL thanks to several suppliers' SPLs. The SIFM is designed as follows. First, the root feature SIFM has two subfeatures: the feature Suppliers and the common root feature of SSFMs. Then, the feature Suppliers contains as many subfeatures as there are suppliers and those features are mutually exclusive (only one supplier must be selected). The SIFM contains the super-set of the features from all the suppliers and constraints are specified to inter relate features of SIFM and SSFMs. In addition, cross-tree constraints between features are specified in such a way that each child feature F is related to the corresponding features F in each appropriate SSFM.

This approach has the advantage to be realizable by current feature modeling tools and techniques. However, it

leads to reasoning on a large set of features (i.e., all the features of SIFM and SSFMs) related by a large number of constraints. The number of variables to be generated may become an issue in terms of computational or space complexity and hinder some automated analysis operations of feature models.

In the context of feature-based configuration, several other works proposed techniques to separate the configuration process into different steps or stages [92, 105]. Hubaux et al. [106] provide view mechanisms to decompose a large feature model. However they do not propose a comprehensive solution when dealing with cross-tree constraints. They also consider that the root feature should always be included, which is a limitation not imposed in [102].

4.4. Product Derivation. The product derivation process can be defined as a process of constructing products from Software Product lines, based on the modeling of variability and the choices made by the product configurator [107].

Feature Diagrams are mostly used for *product configuration* during product derivation. A feature configuration corresponds to an individual product, but lacks details on how the selected features are combined into the actual software product. Many works have thus started about 15 years ago to investigate the modeling and derivation of functional [3, 108, 109] and static [55, 110] aspects of SPL, with however much less emphasis on modeling and derivation of behavior [12, 58, 72, 78, 111], be it interaction-based (focusing on the global interactions between actors and components, e.g.; UML sequence diagrams) or state-based (concentrating on the internal states of individual components, e.g., UML StateCharts).

Product derivation methods slightly differ depending on whether the variability modeling follows an Amalgamated Approach or a Separated Approach, as defined in Sections 3.2 and 3.3.

4.4.1. Product Derivation in Amalgamated Approaches. Ziadi et al. [12] propose an algebraic specification of UML sequence diagrams as reference expressions, extended with variability operators (optionality, choice, etc.). Generic and specific constraints then guide the derivation process. Behavioral product derivation is formalized using Reference Expressions for Sequence Diagrams (RESD), that are expressions on basic Sequence Diagrams (bSDs) composed by interaction operators to provide the so-called Combined Sequence Diagrams of UML2. A RESD is an expression of the form:

<RESD>::=<PRIMARY>(“**alt**”<RESD> | “**seq**”<RESD>)∗
<PRIMARY>::=E$_\emptyset$| <IDENTIFIER> |"("<RESD>")"
|“**loop**” "("<RESD>")"

where IDENTIFIER refers to a Basic Sequence Diagram and E$_\emptyset$ is the empty Sequence Diagram (without any interaction). As introduced in Section 3.2.1, variability is introduced through three concepts: *optionality*, *variation,* and *virtuality* that are also formalized as algebraic operators to extend RESDs with variability mechanisms. The optional expression (OpE) is specified in the following form:

OpE::= “**optional**” <IDENTIFIER> "[" <RESD>"]"

A Variation expression (VaE) is defined as

VaE::= “**variation**” <IDENTIFIER> "[" <RESD>"," <RESD> ("," <RESD>)∗"]"

A virtual expression specifies a virtual SD. It is defined by a name and a reference expression:

ViE::= “**virtual**” <IDENTIFIER> "[" <RESD>"]"

SDs for Product Lines (RESD-PL) can now be defined as:

<RESD-PL>::=<PRIMARY-PL>(“**alt**” <RESD-PL> | “**seq**”<RESD-PL>)∗
<PRIMARY-PL>::=E$_\emptyset$| <IDENTIFIER> | "("<RESD-PL>")" |“**loop**” "("<RESD-PL>")" | VaE | OpE |ViE

The first step towards product behaviors derivation is to derive the corresponding product expressions from PL-RESD. Derivation needs some decisions (or choices) associated to these variability expressions to be made to produce a product specific RESD. A decision model is made of the following

(i) The presence or absence of optional expressions.

(ii) The choice of a variant expression for variation expressions.

(iii) The refinement of virtual expressions.

An Instance of a Decision Model (noted hereafter IDM) for a product P is a set of pairs $(\texttt{name}_i, \texttt{Res})$, where $\texttt{name}_i$ designates a name of an optional, variation, or virtual part in the PL-RESD and Res is its decision resolution related to the product P. Decision resolutions are defined as follows.

(i) The resolution of an optional part is either TRUE or FALSE.

(ii) For a variation part with $E_1, E_2, E_3, \ldots$ as expression variants, the resolution is i if E_i is the selected expression.

(iii) The resolution of a virtual part is a refinement expression E.

The derivation $[[\texttt{PLE}]]_{DMi}$ can then be seen as a model specialization through the interpretation of a RESD-PL PLE in the *DMi* context, where *DMi* is the instance of the decision model related to a specific product. For each algebraic variability construction, the interpretation in a specific context is quite straightforward.

(1) Interpreting an optional expression means deciding on its presence or not in the product expression. This is defined as

$$[[\textbf{optional}\ name\ [\texttt{E}]]]_{DMi} = \begin{cases} \texttt{E} & \text{if } (name, TRUE) \in DMi \\ \texttt{E}_\emptyset & \text{if } (name, FALSE) \in DMi \end{cases} \quad (1)$$

(2) Interpreting a variation expression means choosing one expression variant among its possible variants. This is defined as:

$$[[\textbf{variation}\ name\ [\texttt{E}_1, \texttt{E}_2, \ldots]]]_{DMi} = \texttt{E}_j\ if\ (name, j) \in DMi \quad (2)$$

(3) Interpreting virtual expressions means replacing the virtual expression by another expression:

$$[[\textbf{virtual } name[\mathtt{E}]]]_{DMi} = \begin{cases} \mathtt{E}' & if\ (name, \mathtt{E}') \in DMi, \\ \mathtt{E} & otherwise. \end{cases} \quad (3)$$

The derived product expressions are expressions without any variability left, that is, expressions only involving basic SDs and interaction operators: `alt, seq, and loop`. Since the empty expression ($E_\emptyset$) is a neutral element for the sequential and the alternative composition, and idempotent for the loop, derived RESD can be further simplified using algebraic rewriting rules:

(i) $\mathtt{E\ seq\ E}_\emptyset = \mathtt{E}; \mathtt{E}_\emptyset\ \mathtt{seq\ E} = \mathtt{E}$

(ii) $\mathtt{E\ alt\ E}_\emptyset = \mathtt{E}; \mathtt{E}_\emptyset\ \mathtt{alt\ E} = \mathtt{E}$

(iii) $\mathtt{loop\ (E}_\emptyset\mathtt{)} = \mathtt{E}_\emptyset$.

The second part of the derivation process proposed in [12] is to leverage StateCharts synthesis from these scenarios [112], from which direct implementations can easily be obtained [113].

Along the same general lines, Gomaa and Shin [59] define product derivation as a tailoring process involving selection of the appropriate components and setting of the parameter values for individual components to include in the product line member.

For Morin et al. [62], product derivation starts by computing a feature diagram from the product line model. Then, for a selected group of features, the *derive operation* (implemented in a generic way in the base metamodel) is called. Alternatively, Perrouin et al. [64] first configure the feature model, based on the user's feature selection, and then compose the model fragments associated to the selected features.

4.4.2. Product Derivation in Separated Approaches. As introduced in Section 3.3.1, the idea of [66] was to separate the representation of a model family (product line model) into a *feature model* (defining feature hierarchies, constraints, and possible configurations) and a *model template* (containing the union of model elements in all valid template instances). Model template elements (structural or behavioral) can be annotated with *presence conditions* (PCs) and *metaexpressions* (MEs). To derive an individual product (an instance of a model family), the configurator must first specify a valid feature configuration. Based on it, the model template is instantiated automatically. To improve the effectiveness of template instantiation, the process can be specialized by introducing additional steps: *patch application* and *simplification*. The complete template instantiation algorithm can be summarized as follows: an initial evaluation of MEs and explicit PCs, followed by computing implicit PCs and information required for patch application; then elements whose PCs are false are removed and patches are applied; the process ends with a simplification phase.

In [47], Czarnecki et al. also propose a staged configuration approach where feature models are stepwise specialized and instantiated according to the stakeholder interests at each development stage. Specialization and configuration are distinguished: specialization is defined as the process in which variabilities in feature models are removed (i.e., a specialized feature model has fewer variabilities than its parent feature model, a fully specialized feature model has no variability while a configuration is an instantiation of a feature model). The concept of multilevel staged configuration is also introduced, referring to a sequential process in which a feature model is configured and specialized by stakeholders in the development stages.

Hubaux et al. propose a formalization of this kind of multistage configuration [105]. They notably propose the formalism of feature configuration workflow in order to configure a large feature model in different steps, possibly carried out by different stakeholders.

For OVM, Pohl et al. [24] dedicate several chapters of their book to explain how product line variability can be exploited to develop different applications.

In [71] Sinnema et al. discuss in detail the entire COVAMOF Derivation Process. It is realized in four steps: product definition (the engineer creates a new Product entity in the CVV), product configuration (the engineer binds variation points to new values or variants based on customer requirements), product realization (execute the effectuation actions for each of the variants that are selected for the Product entity) and product testing (determine whether the product meets both the functional and the nonfunctional requirements), and the last three steps can occur in one or more iterations.

The goal of DOPLER [74] is to guide stakeholders through product derivation and to automatically generate product configurations. Based on the decision values set by a user, the assets required for composing the product are automatically determined and product configurations can be generated. In [114] Rabiser et al. extend the DOPLER metamodel to provide additional means to support, control and manage derivation: guidance, tasks, roles, users, and property.

Haugen et al. [54] use two transformations to derive products. A *Resolution Transformation* takes a variation model and a resolution model as input and produces a resolved variation model. Then a *Variability Transformation* takes the resolved variation model and a domain-specific model as input and produces a new, resolved domain-specific base model.

Beyond coming with a metamodel and a set of well-formedness rules expressed in OCL, the proposed OMG standard for CVL also explicitly addresses the derivation process, that is seen as the dynamic semantics of CVL (i.e., deriving products is done by "executing" CVL on a given resolution model). Semantically, the aim of deriving a resolved model from a base model and a variability model (for a given resolution model) is to reduce the solution space cardinality (the set of all possible resolved models for a given base model and a given variability model). This derivation is thus obtained by considering a variability model

as a *program* parameterized by the resolution model and operating on the base model, to provide a resolved model. Initially, the resolved model is equal to the base model. Then the execution of each *statement* of the variability model adds new constraints on the solution space, hence progressively reducing its cardinality, eventually down to 1 to get a fully resolved model, or to 0 if there are inconsistencies in the CVL model.

Since the CVL semantics is defined operationally for each statement as adding new constraints on the solution space, it boils down to giving the pre- and postcondition of the execution of each Variation Point metaclass of a CVL model. These constraints are defined using OCL pre and post conditions on an abstract *eval* operation, woven into each relevant class of the CVL metamodel. On the implementation side, Kermeta can readily be used to get an interpretor for deriving products from a product line, as implemented in [115].

4.5. Test Generation. Testing is an important mechanism both to identify defects and assure that completed products work as specified. This is a common practice in single-system development, and continues to hold in Software Product Lines. However, in the early days of SPL research, very few SPL processes addressed the testing of end-product by taking advantage of the specific features of a product line (commonality and variabilities). It was indeed clear that classical testing approaches could not directly be applied on each product since, due to the potentially huge number of products, the testing task would be far too long and expensive [116]. Hence there was a need for testing methods, adapted to the product line context, that allow reducing the testing cost [117].

For example, the early approach presented in [14, 118] is based on the automation of the generation of application system tests, for any chosen product, from the system requirements of a Product Line [119]. These PL requirements are modeled using enhanced UML use cases which are the basis for the test generation. Product-specific test objectives, test scenarios, and test cases are successively generated through an automated process. The key idea of the approach is to describe functional variation points at requirement level to automatically generate the behaviors specific to any chosen product. With such a strategy, the designer may apply any method to produce the domain models of the product line and then instantiate a given product: the test cases check that the expected functionalities have correctly been implemented. The approach is adaptive and provides automated test generation for a new product as well as guided test generation support to validate the evolution of a given product.

More recently the SPL testing field has attracted the attention of many more researchers, which results in a large number of publications regarding general and specific issues. da Mota Silveira Neto et al. [120] present a systematic mapping study, performed in order to map out the SPL testing field, through synthesizing evidence to suggest important implications for practice, as well as identifying research trends, open issues, and areas for improvement. Their goal was to identify, evaluate, and synthesize state-of-the-art testing practices in order to present what has been achieved so far in this discipline.

They identified four main test strategies that have been applied to software product lines.

(i) Incremental testing of product lines: the first product is tested individually and the following products are tested using regression testing techniques. Regression testing focuses on ensuring that everything used to work still works, that is, the product features previously tested are retested through a regression technique.

(ii) Opportunistic reuse of test assets: this strategy is applied to reuse application test assets. Assets for one application are developed. Then, the application derived from the product line use the assets developed for the first application. This form of reuse is not performed systematically, which means that there is no method that supports the activity of selecting the test assets.

(iii) Design test assets for reuse: test assets are created as early as possible in domain engineering. Domain test aims at testing common parts and preparing for testing variable parts. In application engineering, these test assets are reused, extended and refined to test specific applications. General approaches to achieve core assets reuse are: repository, core assets certification, and partial integration. The SPL principle design for reuse is fully addressed by this strategy, which can enable the overall goals of reducing cost, shortening time-to-market, and increasing quality.

(iv) Division of responsibilities: this strategy relates to select testing levels to be applied in both domain and application engineering, depending upon the objective of each phase, that is, whether thinking about developing for or with reuse. This division can be clearly seen when the assets are unit tested in domain engineering and, when instantiated in application engineering, integration, system, and acceptance testing are performed.

Specific testing activities are often split among the two types of activities: domain engineering and application engineering. Alternatively, the testing activities can be grouped into core asset and product development. From the set of studies they overview, around four adopt (or advocate the use of) the V-model as an approach to represent testing throughout the software development life cycle. However, there is no consensus on the correct set of testing levels for each SPL phase.

From the amount of studies analyzed in [120], only a few addressed testing nonfunctional requirements. They point out that during architecture design, static analysis can be used to give an early indication of problems with non-functional requirements. One important point that should be considered when testing quality attributes is the presence of trade-offs among them, for example, the trade-off between modularity and testability. This leads to natural

pairings of quality attributes and their associated tests. When a variation point represents a variation in a quality attribute, the static analysis should be sufficiently complete to investigate different outcomes. da Mota Silveira Neto et al.. highlight that investigations towards making explicit which techniques currently applied for single-system development can be adopted in SPL are needed, since studies do not address such an issue.

Their mapping study has also outlined a number of areas in which additional investigation would be useful, specially regarding evaluation and validation research. In general, SPL testing lack evidence, in many aspects. Regression test selection techniques, test automation and architecture-based regression testing are points for future research as well as techniques that address the relationships between variability and testing and techniques to handle traceability among test and development artifacts.

5. Conclusions

SPL engineering is a process focusing on capturing the *commonalities* (assumptions true for each family member) and *variability* (assumptions about how individual family members differ) between several software products. Models have been used for long as *descriptive* artifacts, and proved themselves very helpful for formalizing, sharing, and communicating ideas. Modeling variability in SPL has thus already proven itself very useful, as highlighted by the popularity of feature modeling languages and their supporting tools.

In many cases we have shown that we could go beyond that, to be able to perform computations on Variability Models, for a variety of purposes, such as validation of the consistency of models, automatic composition, or decomposition of variability models, production of new artifacts (e.g., tests), and of course concrete product derivation. These usages of variability models require that they are no longer informal, and that the language used to describe them has a well-defined abstract syntax (i.e., metamodel) and semantics. Model-Driven Engineering (MDE) makes it possible to easily implement a set of tools to process variability models, either *endomorphic* tools, processing variability models on their own, for validation (self consistency) or composition/decomposition purposes, or *exomorphic* tools, that is, generating other artifacts from variability models, such as concrete software products or test cases.

The goal of this paper was not to present an exhaustive survey on variability modeling methods and related tools, but to organize the plethora of existing approaches into several classification dimensions, and provide representative examples of Model-Driven Engineering tools and algorithms exploiting them. The reader interested in more systematic literature reviews can check [22, 120–122].

The recent outburst of variability modeling methods that we are witnessing is somehow resembling the blossoming of so many general purpose modeling languages of the early 90's, that were ultimately unified by the OMG into the UML.

Maybe it is also time for variability modeling methods to be unified into something well accepted by the community. It might be the case that the OMG is again playing this unifying role with the Common variability Language (CVL), that we introduced in this paper.

Acknowledgments

The authors wish to thank Clementine Nebut, Tewfik Ziadi, Paul Istoan, and Mathieu Acher for so many fruitful discussions on the topic of Model-Driven Engineering for Software Product Lines. This paper, largely based on common work with them, could not have been written without all their contributions to the field.

References

[1] L. M. Northrop, "A framework for software product line practice," in *Proceedings of the Workshop on Object-Oriented Technology*, pp. 365–376, Springer, London, UK, 1999.

[2] F. van der Linden, "Software product families in Europe: the esaps & cafe projects," *IEEE Software*, vol. 19, no. 4, pp. 41–49, 2002.

[3] G. Halmans and K. Pohl, "Communicating the variability of a software product family to customers," *Software and System Modeling*, vol. 2, no. 1, pp. 15–36, 2003.

[4] A. Maccari and A. Heie, "Managing infinite variability in mobile terminal software: research articles," *Software: Practice and Experience*, vol. 35, no. 6, pp. 513–537, 2005.

[5] Software Product Line Conference—Hall of Fame, http://splc.net/fame.html.

[6] L. M. Northrop, "SEI's software product line tenets," *IEEE Software*, vol. 19, no. 4, pp. 32–40, 2002.

[7] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, ACM Press, Addison-Wesley, New York, NY, USA, 2000.

[8] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and variability in software engineering," *IEEE Software*, vol. 15, no. 6, pp. 37–45, 1998.

[9] G. Perrouin, *Architecting software systems using model transformation and architectural frameworks [Ph.D. thesis]*, University of Luxembourg (LASSY)/University of Namur (PReCISE), 2007.

[10] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Henk Obbink, and K. Pohl, "Variability issues in software product lines," in *Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE'01)*, pp. 13–21, Springer, London, UK, 2002.

[11] P. Heymans and J. C. Trigaux, "Modelling variability requirements in software product lines: a comparative survey," Tech. Rep., FUNDP Namur, 2003.

[12] T. Ziadi and J. M. Jézéquel, "Product line engineering with the UML: deriving products," in *Software Product Lines*, pp. 557–586, Springer, New York, NY, USA, 2006.

[13] C. Nebut, S. Pickin, Y. le Traon, and J. M. Jézéquel, "Automated requirements-based generation of test cases for product families," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 2003.

[14] C. Nebut, Y. le Traon, and J. M. Jézéquel, "System testing of product families: from requirements to test cases," in *Software Product Lines*, pp. 447–478, Springer, New York, NY, USA, 2006.

[15] J. M. Jézéquel, "Model driven design and aspect weaving," *Journal of Software and Systems Modeling*, vol. 7, no. 2, pp. 209–218, 2008.

[16] T. Ziadi, L. Héelouët, and J. M. Jézéquel, "Towards a UML profile for software product lines," in *Software Product-Family Engineering*, vol. 3014 of *Lecture Notes in Computer Science*, pp. 129–139, 2003.

[17] D. Batory, R. E. Lopez-Herrejon, and J. P. Martin, "Generating product-lines of product-families," in *Proceedings of the Automated Software Engineering (ASE'02)*, pp. 81–92, IEEE, 2002.

[18] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, *Generative Programming for Embedded Software: An Industrial Experience Report*, Lecture Notes in Computer Science, 2002.

[19] J. Bzivin, N. Farcet, J. M. Jézéquel, B. Langlois, and D. Pollet, "Reective model driven engineering," in *Proceedings of UML 2003*, G. Booch, P. Stevens, and J. Whittle, Eds., vol. 2863 of *Lecture Notes in Computer Science*, pp. 175–189, Springer, San Francisco, Calif, USA, 2003.

[20] P. A. Muller, F. Fleurey, and J. M. Jézéquel, "Weaving executability into objectoriented meta-languages," in *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'05)*, Lecture Notes in Computer Science, pp. 264–278, Springer, Montego Bay, Jamaica, 2005.

[21] S. Pickin, C. Jard, T. Jéron, J. M. Jézéquel, and Y. le Traon, "Test synthesis from UML models of distributed software," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 252–268, 2007.

[22] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: a systematic review," in *Software Product Line Conference*, pp. 81–90, Carnegie Mellon University, Pittsburgh, Pa, USA, 2009.

[23] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. 2, no. 1, pp. 1–9, 1976.

[24] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, New York, NY, USA, 2005.

[25] J. Estublier, "Software configuration management: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering (ICSE)*, pp. 279–289, 2000.

[26] M. Erwig, "A language for software variation research," in *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*, pp. 3–12, ACM, New York, NY, USA, October 2010.

[27] M. Erwig and E. Walkingshaw, "The choice calculus: a representation for software variation," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, 2011.

[28] D. M. Weiss and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Longman, Boston, Mass, USA, 1999.

[29] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques: research articles," *Software Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[30] F. Bachmann and P. Clements, "Variability in software product lines," Tech. Rep. cmu/sei-2005-tr-012, Software Engineering Institute, Pittsburgh, Pa, USA, 2005.

[31] J. Meekel, T. B. Horton, and C. Mellone, "Architecting for domain variability," in *Proceedings of the 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, pp. 205–213, 1998.

[32] F. Bachmann and L. Bass, "Managing variability in software architectures," *SIGSOFT Software Engineering Notes*, vol. 26, no. 3, pp. 126–132, 2001.

[33] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Featureoriented domain analysis (foda) feasibility study," Tech. Rep., Carnegie-Mellon University Software Engineering Institute, 1990.

[34] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley Longman, Redwood City, Calif, USA, 2004.

[35] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, ACM Press, Addison-Wesley, New York, NY, USA, 2000.

[36] P. Y. Schobbens, P. Heymans, J. C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[37] M. L. Griss, "Implementing product-line features with component reuse," in *Proceedings of the 6th International Conerence on Software Reuse (ICSR-6)*, pp. 137–152, Springer, London, UK, 2000.

[38] P. Y. Schobbens, P. Heymans, and J. C. Trigaux, "Feature diagrams: a survey and a formal semantics," in *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pp. 136–145, IEEE Computer Society, Washington, DC, USA, 2006.

[39] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business sucCess*, ACM Press/Addison-Wesley, New York, NY, USA, 1997.

[40] J. M. Jézéquel, "Reifying configuration management for object-oriented software," in *International Conference on Software Engineering, ICSE'20*, Kyoto, Japan, April 1998.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[42] J. M. Jézéquel, "Reifying variants in configuration management," *ACM Transaction on Software Engineering and Methodology*, vol. 8, no. 3, pp. 284–295, 1999.

[43] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: a feature-oriented reuse method with domain-specific reference architectures," *Annuals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[44] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating feature modeling with the rseb," in *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*, p. 76, IEEE Computer Society, Washington, DC, USA, 1998.

[45] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecure (WICSA'01)*, pp. 45–54, IEEE Computer Society, August 2001.

[46] M. Riebisch, "Towards a more precise definition of feature models," in *Modelling Variability for Object-Oriented Product Lines*, pp. 64–76, 2003.

[47] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[48] M. Eriksson, J. Börstler, and K. Borg, "The pluss approach—domain modeling with features, use cases and use case realizations," in *Software Product Lines (SPLC)*, Lecture Notes in Computer Science, pp. 33–44, 2005.

[49] A. Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, p. 2002, 2001.

[50] D. S. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of the 9th international conference on Software Product Lines (SPLC'05)*, pp. 7–20, 2005.

[51] S. Ferber, J. Haag, and J. Savolainen, "Feature interaction and dependencies: modeling features for reengineering a legacy product line," in *Software Product Lines (SPLC)*, Lecture Notes in Computer Science, pp. 235–256, 2002.

[52] P. Istoan, *Méthodologie pour la dérivation des modèles comportementaux des produits dans les lignes de développement logiciel [Ph.D. thesis]*, Université de Rennes 1, University of Luxembourg (LASSY), 2013.

[53] A. Classen, P. Heymans, and P. Y. Schobbens, "What's in a feature: a requirements engineering perspective," in *Proceedings of 11th International Conference on Fundamental Approaches to Software Engineering*, pp. 16–30, Springer, 2008.

[54] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pp. 139–148, Limerick, UK, September 2008.

[55] M. Clauss, "Generic modeling using UML extensions for variability," in *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages*, pp. 11–18, 2001.

[56] M. Clauss and I. Jena, "Modeling variability with UML," in *Proceedings of the GCSE Young Researchers Workshop*, 2001.

[57] T. Ziadi, L. Hélouët, and J. M. Jézéquel, "Modeling behaviors in product lines," in *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02)*, pp. 33–38, 2002.

[58] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison Wesley, Longman, Redwood City, Calif, USA, 2004.

[59] H. Gomaa and M. E. Shin, "Multiple-view modelling and meta-modelling of software product lines," *IET Software*, vol. 2, no. 2, pp. 94–122, 2008.

[60] E. A. de Oliveira Jr., I. M. de Souza Gimenes, E. H. Moriya Huzita, and J. C. Maldonado, "A variability management process for software product lines," in *Proceedings of the Centre for Advanced Studies on Collaborative Research Conference (CASCON '05)*, pp. 225–241, 2005.

[61] B. Morin, J. Klein, O. Barais, and J. M. Jézéquel, "A generic weaver for supporting product lines," in *Proceedings of the 13th International Workshop on Early Aspects (EA'08)*, pp. 11–18, ACM, New York, NY, USA, 2008.

[62] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. M. Jézéquel, "Weaving variability into domain metamodels," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, pp. 690–705, 2009.

[63] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and meta-models in clafer: mixed, specialized, and coupled," in *Software Language Engineering*, B. Malloy, S. Staab, and M. van den Brand, Eds., vol. 6563 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, Berlin, Germany, 2011.

[64] G. Perrouin, J. Klein, N. Guel, and J. M. Jézéquel, "Reconciling automation and exibility in product derivation," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pp. 339–348, IEEE Computer Society, Washington, DC, USA, 2008.

[65] G. Perrouin, "Coherent integration of variability mechanisms at the requirements elicitation and analysis levels," in *Proceedingd of the Workshop on Managing Variability for Software Product Lines: Working with Variability Mechanisms at 10th Software Product Line Conference*, August 2006.

[66] K. Czarnecki and M. Antkiewicz, "Mapping features to models: a template approach based on superimposed variants," in *Proceedings of the 4th International Conference on Generative Programming and Component Enginering (GPCE'05)*, pp. 422–437, Springer, 2005.

[67] M. A. Laguna and B. González-Baixauli, "Product line requirements: multiparadigm variability models," in *Proceedings of the 11th Workshop on Requirements Engineering WER*, 2008.

[68] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, pp. 4–19, Springer, Berlin, Germany, 2009.

[69] A. Metzger, K. Pohl, P. Heymans, P. Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis," in *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*, pp. 243–253, New Delhi, India, 2007.

[70] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: a framework for modeling variability in software product families," in *Proceedings of the 3rd Software Product Line Conference (SPLC'04)*, pp. 197–213.

[71] M. Sinnema, S. Deelstra, and P. Hoekstra, "The COVAMOF derivation process," in *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components (ICSR'06)*, pp. 101–114, 2006.

[72] J. Bayer, O. Flege, and C. Gacek, "Creating product line architectures," in *IW-SAPF*, pp. 210–216, 2000.

[73] G. Campbell, N. Burkhard, J. Facemire, and J. O'Connor, "Synthesis guidebook," Tech. Rep. SPC-91122-MC, Software Productivity Consortium, Herndon, Va, USA, 1991.

[74] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study," *Automated Software Engineering*, vol. 18, no. 1, pp. 77–114, 2011.

[75] J. Xabier Mansell and D. Sellier, "Decision model and exible component definition based on xml technology," in *Software Product Family Engineering (PFE)*, Lecture Notes in Computer Science, pp. 466–472, 2004.

[76] European Software Engineering Institute Spain and IKV++ Technologies AG Germany, "Master: model-driven architecture instrumentation, enhancement and renement," Tech. Rep. IST-2001-34600, IST, 2002.

[77] K. Schmid and I. John, "A customizable approach to full lifecycle variability management," *Science of Computer Programming*, vol. 53, no. 3, pp. 259–284, 2004.

[78] C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line development: the KobrA approach," in *Proceedings of the 1st Conference on Software Product Lines: Experience and Research Directions: Experience and Research Directions*, pp. 289–309, Kluwer Academic, Norwell, Mass, USA, 2000.

[79] F. Fleurey, O. Haugen, B. Moller-Pedersen, G. Olsen, A. Svendsen, and Z. Xiaorui, "A generic language and tool for variability modeling," Tech. Rep., SINTEF, 2009.

[80] D. Beuche, "Modeling and building software product lines with pure: variants," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, p. 358, Limerick, Ireland, September 2008.

[81] C. W. Krueger, "The biglever software gears unified software product line engineering framework," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, p. 353, Limerick, UK, September 2008.

[82] M. Acher, P. Collet, P. Lahire, and R. France, "Slicing feature models," in *Proceedings of the 26th IEEE/ACM International*

Conference on Automated Software Engineering (ASE'11), pp. 424–427, January 2011.

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

[84] T. Asikainen, T. Männistö, and T. Soininen, "A unified conceptual foundation for feature modelling," in *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pp. 31–40, IEEE Computer Society, August 2006.

[85] D. Fey, R. Fajta, and A. Boros, "Feature modeling: a metamodel to enhance usability and usefulness," in *Proceedings of the 2nd International Conference on Software Product Lines (SPLC 2)*, pp. 198–216, Springer, London, UK, 2002.

[86] V. Vranic, "Reconciling feature modeling: a feature modeling metamodel," in *Proceedings of the 5th Net.ObjectDays*, pp. 122–137, Springer, 2004.

[87] M. O. Reiser, R. T. Kolagari, and M. Weber, "Unified feature modeling as a basis for managing complex system families," in *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'07)*, pp. 79–86, 2007.

[88] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: there and back again," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pp. 23–34, Kyoto, Japan, September 2007.

[89] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[90] M. Mendonca, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pp. 231–241, IEEE, 2009.

[91] T. T. Tun and P. Heymans, "Concerns and their separation in feature diagram languages—an informal survey," in *Proceedings of the Workshop on Scalable Modelling Techniques for Software Product Lines (SCALE@SPLC'09)*, pp. 107–110, 2009.

[92] M. Mendonca and D. Cowan, "Decision-making coordination and efficient reasoning techniques for feature-based configuration," *Science of Computer Programming*, vol. 75, no. 5, pp. 311–332, 2010.

[93] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, "Automated error analysis for the agilization of feature modeling," *Journal of Systems and Software*, vol. 81, no. 6, pp. 883–896, 2008.

[94] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, and A. Ruiz-Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pp. 225–234, IEEE, Limerick, UK, September 2008.

[95] M. Janota, *SAT solving in interactive configuration [Ph.D. thesis]*, Department of Computer Science at University College Dublin, 2010.

[96] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pp. 233–242, 2007.

[97] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 127–136, 2004.

[98] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling stepwise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[99] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel, "Introducing variability into aspect-oriented modeling approaches," in *Proceedings of 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Lecture Notes in Computer Science, pp. 498–513, Springer, 2007.

[100] B. Morin, F. Fleurey, N. Bencomo et al., "An aspect-oriented and model-driven approach for managing dynamic variability," in *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, vol. 5301 of *Lecture Notes in Computer Science*, pp. 782–796, 2008.

[101] M. Acher, P. Collet, P. Lahire, A. Gaignard, R. France, and J. Montagnat, "Composing multiple variability artifacts to assemble coherent workows," *Software Quality Journal*, p. 40, 2011, Special issue on Quality Engineering for Software Product Lines.

[102] M. Acher, P. Collet, P. Lahire, and R. France, "Decomposing feature models: Language, environment, and applications," in *Proceedings of the Automated Software Engineering (ASE'11)*, IEEE/ACM, Lawrence, Kansas, USA, November 2011, short paper: demonstration track.

[103] M. Acher, P. Collet, P. Lahire, and R. France, "Separation of concerns in feature modeling: support and applications," in *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12)*, Hasso-Plattner-Institut Potsdam, ACM, Potsdam, Germany, March 2012.

[104] H. Hartmann, T. Trew, and A. Matsinger, "Supplier independent feature modelling," in *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pp. 191–200, IEEE, 2009.

[105] A. Hubaux, A. Classen, and P. Heymans, "Formal modelling of feature configuration workows," in *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pp. 221–230, IEEE, 2009.

[106] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans, "A preliminary review on the application of feature diagrams in practice," in *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pp. 53–59, 2010.

[107] S. Deelstra, M. Sinnema, and J. Bosch, *Experiences in Software Product Families: Problems and Issues During Product Derivation*, Lecture Notes in Computer Science, 2004.

[108] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, "Use case description of requirements for product lines," in *Proceedings of the International Workshop on Requirement Engineering for Product Line (REPL'02)*, pp. 12–18, September 2002.

[109] T. von der Massen and H. Lichter, "Requiline: a requirements engineering tool for software product lines," in *Software Product-Family Engineering*, Lecture Notes in Computer Science, pp. 168–180, 2004.

[110] T. Ziadi and J. M. Jézéquel, "Product line engineering with the UML: products derivation," in *Families Research Book*, Lecture Notes in Computer Science, chapter WP4, Springer, New York, NY, USA, 2004.

[111] S. Robak, R. Franczyk, and K. Politowicz, "Extending the UML for modeling variability for system families," *International Journal of Applied Mathematics Computer Sciences*, vol. 12, no. 2, pp. 285–298, 2002.

[112] T. Ziadi, L. Hélouët, and J. M. Jézéquel, "Revisiting statechart synthesis with an algebraic approach," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pp. 242–251, ACM, Edinburgh, UK, May 2004.

[113] F. Chauvel and J. M. Jézéquel, "Code generation from UML models with semantic variation points," in *Proceedings of MOD-ELS/UML'2005*, S. Kent L. Briand, Ed., vol. 3713 of *Lecture Notes in Computer Science*, Springer, Montego Bay, Jamaica, 2005.

[114] R. Rabiser, P. Grunbacher, and D. Dhungana, "Supporting product derivation by adapting and augmenting variability models," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pp. 141–150, IEEE Computer Society, Kyoto, Japan, 2007.

[115] M. Gouyette, O. Barais, J. le Noir et al., "Movida studio: a modeling environment to create viewpoints and manage variability in views," in *IDM-7éme journées sur l'Ingénierie Dirigée par les ModèLes-2011*, I. Ober, Ed., vol. 1, pp. 141–145, Polytech, Université Lille 1, Service Reprographie de Polytech, Lille, France, 2011.

[116] J. D. McGregor, "Building reusable testing assets for a software product line," in *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, p. 378, Limerick, UK, September 2006.

[117] J. D. McGregor, "Testing a software product line," in *PSSE*, pp. 104–140, 2007.

[118] C. Nebut, Y. le Traon, F. Fleurey, and J. M. Jézéquell, "A requirement-based approach to test product families," in *Proceedings of the 5th Workshop on Product Families Engineering (PFE'05)*, vol. 3014 of *Lecture Notes in Computer Science*, Springer, 2003.

[119] C. Nebut, F. Fleurey, Y. le Traon, and J. M. Jézéquel, "Requirements by contracts allow automated system testing," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pp. 85–96, IEEE, 2003.

[120] P. A. da Mota Silveira Neto, I. D. Carmo MacHado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[121] S. Mujtaba, K. Petersen, R. Feldt, and M. Mattsson, "Software product line variability: a systematic mapping study," 2008.

[122] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," *Information and Software Technology*, vol. 49, no. 7, pp. 717–739, 2007.