Software diversity: state of the art and perspectives

Ina Schaefer, Rick Rabiser, David Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, Karina Villela

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Software Diversity – State of the Art and Perspectives

Ina Schaefer[1], Rick Rabiser[2], Dave Clarke[3], Lorenzo Bettini[4], David Benavides[5], Goetz Botterweck[6], Animesh Pathak[7], Salvador Trujillo[8], Karina Villela[9]

[1] TU Braunschweig, Germany, e-mail: i.schaefer@tu-bs.de
[2] Christian Doppler Laboratory for Automated Software Engineering, JKU Linz, Austria, e-mail: rick.rabiser@jku.at
[3] Katholieke Universiteit Leuven, Belgium, e-mail: dave.clarke@cs.kuleuven.be
[4] Dipartimento di Informatica, Università di Torino, Italy, e-mail: bettini@di.unito.it
[5] Dpto. de Lenguajes y Sistemas Informaticos, University of Seville, Spain, e-mail: benavides@us.es
[6] Lero, the Irish Software Engineering Research Centre, University of Limerick, Ireland, e-mail: goetz.botterweck@lero.ie
[7] INRIA Paris-Rocquencourt, France, e-mail: Animesh.Pathak@inria.fr
[8] IKERLAN, Spain, e-mail: STrujillo@ikerlan.es
[9] Fraunhofer Institute for Experimental Software Engineering, Germany e-mail: karina.villela@iese.fraunhofer.de

Received: date / Revised version: date

Abstract. Diversity is prevalent in modern software systems to facilitate adapting the software to customer requirements or the execution environment. Diversity has an impact on all phases of the software development process. Appropriate means and organizational structures are required to deal with the additional complexity introduced by software variability. This introductory article to the special section "Software Diversity – Modeling, Analysis and Evolution" provides an overview of the current state of the art in diverse systems development and discusses challenges and potential solutions. The article covers requirements analysis, design, implementation, verification and validation, maintenance and evolution as well as organizational aspects. It also provides an overview of the articles which are part of this special section and address particular issues of diverse systems development.

1 Introduction

In today's software systems, typically different system variants are developed simultaneously to address a wide range of application contexts or customer requirements. This variation is referred to as *software diversity*. Diversity impacts all phases of software development which leads to an increase of complexity, because variability has to be anticipated and managed in requirements analysis, design, implementation, and validation. Furthermore, it has to be considered during maintenance and evolution and requires appropriate organizational structures for its development.

In the early phases of software development, the diversity of a system to be developed has to be planned ahead starting from variable user requirements for a family of systems. These requirements have to be adequately represented in order to facilitate tracing them in subsequent development steps. Suitable modeling and specification techniques are required to specify system diversity during system design. These specification techniques can be (i) syntax-oriented, describing the admissible variability space with explicit linguistic constructs and defining it bottom-up from concretely specified building blocks, or (ii) semantics-oriented, specifying the variability space top-down by starting from a library of available components and restricting the admissible compositions by successively adding behavioral constraints. A particular concern is the representation of variability in the software architecture since architectural design is the essential means for structuring software systems, decomposing functionalities and enabling distributed development. During the implementation phase, programming language constructs are necessary which support the realization of diverse systems and enable reusing common code fragments for different system variants.

Diversity increases system complexity and leads to a greater risk for system failures. Efficient validation and verification methods are, thus, essential to guarantee qualities of diverse systems, such as security, consistency, correctness or performance. Like all modern software systems, diverse systems have to be adapted to address changing requirements over time. Hence, approaches supporting the evolution of diverse systems are required that allow evolving a set of diverse systems to new system versions meeting evolving user, market or technology needs.

The special section "Software Diversity – Modeling, Analysis and Evolution" provides an in-depth overview of existing techniques and tools for the modeling, implementation, analysis, and evolution of diverse systems. This introductory article reviews the state of the art

in diverse systems modeling at the requirements, design and implementation level. It provides an overview of quality assurance approaches for diverse systems and discusses support for the evolution of diverse systems. Additionally, it reviews organizational and economical aspects concerning diverse systems development. Finally, the articles contained in this special section – each of which covers one particular aspect in diverse systems development – are introduced and put into the general context. This introductory article is partly based on a previous state of the art survey [194].

2 Variability Modeling of Diverse Systems

Managing variability involves understanding the required and desired variability of diverse systems and depends strongly on the software development practices in a particular environment. Variability can either be an emergent or a planned property of software systems resulting from diverse decisions made by architects and developers to address different user requirements. Experience shows that knowledge about variability is mostly tacit in nature (and often documented *"only in the heads of the developers"*) and affects not only code, but also other kinds of development artifacts like documentation, test cases, configuration settings, etc.

Variability knowledge is typically made explicit by describing it in models. The process of documenting and defining the variability of a system, with the goal to make the tacit knowledge in the heads of different stakeholders available, is known as *variability modeling* [57]. Variability models define *the commonalities and variability of a system's artifacts with organization-specific and domain-specific properties and dependencies*. They capture the possible variants together with constraints and dependencies. Variability models can cover a system's problem space (stakeholder needs and desired features) and its solution space (architecture and components of the technical solution).

Problem space variability (also known as product line variability [168]) is relevant to the domain and needs to be understandable by domain experts utilizing the model, e.g., for configuring products. Variability models therefore define the available set of choices and the relationships among these. *Solution space variability* (also referred to as software variability [168]) means the variability of diverse reusable artifacts, such as architectural elements, components, test cases, or documents. Managing variations at different levels of abstraction and for diverse development artifacts is a daunting task, especially when the systems supporting various products are very large, as is common in industrial settings [31]. Mappings between the problem space and the solution space are important when configuring and assembling a product based on customers' requirements. Establishing *traceability between the two spaces* is also a prerequisite for automation [66].

Numerous approaches have been proposed for variability modeling, mainly in the domain of software product lines (SPL) [54,186]. Some *surveys on* particular *variability modeling/management approaches* exist, most notably on feature-oriented variability modeling [30,51,204] and on decision-oriented variability modeling [202]. Some more general surveys discuss a particular selection of approaches, e.g., [48,206].

In this section, we focus on problem space variability modeling approaches and, in particular, discuss feature-oriented and decision-oriented variability modeling as the two most prominent approaches. Solution space variability will be covered in the next section.

2.1 Feature-oriented Variability Modeling

Feature modeling is currently the most widely used approach for modeling variability. In general, a feature model captures stakeholder visible characteristics and aspects of a system, such as functional features of individual products (that might be built based on the variability model) as well as software quality attributes of both the system and the individual products to provide an overview of a system's capabilities. Starting from FODA (Feature Oriented Domain Analysis [122]), the feature-oriented view of product lines has already gone far beyond variability modeling and system documentation. Today numerous variants of feature-based variability modeling tools and techniques are available (see [30] for an extensive list). Several authors have also proposed different formal interpretations of feature models, e.g., [24,203]. Most feature models can be translated into one large formula. Each valid assignment of the formula then corresponds to a legal configuration. Hence, the feature model (and the corresponding formula) "globally" describes the set of all legal configurations. The task of configuring a feature model is directly related to the problem of satisfying the formula. We can distinguish feature models based on the type of logic that is required to represent their semantics. For instance, Boolean feature models can be represented with propositional logic.

A feature model represents the information of all possible products of a diverse system (e.g., a software product line) in terms of features and relationships among them [30]. A feature model is represented as a hierarchically arranged set of features composed by: (i) relationships between a parent (or compound) feature and its child features (or sub-features); (ii) cross-tree (or cross-hierarchy) constraints that are typically inclusion or exclusion statements such as: if feature F is included, then features A and B must be included too, or that A and B are mutually incompatible. In the general case these constraints can be arbitrary logical clauses, e.g., feature m implies (feature n or (feature p and feature q)).

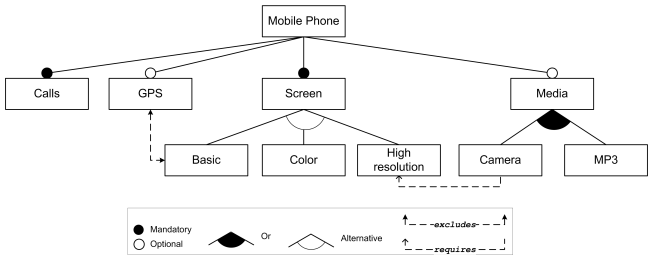

Fig. 1. Simplified feature model example inspired by the mobile phone industry taken from [30].

Fig. 1 depicts a simplified feature model [30] inspired by the mobile phone industry. The example illustrates features used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. According to the model, all phones must include support for calls, and displaying information in either a basic, color or high resolution screen. Furthermore, the software for mobile phones may optionally include support for GPS and multimedia devices such as camera, MP3 player or both of them.

Feature models are used in different scenarios of software production ranging from model-driven development [218], feature-oriented programming [24], software factories [93], to generative programming [59]. There are different feature model languages. We refer the reader to [204] for a detailed survey on the different feature model languages.

2.2 Decision-Oriented Variability Modeling

Decision modeling approaches [202] are a rather large family of approaches that exists nearly as long as feature-oriented modeling. Similar to the role which the FODA report [122] plays in the context of feature-based variability management most – if not all – existing decision modeling approaches have been influenced by the Synthesis method [45]. Decisions were defined as *"actions which can be taken by application engineers to resolve the variations for a work product of a system in the domain"* [45]. Many other researchers have also actively been publishing their research results in this area (see [202] for an overview).

Decisions are *often represented as questions* with a *defined set of possible answers*. Products are derived from a decision model by setting values to the decisions, e.g., through answering questions and following the sequence defined by the decisions' dependencies. One can say, from a historic point of view, that while the main purpose of feature models is domain analysis, the main focus of decision models is supporting derivation and configuration of products [57,192,191]. The set of values possible for a decision is defined by its data type, e.g., Boolean (answer to question can be yes or no), Enumeration/Set (users can select from a set of possible answers), Number (users can set a number as an answer), or String (users can set a text as an answer). Answering a question (and thereby selecting one or more possible values) sets a value on a decision. All decision modeling approaches allow creating *dependencies among decisions*. In the simplest approaches decisions can only lead to setting other decisions [21]. Other approaches allow rather complex combinations of formulas and conditions as a basis for determining the restrictions in making decisions [66] or explicitly support set-based descriptions [200].

id	question	range	cardinality	constraints
GPS	Do you want GPS?	yes/no	1	GPS.yes excludes Screen.Basic
ScreenType	Which type of screen do you want?	Basic | Color | High resolution	1:1	Screen.Basic excludes GPS.yes
SupportedMedia	Which media shall be supported?	Camera, MP3	0:2	SupportedMedia.Camera => ScreenType.High resolution

Fig. 2. Simplified decision model in a tabular representation for the mobile phone example from Fig. 1.

Fig. 2 depicts a decision model for the previously presented mobile phone example in a tabular notation. In difference to the feature model depicted in Fig. 1, mandatory features are not modeled at all.

The different decision modeling approaches address the *relationship of decisions to the reusable artifacts in a system* in different ways. Decisions are either referenced from the artifacts [45,200] or they reference artifacts themselves [21].

3 Diversity in System Design

Approaches to manage diversity at the design level handle *solution space variability* (or software variability [168]). Their main purpose is to represent the variability at the level of the product artifacts, such as architectural models, behavioral models or test suites. In this section, we review syntax-oriented concepts to represent variability in the solution space. In particular, we focus on architectural variability modeling which is perceived as essential for modeling complex diverse systems. As an alternative to the variability modeling approaches presented here, the paper by Jörges et al. [119] presents a constraint-based approach to represent product line variability where all possible variants satisfying the constraints are synthesized from a given set of artifacts.

3.1 Solution Space Variability

Two main approaches [223] to modeling solution space variability of software product lines exist:

- annotative approaches or superimposed variants [58] representing negative variability—all variants of the product line are included within the same model.
- compositional approaches representing positive variability—features are modeled as formal entities, possibly as refinements to a core architecture.

Superimposed variants are a monolithic representation of a system. They allow a fine-grained representation of the differences among product variants. The key problem with superimposed variants, however, is that they lack modularity and are, thus, susceptible to scalability problems. Compositional approaches rely on breaking the software product line into appropriate modules. The key disadvantage of the compositional approaches is their expressiveness: they do not deal with the removal and non-monotonic modification of behaviour when features are selected. In order to allow a modular, but yet flexible and expressive notion of variability, there also exist *transformational approaches*. In these works, variability is represented by transforming a base model in order to obtain a product variant.

Annotative approaches. Annotative variability modeling approaches consider one model representing all products of the product line. This model is sometimes also called 150%-model. Variant annotations, e.g., using UML stereotypes [92] or presence conditions [58], define which parts of the model have to be removed to derive a concrete product model. Dependencies are often defined using the UML constraint language OCL (Object Constraint Language). Similarly, decision maps in KobrA [21] define which parts of the product artifacts have to be modified for certain products.

For behavioral models captured by variants of standard modeling formalisms, such as LTS, CCS, automata, Petri Nets and so forth, variability is generally represented by labels on transitions in the models. The labels express variability in different ways, from coarse-grained to fine-grained:

- *must* or *may*. Must-transitions express behaviour common to all variants, and may-transitions express behaviour which may not always be present [78,141, 142].
- a feature name [53], indicating the presence of the transition whenever of the given feature is selected.
- an *application condition* [58,197], which is a predicate over features and, thus, corresponds to a set of sets of features (or equivalently a set of feature configurations [144]), defining precisely which feature configurations the transition belongs to.

Modal transition systems (MTS) [143] are labelled transition systems with must- and may-transitions. The former transitions represent the commonality and the latter the variability of a software product line within the same model at a coarse level of abstraction. Fischbein *et al.* [78] propose using MTS instead of ordinary labelled transition systems for modeling software product lines since MTS are more suited for the refinement of models of software product lines and provide a suitable definition of conformance. Larsen, who co-invented MTS, and collaborators independently applied MTS (specified via modal I/O automata) to modeling software product lines [142,141]. Variability is not modeled directly as in feature models, but implicitly via so-called variability models, which are actually behavioural models of the environment. In one approach, configuration of a software product line amounts to finding a suitable refinement of the variability model [141]. In the other approach, configuration is achieved via composition with selected variability models, where the variability models can in addition ignore transitions not relevant for the given variant. The results of this line of research culminated in Nyman's PhD thesis [179].

In a series of papers [76,75,19,18], the MTS approach is combined with Deontic Logic to specify software product lines. The superimposed behaviour of the product line is specified by an MTS as before, and an extended version of the classical Hennessy-Milner logic is developed for reasoning about MTSs [19,18]. This logic employs a deontic interpretation for reasoning about permitted and obliged behaviour, which is used to express both feature model and behavioural constraints in the same framework.

Classen *et al.* [53] model a product line as a single labelled transition system in a formalism known as Feature Transition Systems (FTS). In this model, transitions are labelled with the single feature they correspond to. Transitions are ordered to deal with the situation when two or more features are selected and some transition should override another. PL-CSS [94] is an extension of CCS with a variant operator to represent a family of processes. The variant operation expresses a choice based on which variant is selected. For a given run, a consistent choice is made each time the choice operator is encountered.

Feature Petri Nets [173] label Petri net transitions with *application conditions*, which are propositional formulae over features corresponding to the set of feature combinations for which the transition is valid. Application conditions provide a convenient syntactic approach for avoiding a blowout in the labels used. Dynamic Feature Petri Nets are also considered. In this model feature configurations can change at run-time.

The *Orthogonal Variability Model* introduced by Pohl *et al.* [186] captures the variability of product line artifacts in a variability model that is separated from the artifact model. Orthogonal variability models consist of variation points (description of existing differences), variants (different possibilities to satisfy a variation point), variability dependencies (possible choices, i.e., alternative, optional, mandatory), and constraint dependencies (constraints on variant selection, i.e., requires and excludes). Explicit links are drawn between variants and elements in concrete system models (e.g., UML class or use case diagrams). If a variant is not selected, the associated model elements are removed. Hence, the OVM approach is an instance of annotative variability modeling. Fig. 3 shows how the variability of a use case diagram of the mobile phone example from the previous section can be specified using OVMs where

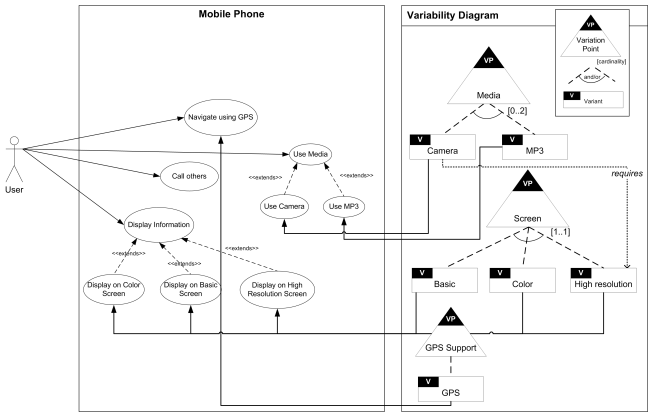

Fig. 3. Variability of a use case diagram for mobile phones specified by an OVM [186]

the triangles labeled *VP* denote variation points and the rectangles labeled *V* denote associated variants that are related to specific use cases.

Compositional approaches. Compositional approaches associate model fragments with product features that are composed for a particular feature configuration. In [104,175,223], variant models are constructed by aspect-oriented composition. Similarly, Jörges et. al.[119] employ hierarchical modeling as an aspect-oriented mechanism for specifying variability. Noda *et al.* [175] combine class diagrams and state charts into one aspect. Aspects are then composed to generate specific model variants. Wende *et al.* [104] distinguish between collaborative features that are added to specific points in a base model and aspectual features that have to be applied multiple times to the base model.

StateCharts have been used as a modular design framework for highly-entangled software components [190]; techniques have been developed for merging such specifications [174]. Also in [68], model fragments are merged in order to provide the variability model of a product line.

Harhurin and Hartmann [101] employ a service-oriented approach [160] to specifying software product lines and reasoning about feature interactions, focusing on consistency of the specification, formalized in terms of Broy's foundational framework [40].

Feature-oriented model-driven development (FOMDD) [218] combines feature-oriented programming (FOP) with the principles of model-driven engineering. Model fragments that are associated to single product features are encapsulated into feature modules. In a feature module, modeling elements can be added or refined. For a particular feature configuration, the respective feature modules are composed by adding and refining elements following the principles of stepwise refinement [27]. Apel *et al.* [12] apply model superposition to compose model fragments which is similar to FOMDD [218] without the explicit refinement statements. Model superimposition considers models with a hierarchical structure that is preserved when models are composed. FeatureAlloy [15], an extension of the modeling language Alloy, supports collaboration-based design, step-wise refinement and feature composition, and, thus, represents a formal modeling language analogous to the programming languages used in feature-oriented development.

Transformations. Apart from positive and negative variability representations, model transformations are used for capturing system diversity. The common variability language (CVL) [102], for instance, represents the variability of a base model by rules describing how modeling elements of the base model have to be substituted in order to obtain a particular product model. In [115], graph transformation rules capture the variability of a single kernel model comprising all commonality. The constraint-based approach described in [118] employs a configurable model transformation to obtain a specific product model from a hierarchical model representing a system family.

Delta modeling [49,197] is a modular approach to represent system variability via transformations. A diverse set of systems is represented by a designated core model and a set of model deltas explicitly specifying changes to the core model in order to obtain other system variants. In order to generate a particular product model for a given feature configuration, the model deltas that have to be applied for this feature configuration are selected and applied one-by-one to the core model. The result is a product model realizing the particular feature configuration.

3.2 Architectural Variability

A particular focus in the development of diverse systems is the representation of variability in the software architecture, since the architecture is perceived as a central element in the development process. For modeling architectural variability, all three types of variability modeling are used. For instance, in [155], the variability modeling language (VML) specializes the ideas of OVM for architectural models constituting an annotative approach. Also for UML component diagrams, UML stereotypes [227,92] or presence conditions [58] can be used to model variable parts of the architecture.

Compositional approaches for capturing architectural variability usually capture variations of the architecture by selecting particular component variants. Plastic partial components [183] model component variability inside the components by extending partially defined components with variation points and associated variants. Variants can be cross-cutting or non-cross-cutting architectural concerns that are composed with the common component architecture by weaving mechanisms that have to be specified by the component designer.

However, in this approach variants cannot contain variable components.

The Koala component model [180] is a first approach aiming at hierarchical variability modeling combining the variability representation with the hierarchical component structure. In Koala, the variability of a component is described by the variability of its sub-components. The selection among different sub-component variants is realized by *switches* that are used as designated components. Via explicit *diversity interfaces*, information about selected variants is communicated between sub-components and super-components in order to configure the switches to select a specific sub-component variant. Diversity interfaces and switches in Koala can be understood as concrete language constructs targeted at the implementation level to express variation points and associated variants. Hierarchical variability modeling for software product lines [98] generalizes the ideas of the Koala component model to a design level. It abstracts from the concrete language constructs used in Koala. Instead, it provides a general meta-model that integrates component variability and component hierarchy to foster component-based development of diverse systems during architectural design.

On the architectural level, several transformational approaches for variability modeling provide a modular, but still expressive mechanism. In [164], a resemblance operator is provided that allows creating a component that is a variant of an existing component by adding, deleting, renaming or replacing component elements. The old and the new component can be used together to build further components. Hendrickson *et al.* [108] use change sets containing additions and removals of components and component connections that are applied to a base line architecture. Relationships between change sets specify which change sets may be applied together. However, the order in which change sets are applied cannot be explicitly specified. Conflicts between change sets have to be resolved by excluding the conflicting combination using a relationship and providing a new change set covering the combination. This may lead to a combinatorial explosion of change sets to represent all possible variants.

Δ-MontiArc [97,96] applies the ideas of delta modeling to architecture description languages. A family of software system architectures is represented by a designated core architecture comprising hierarchically structured components that communicate via connected ports. Architectural deltas modify the core architecture to realize the architecture of other system variants. A delta can add or remove components, ports and connections and modify components by changing their internal structure. In order to obtain the architecture of a particular system variant, a subset of the architectural deltas is selected and the specified modifications are applied to the core architecture. In Δ-MontiArc, the subset of deltas that are necessary for a particular system variant have to be provided explicitly. An ordering between the different deltas can be defined capturing essential dependencies between deltas in order to ensure that the generated architectures are well-defined. Δ-MontiArc provides a modular and expressive language to represent variant-rich distributed architectures, such as function nets in the embedded systems domain or service-oriented architectures in cloud computing environments.

3.3 Mapping problem and solution space

When modeling variability, features or decisions are just (problem space) abstractions of the variability realized in real development artifacts. Understanding how features or decisions (or other problem space constructs) map to artifacts in the solution space is, thus, essential for the design of diverse systems. In practice, a wide range of mapping techniques are used. They typically relate decisions or features to *variation points* (locations in artifacts where variability occurs). Schmid and John [200] provide a set of artifact-notation-independent primitives for expressing variability in artifacts, such as optionality, alternative, set selection, and value reference. Some approaches associate artifacts with *inclusion* or *application conditions* (e.g., [58,103,66,49,197]), and some associate features or decisions with the artifacts to be included (e.g.,[21]). Other variability modeling approaches define a separate artifact model, which exposes artifact abstractions to the decision or feature model (e.g., DOPLER [66] and pure::variants [90]). FOSD [10] research has looked into different approaches of representing variability in artifacts, including conditions as annotations on product elements or artifact composition. Particularly flexible is the loose programming approach [140], a concrete realization of constraint-based variability modeling and constraint-driven product synthesis.

4 Diversity in Implementation

For diversity on the implementation level, we can distinguish the same three approaches to support variability and code reuse that we have seen on the modeling and design level.

First, *annotative* approaches mark the source code of the whole product line with respect to product features and remove marked code depending on the feature configuration. Prominent instances of annotative variability on the code level are conditional compilation, frames [23] and Colored Featherweight Java [124].

Second, *compositional approaches* assemble product implementations from code fragments associated to the product features. Most of these approaches rely on advanced program modularization techniques developed in the object-oriented programming paradigm, such as mixins [38], traits [34], or aspects [129]. In this section,

we survey these and other object-oriented extensions that enhance the standard object-oriented class-based inheritance mechanisms in order to deal with diversity of software (we refer the reader to [213,169,70] for an insightful review of the limitations of class-based object-oriented languages).

Third, as an extension of the compositional implementation approaches, delta-oriented programming [198] instantiates the ideas of delta modeling [49,197] to the implementation level, constituting a *transformational implementation technique* that we discuss at the end of this section.

Mixins [38] were proposed as a solution to limitations and problems of class-based single and multiple inheritance. A *mixin* (a class definition parametrized over the superclass) can be viewed as a function that takes a class as a parameter and derives a new subclass from it. The same mixin can be applied to many classes (this operation is known as *mixin application*), obtaining a family of subclasses with the same set of methods added and/or redefined. Since a subclass can be implemented before its superclass has been implemented, mixins remove most of the dependencies of the subclass on the superclass. Mixins have become a focus of active research in many communities and contexts: software engineering [77,207], programming language design [38,221,83,8], module systems [7,113], and distributed mobile code applications [32]. While mixins solve many problems encountered with multiple inheritance, their "linearization" strategy may still pose obstacles to code reuse.

Differently from class-based languages, object-based languages use object composition and *delegation* as mechanism for reuse code (see, e.g., [219,47,79,9]). Every object has a list of *parent* objects: when an object cannot answer a message it forwards it to its parents until there is an object that can process the message. However, runtime type errors ("message-not-understood") can arise when no delegates are able to process the forwarded message [222] and also combining delegation with a static type discipline poses some problems [132]. In [33], a language is presented for *incomplete objects* (instances of abstract classes), providing object composition, dynamic method redefinition and delegation. All these operations are type safe, and possible ambiguities due to method name clashing are checked statically. The object-based paradigm seems to be more appropriate to write components that can be reused and customized in a dynamic way, at run-time. However, behaviors that are chosen at run-time always introduce an overhead, which is not ideal in situations where the diversity of software can and must be decided statically (e.g., as in many software product lines).

In [34], a novel approach to the development of software product lines is presented that provides flexible code reuse with static guarantees. The main idea is to overcome the limitations of class-based inheritance with regard to code reuse by replacing it with *trait composition*. A *trait* [70] is a set of methods, independent from any class hierarchy. In [34], class-based inheritance (which limits the possibilities for composing products from building blocks in an arbitrary way) is ruled out and classes are built only by composition of traits, interfaces and records. Thus, the concepts of types, state, and behavior are separated into different and orthogonal linguistic concepts (interfaces, records and traits, respectively) which become the reusable building blocks that can be assembled into classes to be reused in several products of a product line.

All the aforementioned approaches have limitations with regard to expressing features that involve multiple different classes or objects. The notion of a *crosscutting concern* has been introduced to address that. This notion has been popularized in the field of *aspect-oriented programming* [129], as a way to compose advices into methods, where single pieces of advice can potentially end up in multiple different methods of different classes; *pointcuts* are used as a declarative means to identify the places where to introduce such advice. The issues involved in crosscutting concerns had already been addressed to different degrees in other approaches as well. For instance, layering of class definitions was proposed in Smalltalk, primarily for testing different implementation variations of a system [91]. *Subjective objects* [208] were introduced to an object-based language, where method dispatch is influenced by the object (subject) from which a message originates, and can potentially affect several different receiver objects in different delegation chains.

Mixin layers [207] were introduced to group mixins into layers that can then be applied to a class hierarchy in concert. This gives rise to a notion of *feature-oriented programming* [27] which allows implementing diverse systems by complementing class-based inheritance by class refinement. A feature module contains class definitions and class *refinements*. A class refinement can modify an existing class by adding new fields/methods, by wrapping code around existing methods or by changing the superclass. Mixin layers and feature-oriented programming share concepts and ideas with aspect-oriented programming, but also show some differences. Most prominently, feature-oriented programming provides support for *heterogeneous* crosscutting concerns, while aspect-oriented programming provides support for *homogeneous* crosscutting concerns. The distinction between heterogeneous and homogeneous crosscutting concerns was introduced in [56], and the implications of that difference, together with a suggestion of a unified approach, is discussed in [11]. *Context-oriented programming* [112] was introduced as a way to provide dynamic composition of heterogeneous crosscutting concerns, employing a dynamically scoped discipline for layer activation and deactivation.

Delta-oriented programming [198] is an extension of feature-oriented programming that aims at providing a

flexible modular approach for implementing SPLs. It relies on the notion of program deltas [196,197] that was first introduced in [154] to describe the modifications of object-oriented programs. The implementation of a product line in delta-oriented programming is organized into a *core module* and a set of *delta modules*. Delta modules specify changes of the core module in order to implement products. A delta module can add classes, remove classes or modify classes by changing the class structure. Delta modules have application conditions attached to define for which feature configuration the specified modifications are to be carried out. Thus, in order to generate the product implementation for a feature configuration, the modifications of the respective delta modules are applied to the core module.

5 Quality Assurance for Diverse Systems

Diversity in systems necessarily entails an additional degree of complexity, which makes ensuring system quality more difficult. Variability is the root of increased complexity, as not all properties of a system are preserved across all variants. Thus, any analysis needs to take such variability into account. Scalable techniques exploiting modularity, compositionality, incrementality, and reuse of formal artifacts are desirable.

Key challenges for the quality assurance of diverse systems include:

- scalability of the techniques involved;
- lifting, where possible, analyses to the level of the entire system family, avoiding to have to generate all instances of the system in order to perform the checks [189]; and
- developing expressive and convenient languages for expressing properties modulo variability.

Kishi and Noda remark that reuse techniques should be applied to verification artifacts, and that they should be organized in the same way that other core assets of a product line are [131,176].

5.1 Feature Model Analysis

Feature model analysis [25] aims, among other things, to find inconsistencies in feature models, such as whether a feature model has any satisfying configurations. A comprehensive survey of feature model analysis appeared recently [30], covering topics such as whether a feature model has any instances and whether a (partial) selection of features conforms to the feature model. Recent work has focused on the modularity and views of feature models [1,50,114], evolution of feature models [61,217, 89,88,205,6] and linking feature models with other artifacts, such as software architectures [193], models [58] and code [60].

5.2 Type Systems and Static Analysis

The goal of type checking the code base of a diverse system family is to ensure that all configurations are type safe, up to the degree of type safety provided by the base language, *without* having to actually generate each configuration. Other static analysis techniques check for other deficiencies, without ensuring complete type safety.

For programming languages with constructs designed specifically for building diverse systems such as software product lines, the challenge of developing type systems and other static analyses has only recently been taken up. Thaker *et al.* [215] describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. *Lightweight Feature Java* (LFJ) [63] provides a formal model of this approach.

An alternative approach is *Featherweight Feature Java* (FFJ) [14], although for this system type checking occurs only on the generated product. More recent work [13] refines the work on FFJ, expressing the code refinements into modules rather than as low-level annotations, and type checking works at the level of product lines. *Coloured Featherweight Java* [124], which employs a notion of colouring of code analogous to but more advanced than `#ifdef`s, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe.

Recent work addresses non-monotonic refinement mechanisms that can remove or rename classes and methods. Kuhlemann *et al.* [136] approach the problem using a SAT solver for an appropriate encoding of the problem, whereas Schaefer *et al.* [195] generate detailed dependency constraints for checking delta-oriented software product lines.

A number of static analysis techniques have been developed for the design models or code of software product lines. Heidenreich [105] describes techniques for ensuring the correspondence between solution space models, and problem space models, which is realised in the FeatureMapper tool. In this tool, models are checked for well-formedness against their meta-model. Similarly, Czarnecki and Pietroszek [58] provide techniques for ensuring that no ill-structured instance of a feature-based model template will be generated from a correct configuration.

Language independent frameworks [16,125] operate at the product line level for reference checking—checking which dependencies are present and satisfied—and for checking syntactic correctness.

Abstract Delta Modeling [49] is an abstract framework for describing conflicts between code refinements and conflict resolution in the setting of delta-oriented programming. The DECIMAL tool [182] performs a large

variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system.

5.3 Feature Interaction Analysis

Feature interaction is a long studied topic in the area of telecommunications systems, the goal of which is to determine whether combinations of features cause unwanted or unexpected behaviour. Often, features are developed in isolation and, when multiple features are added into the same product, interactions may occur. Generally, these are modifications of feature behaviour compared to when features operate in isolation. Two surveys covering the topic up to 2002 exist [127,43]. Some of the major research challenges include cheaply predicting when feature interactions may possibly occur, detecting precisely which feature interactions do occur, and resolving those [43]. Formal approaches to feature interaction rely on a specification of features in some logic and feature interaction amounts to an inconsistency or unsatisfiability test, a deadlock, nondeterminism, or the failure of some other safety or liveness property [210, 117]. Model checking approaches often consider single features specified in isolation and check those pairwise for interactions [44,184,15]. One particularly interesting recent approach relies on so-called *conflict-tolerant features* [69]. This approach provides a methodology and formal framework for avoiding conflicts due to feature interaction, and for ensuring that the composition mechanism selects the appropriate features via a priority-based scheme.

5.4 Model Checking

Most approaches applying model checking to product lines extend existing analysis techniques to deal with optional behaviour. Compositional model checking of software product lines involves representing the behaviour of features, e.g., as an LTS or state machine. For each property of a feature, constraints on interface states that composed features must satisfy can be generated [35, 148]. CTL can be used as a property specification language and three-valued model checking is used to ensure open verification [150]. Different programming language composition mechanisms, such as collaboration-based designs [80,147], and cross-cutting (aka aspects) features [149,133], have also been experimented with.

Thang [216] addresses the feature interaction problem using open incremental model checking considering models, where overriding incremental updates to models is possible. Liu *et al.* [153] propose an incremental, compositional model checking technique for the composition of features that computes and manages variation point obligations, and enables the reuse of verification artifacts when a new product is composed, where possible, only requiring re-verification when the obligations are not met. Guelev *et al.* [95] present criteria for checking when adding new features violates important properties of a system, which are computationally simpler than rechecking the system with the new feature added.

Model checking has also been applied to superimposed variants formalisms. Properties of Classen *et al.*'s Feature Transition Systems [53] are specified and checked using LTL. Safety properties that hold for the entire model are guaranteed to hold for all generated properties, and violations of a property result in a counterexample trace along with the products that violate the property. A version of the modal-μ calculus is used to reason about PL-CSS expressions [94]. The semantics of the formula presented is particularly interesting: rather than simply stating whether a formula is true or false, the semantics gives the set of variants for which the formula is true.

Lauenroth *et al.* [144] apply CTL model checking to a version of I/O automata where variability information (given by an OVM model [186]) is associated with transitions. Transitions are contingent on a set of feature configurations, though without loss of generality a single label can be used with an appropriate encoding. Their approach verifies that every valid feature configuration fulfills the specified properties.

In [119], model checking is used for verifying the consistent specification of variability, e.g., by demanding the absence of any underspecified (i.e. incomplete) variation points, which work in a fully hierachical fashion along the lines of [209].

5.5 Deductive Verification

Deductive verification of a software product line consists of proving that it satisfies certain functional requirements using a program logic, such as Hoare logic [17] or dynamic logic [100]. Perhaps the earliest work on the verification of a diverse system is Fisler and Robert's [82] application of the ACL2 theorem prover [126] to a feature-oriented telecommunications software system. Fisler and Robert highlight the key verification challenge, namely that a software product line can have a number of products exponential in the number of features. The approach verifies features in isolation, as open systems, and employs a lightweight analysis to determine which properties remain valid when features are composed into products—this is the standard *feature interaction* issue. They also recognize that features are often implemented as cross-cutting modules employing invasive composition [20]; thus modules are less cohesive and violate standard assumptions required for modular verification.

Batory *et al.* [26] propose the composition of proofs for conservative system extensions. The verification technique combines abstract state machines and the AHEAD methodology, and depends on the traceability of extended program elements, associated theorems and proof struc-

tures, implying the need for proof management systems in the ultimate tool chain.

Poppleton [188] propose a correctness-by-construction approach for product lines, wherein features are represented as Event-B models which undergo successive refinement from specification to implementation. Refinement proofs ensure that properties of each feature are preserved. Also the constraint-based variability modeling framework of [199,119], where product variants are synthesized automatically from abstract specifications, creates products that are correct by construction.

Delta-oriented slicing [41] was introduced to reduce the deductive verification effort across a software product line, combining techniques such as proof slicing [224] and proof reuse [29]. The technique conservatively infers which specifications remain valid for a newly generated product and which have to be re-proven.

A few works (in particular [42,81]) identify a number of shortcomings with existing approaches and challenges to be addressed in the future before scalable verification of diverse systems becomes a reality:

- Verification conditions explode (exponentially) due to variability. They should not be expressed as case distinctions in specifications, but must be addressed with compositional techniques and proof reuse.
- Formalizations of richer notions of composition are required to capture the wide variety of software composition techniques.
- The specification of behavioural constraints needs to be rich enough to enable modular verification.
- Techniques are required for determining when adding code fragments introduces (un)desirable properties into a system and, thus, invalidate existing proofs.

5.6 Testing and Run-time Verification

Another approach to quality assurance is testing [163]. A survey of testing in software product line engineering is presented in [137]. The key goal of this research is to make test suites more effective and less costly. Muccini and van der Hoek [171] provide a set of challenges and opportunities for the problem of testing software product line architectures. In general, testing can be made more effective by reducing the combinatorics, by either reusing test cases for different feature configurations, by detecting when certain tests subsume other tests, or by determining which features or feature combinations a certain test is *not* applicable to.

Pohl and Metzger provide a general overview of the area and several principals for approaching product line testing [187]. Cohen *et al.* [55] provide formal techniques for assessing the coverage and adequacy of test suites. Kang *et al.* [123] provide a basis for a formal framework for product line test development linking product line concepts to testing concepts to provide a systematic way for deriving product line tests.

Specification-based testing approaches of software product lines have also been proposed, exploiting, for example, reuse of test cases [121], and incremental refinement of test suites to match the selected feature configuration [220]. An alternative approach reduces the number of tests by determining which features are not relevant for a particular test case, so the number of configurations to which that test case applies is reduced [130]. Oster *et al.* [181] employ combinatorial testing which tests a subset of all possible products in the product line.

6 Evolution of Diverse Systems

When developing large software-intensive systems, engineers often get to a point where criteria like maintainability, traceability, and consistency get increasingly important for effective development, while they are increasingly hard to keep at a certain level. This problem is directly related to the increasing complexity of an evolving system [145]. Diverse systems are typically very complex systems that are used for many years and are inevitably subject to continuous evolution. Methodologies and guidelines are needed that assist software engineers in making well-founded choices with respect to different types of evolution (such as those defined in [62]). Various researchers summarize research challenges arising from software evolution, e.g., van Deursen *et al.* [65] and Mens *et al.* [167].

Depending on the type of system and the modeling language used for its representation different strategies for evolution have to be applied. For instance, Mens and D'Hondt [166] aim to support software evolution of UML-based models. For other types of modeling languages, there is currently little support for their evolution and many research challenges remain [65]. Several authors address the evolution of particular types of models and focus on special challenges in this context. Examples are the evolution of reactive systems [172], software architectures [87] and workflow descriptions [46, 139], e.g., based on domain-specific languages [120,84] and domain-specific modeling [39,128,159].

When evolving software, in particular when doing so with model-based techniques, one has to consider the consistency of the models. One potential approach to this challenge is to strive to preserve consistency among models, as suggested, e.g., by Engels *et al.* [74]. Such consistency preserving techniques, however, only work under certain restrictive assumptions, which can become unrealistic in practice. When dealing with multiple views, multiple stakeholders and very large, diverse systems one quickly reaches a situation where the presence of inconsistencies has to be accepted and dealt with [178]. The *One-Thing Approach* [157], which supports an extreme style of model driven design [135,158], and has been re-

alized within the jABC modeling framework [211], has been specifically design for this purpose.

When dealing with evolution in a modeling context, evolution is not limited to the models themselves. In this sense van Deursen *et al.* [65] argue that besides the evolution of models (regular evolution), other artifacts and aspects that are affected by evolution are languages and metamodels (metamodel evolution) as well as the infrastructure, code generators and frameworks (platform evolution). Moreover, it might be necessary to add additional languages (*abstraction evolution*).

Finding the right granularity for evolution is an art, but essential to make evolution of diverse systems manageable. A popular approach is to support evolution on the level of architectural elements with components as the units of evolution [180]: components are treated as black boxes and their internal structure is thus not a concern for evolution.

When modeling variability among diverse systems and managing the evolution of these systems, techniques for handling and expressing differences between models are helpful, e.g., model comparison [72,73], delta models [73,71], and change operators [146,109]. Despite its importance, comparably few publications discuss product line evolution, e.g., [36,62,162,212]. Managing evolution however is success-critical, especially in model-based product line approaches to ensure consistency after changes to meta-models, models, and actual development artifacts. Some approaches provide explicit support for particular aspects of product line evolution [67,106, 64,165,161,138,199,119].

7 Managing Diverse System Development

The development of diverse systems can occur in different ways, ranging from a series of single system developments to SPL Engineering [186]. According to Hetrick *et al.* [110], the characteristic that distinguishes software product lines from previous efforts is predictive versus opportunistic software reuse. Rather than putting general software components into a library in hope that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well-defined product line.

However, moving from a traditional engineering approach to SPL engineering requires many technical, financial, organizational, process, and market considerations to be addressed [5]. There is no "one-fits-all" approach, and knowledge and experience play an important role when trying to adopt the approach. This section presents an overview of economical, organizational and process aspects of diverse system development.

7.1 Economical Aspects

There is a clear need for demonstrating the business performance of product lines, because on the one hand they have the potential to substantially increase productivity, but on the other hand they are commonly associated to long-term strategic planning, initial investment, and long-term payback.

Several economic models and analysis approaches have been proposed in order to estimate the expected benefits of adopting SPL engineering and the required investment. According to Ali Babar *et al.* [5], they differ in the aspects of SPL economics that are taken into consideration, the depth of analysis, and the applied techniques. The authors compare 12 SPL economic models with the goal of helping practitioners decide which model or set of models best serves their needs. The study concludes that modeling SPL economics is a challenging task and there is a clear need for many more empirical studies. The main difficulties concerning the later are the confidentiality of financial data and lack of support from executives. As an alternative, some researchers use simulation models [86]. Among the directions for future research, Ali Babar *et al.* [5] mention the need for market-oriented economic models, the identification of cost drivers on finer levels of granularity, as well as the need to evaluate assets from the perspective of their quality attributes to determine whether reuse will result in an economic gain or loss.

Ahmed and Capretz [2] define a research model with seven key business factors (strategic planning, order of entry to the market, brand name strategy, market orientation, relationships management, business vision and innovation) as independent variables and the SPL business performance as a dependent variable. The authors conclude that carrying out and managing the business of software product lines require comprehensive knowledge of and expertise in these key business factors, in addition to the desired level of excellence in software engineering.

A different approach to deal with the upfront investment required for the transition to SPL engineering is presented by Krueger [134] and exemplified in [110]. The idea is to carefully assess how to reuse as much as possible of an organization's existing assets, processes, infrastructure, and organizational structures, and then find an incremental transition approach such that a small upfront investment creates immediate and incremental return on investment.

According to Krsek *et al.* [156], Krueger's ideas are useful, but are hard to apply in large financial institutions where commonalities across business unit boundaries need to be exploited. In this context, SPL engineering requires a formal approach simply because of the number and size of divisions within the organization. Krsek *et al.* emphasize that recovery and benefit allocation mechanisms between business units can present further challenges in corporate organizations. Investment

can be either funded centrally by the Chief Executive Officer or Chief Information Officer, and not explicitly recovered, or by a specific business unit or a consortium of business units. Their proposal for the latter case is the adoption of a per-use charging model.

7.2 Organizational Infrastructure

The organizational dimension of diverse system development deals with the way the organization is able to manage complex relationships between the developed artifacts and the respective employee responsibilities [151]. From the cooperation with several software development organizations applying SPL principles, Bosch [37] identified a number of alternatives to the traditional organizational model consisting of a domain engineering unit and several application engineering units: 1) domain engineering projects and application engineering projects, 2) domain engineering projects, whose project teams consist of members from most business units, such that afterwards each business unit can extend functionality and make the newer version of the shared assets available, 3) specialized domain engineering units develop and evolve the reusable assets for a subset of the SPL products. According to the author, several factors influence the choice of the organizational model: the size of the product line and the engineering staff, geographical distribution, project management maturity, organizational culture, and the nature of the system family.

Krueger [134] advocates the automatic composition and configuration of different products from the core assets to eliminate, among other problems, the organizational delineation between domain engineering teams and application engineering teams and the consequent "us-versus-them culture".

Ahmed *et al.* [4] compiled six key organizational factors (organizational structure, culture, conflict management, change management, commitment, and learning) from the literature and carried out a survey with the purpose of understanding the influence of these factors in the institutionalization of SPL engineering within an organization. The empirical results strongly support the hypothesis that all those organizational factors, but conflict management are positively associated with the performance of SPL engineering in an organization.

Ganesan *et al.* [85] use source code history logs to understand the current development style of the existing products (fixed or dynamic team structure), as well as to identify product experts and commonalities among developers. The authors base their approach on the assumption that the adoption of SPL engineering starts with the assessment of the current status, which includes organizational stability, maturity, staff turnover, domain expertise, and project management maturity.

7.3 Processes

Clements and Northrop [54] organize SPL development in three essential macro-activities: core asset development, product development, and management. *Core asset development* and *product development* from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products.

In addition, Pohl *et al.* [186] present a framework with two key SPL engineering processes (domain engineering and application engineering), while *Bayer et al.* [28] defined a methodology to develop software product lines that has been refined, populated (in terms of new methods) and applied in several projects [201,116]. The methodology is organized in deployment phases, technical components, and support components.

Several strategies for introducing SPL engineering have been reported, e.g., [201,110,226,156]. Yoshimura *et al.* [226] present a migration process composed of the following activities: estimate economic benefits, redefine the development process, restructure the organization, assess the merge potential, perform merging, and maintain the software product line. Krsek *et al.* [156] propose a set of relevant processes: define funding, structure the organization, define the product line, manage risks, develop acquisition strategy, and others. Hetrick *et al.* [110] report a incremental transition composed of four sequential stages: transition of the infrastructure and core assets, transition of the team organization, transition of the development processes, and transition of the validation and quality assurance.

Some authors have worked on the integration of SPL engineering and agile development [170,99,177,22]. Hanssen and Fægri [99] performed a qualitative case study in which they identified three interacting customer-centric software processes: strategic, tactical and operational. The *strategic process* has a SPL engineering style and implements long-term strategic plans. The *tactical process* has the agile development style and seeks to polish, improve or otherwise simplify to moderate adjustments to the product. The *operational process* aims at sustaining a good level of satisfaction with the software in its day-to-day use. Mohan *et al.* [170] performed a secondary data analysis of a case study and identified a set of successful practices in the process that integrates SPL engineering and agile methods, such as selective refactoring and the development of a flexible architecture.

Ahmed and Capretz [3] proposes a maturity model for SPL engineering, by building upon the SPL maturity evaluation framework proposed by van der Linden *et al.* [152]. This framework prescribes four dimensions (Business, Architecture, Process, and Organization), and respective assessment models. In [3], the five levels of maturity for the SPL engineering process are characterized and an assessment approach based on a fuzzy inference system is proposed.

8 Summary and Overview of Special Section

In this introductory article, we have reviewed the state of the art in the development of diverse software systems. As we have shown, software diversity impacts all phases of software development, from requirements analysis, over system design and implementation, up to quality assurance and system analysis. Furthermore, we have looked at product line evolution and the particular aspects of managing diverse software systems from an organizational and economic perspective. In this special section, we have collected a number of articles focussing on particular aspects in the development of diverse software systems, providing detailed insights into some areas covered in this introductory article.

The article "Visualization of Variability and Configuration Options" [185] by Pleuss and Botterweck considers the problem space variability expressed by feature models which was reviewed in Section 2.1. The authors present an interactive visualization of feature models to support the configuration of product variants by feature selection. Additionally, they provide automatic validation of the selected configurations based on a reasoning engine.

In their article "A Constraint-based Variability Modeling Framework" [119], Jögres et al. present constraint-based variability modeling as a conceptual alternative to structure-oriented variability modeling concepts which we considered in Section 2. The authors illustrate constraint-based variability modeling using two approaches: first, constraint-guarded variability modeling where manually selected configuration options are validated by constraint checking, and second, constraint-driven variability modeling, where the actual product variants are obtained by automatic synthesis techniques to satisfy the given constraints.

The article "Revealing and Repairing Configuration Inconsistencies in Large-Scale Software Systems" [214] by Tartler et al. focusses on the consistency between problem and solution space variability which we considered in Section 3.3. The presented approach derives the variability from Linux configuration models and from the implementation of the Linux kernel and represents both in propositional logic in order to check that configurable and implemented variability match.

The article "A Code Tagging Approach to Software Product Line Development" [111] by Heymans et al. considers the implementation of software product lines which is covered in Section 4. The authors propose a code tagging approach to insert variability into the implementation of existing software systems without changing the existing programming paradigms or development processes. Additionally, the tagging approach allows tracing code-level variability to the feature model which can be used for product configuration.

The article "The ABS Tool Suite: Modeling, Executing and Analysing Distributed Adaptable Object-Oriented Systems" [225] by Wong et al. mainly concerns the design and implementation of diverse software systems which is reviewed in Sections 3 and 4. The authors provide an overview of the Abstract Behavioral Specification (ABS) language and tool suite, which is a comprehensive platform for developing highly adaptive, distributed and concurrent software systems. Using the ABS, system variability is consistently traceable from the requirements level to the object behavior. The analysis capabilities of the associated tool suite range from simulation facilities for debugging to a designated resource analysis.

In their article "Model Checking Software Product Lines with SNIP" [52], Classen et al. focus on the analysis of software product lines which was covered in Section 5. The authors present the SNIP model checker that takes as input the variability specification of the feature model and the behavioral descriptions of the artifacts used to build the product variants. SNIP then allows efficiently analyzing all possible product variants by exploiting their similarities.

In their article "Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions" [107], Heider et al. focus on product line evolution and, in particular, on evolving the products derived from a product line. The continuous evolution of both the reusable artifacts and derived products in product lines is a major challenge in practice as we described in Section 6. Heider et al. explore how different types of product line changes influence the derived products and present a tool-supported approach, which facilitates evolution by capturing and replaying configuration decisions.

Acknowledgements. This research is partly funded by the EternalS Coordination Action (FP7-247758) (`http://www.eternals.eu`), the EU project HATS (FP7-231620) (`http://www.hats-project.eu`). Ina's work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under the grants SCHA1635/1-1 and SCHA1635/2-1. Rick's work has partly been supported by the Christian Doppler Forschungsgesellschaft, Austria and Siemens VAI Metals Technologies. Lorenzo's work has partly been supported by the MIUR project DISCO (PRIN 2008). David's work has partly been supported by the European Commission (FEDER), Spanish Government under the CICYT project SETI (TIN2009-07366); and projects THEOS (TIC-5906) and ISABEL (P07-TIC-2533) funded by the Andalusian Local Government. Goetz' work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (`www.lero.ie`). Animesh's work is done as part of the European FP7 ICT FET CONNECT project (http://connect-forever.eu/).

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Composing feature models. In: SLE. LNCS, vol. 5969, pp. 62–81.

Springer (2009)
2. Ahmed, F., Capretz, L.: Managing the business of software product line: An empirical investigation of key business factors. Inf. and Soft. Technology 49(2), 194–208 (2007)
3. Ahmed, F., Capretz, L., Samarabandu, J.: Fuzzy inference system for software product family process evaluation. Information Sciences 178(3), 2780–2793 (2008)
4. Ahmed, F., Capretz, L., Sheikh, S.: Institutionalization of software product line: An empirical investigation of key organizational factors. Journal of Systems and Software 80(6), 836–849 (2007)
5. Ali, M., Babar, M.A., Schmid, K.: A comparative survey of economic models for software product lines. In: SEAA. pp. 275–278 (2009)
6. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE. pp. 201–210. ACM (2006)
7. Ancona, D., Zucca, E.: A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. Mathematical Structures in Computer Science 12(6), 701–737 (2001)
8. Ancona, D., Lagorio, G., Zucca, E.: Jam - designing a java extension with mixins. ACM TOPLAS 25(5), 641–712 (2003)
9. Anderson, C., Barbanera, F., Dezani-Ciancaglini, M., Drossopoulou, S.: Can Addresses be Types? (a case study: Objects with Delegation). In: WOOD. ENTCS, vol. 82(8), pp. 1–22. Elsevier (2003)
10. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology 8(5), 49–84 (2009)
11. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: ICSE. pp. 122–131. ACM Press (2006)
12. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: International Conference on Model Transformation (ICMT) (2009)
13. Apel, S., Kästner, C., Größlinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Autom. Softw. Eng. 17(3), 251–300 (2010)
14. Apel, S., Kästner, C., Lengauer, C.: Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In: GPCE. pp. 101–112 (2008)
15. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In: ISSRE. pp. 161–170 (2010)
16. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Language-independent reference checking in software product lines. In: FOSD. pp. 65–71. ACM (2010)
17. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd Edition. Texts in Computer Science, Springer (2009)
18. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A logical framework to deal with variability. In: IFM. LNCS, vol. 6396, pp. 43–58. Springer (2010)
19. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: A deontic logical framework for modelling product families. In: VaMoS. pp. 37–44 (2010)
20. Aßmann, U.: Invasive Software Composition. Springer (2003)
21. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison-Wesley (2002)
22. Babar, M., Ihme, T., Pikkarainen, M.: An industrial case of exploiting product line architectures in agile software development. In: SPLC. pp. 171–177 (2006)
23. Bassett, P.G.: Framing software reuse: lessons from the real world. Prentice-Hall (1997)
24. Batory, D.: Feature models, grammars, and propositional formulas. In: SPLC. LNCS, vol. 3714, pp. 7–20. Springer (2005)
25. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated Analysis of Feature Models: Challenges Ahead. Commun. ACM 49(12) (2006)
26. Batory, D., Börger, E.: Modularizing Theorems for Software Product Lines: The Jbook Case Study. Journal of Universal Computer Science 14(12) (2008)
27. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE TSE 30(6), 355–371 (2004)
28. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: PuLSE: A methodology to develop software product lines. In: Proceedings of the 1999 Symposium on Software Reusability. pp. 122–131 (1999)
29. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In: SEFM. pp. 77–86. IEEE Computer Society (2004)
30. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated analysis of feature models 20 years later. Information Systems 35(6), 615 – 636 (2010)
31. Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: From problem to solution space. In: SAICSIT. pp. 182–191 (2005)
32. Bettini, L., Bono, V., Venneri, B.: MoMi: a calculus for mobile mixins. Acta Informatica 42(2-3), 143–190 (2005)
33. Bettini, L., Bono, V., Venneri, B.: Delegation by object composition. Science of Computer Programming 76(11), 992–1014 (2011)
34. Bettini, L., Damiani, F., Schaefer, I.: Implementing Software Product Lines using Traits. In: SAC, OOPS Track. pp. 2096 –2102. ACM (2010)
35. Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: Parameterized interfaces for open system verification of product lines. In: ASE. pp. 258–267 (2004)
36. Bosch, J.: Design and Use of Software Architectures, Adopting and Evolving a Product Line Approach. Addison-Wesley (2000)
37. Bosch, J.: Software product lines: Organizational alternatives. In: ICSE. pp. 91–100 (2001)
38. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP. ACM SIGPLAN Notices, vol. 25(10), pp. 303–311. ACM Press (1990)
39. Braun, V., Margaria, T., Steffen, B., Yoo, H., Rychly, T.: Safe service customization. In: Intelligent Network Workshop, 1997. IN '97., IEEE. vol. vol.2, p. 4 pp. vol.2 (May 1997)
40. Broy, M.: Service-oriented systems engineering: Modeling services and layered architectures. In: FORTE. LNCS, vol. 2767, pp. 48–61 (2003)

41. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: FoVeOOS. LNCS, vol. 6528. Springer (2010)
42. Bubel, R., Din, C., Hänle, R.: Verification of variable software: an experience report. In: FoVeOOS. LNCS, vol. 6528. Springer (2010)
43. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Computer Networks 41(1), 115–141 (2003)
44. Calder, M., Miller, A.: Feature interaction detection by pairwise analysis of LTL properties – A case study. Formal Methods in System Design 28(3), 213–261 (2006)
45. Campbell, G. H., J., Faulk, S.R., Weiss, D.M.: Introduction to synthesis. Tech. rep., INTRO SYNTHESIS PROCESS-90019-N, Software Productivity Consortium, Herndon, VA, USA (1990)
46. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. 24(3), 211–238 (1998)
47. Chambers, C.: Object-Oriented Multi-Methods in Cecil. In: ECOOP. LNCS, vol. 615, pp. 33–56. Springer (1992)
48. Chen, L., Babar, M.A.: A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology 53(4), 344–362 (2011)
49. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: GPCE. ACM (2010)
50. Clarke, D., Proença, J.: Towards a theory of views for feature models. In: FMSPLE. Technical Report, University of Lancaster, U.K. (2010)
51. Classen, A., Heymans, P., Schobbens, P.Y.: What's in a feature: A requirements engineering perspective. In: FASE. LNCS, vol. 4961/200, pp. 16–30. Springer (2008)
52. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model Checking Software Product Lines with SNIP. STTT (October 2012), (in this issue)
53. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: ICSE. IEEE (2010)
54. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley (2001)
55. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: ROSATEA. pp. 53–63 (2006)
56. Colyer, A., Clement, A.: Large-scale AOSD for middleware. In: AOSD. pp. 56–65. ACM Press (2004)
57. Czarnecki, K.: Variability modeling: State of the art and future directions. In: VaMoS. p. 11. ICB-Research Report No. 37, University of Duisburg Essen (2010)
58. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE. pp. 422–437. Springer (2005)
59. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley (2000)
60. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: GPCE. pp. 211–220 (2006)
61. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC. pp. 23–34 (2007)
62. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. Journal of Systems and Software 74(2), 173–194 (2005)
63. Delaware, B., Cook, W.R., Batory, D.S.: Fitting the pieces together: a machine-checked model of safe composition. In: ESEC/SIGSOFT FSE. pp. 243–252 (2009)
64. Deng, G., Gray, J., Schmidt, D., Lin, Y., Gokhale, A., Lenz, G.: Evolution in model-driven software product-line architectures. In: Designing Software-intensive Systems, pp. 1280–1312. Idea Group Inc (2008)
65. van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. In: MoDSE. pp. 41–49. University of Nantes (2007)
66. Dhungana, D., Grünbacher, P., Rabiser, R.: The dopler meta-tool for decision-oriented variability modeling: A multiple case study. Automated Software Engineering 18(1), 77–114 (2011)
67. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software 83(7), 1108–1122 (2010)
68. Dhungana, D., Neumayer, T., Grünbacher, P., Rabiser, R.: Supporting Evolution in Model-Based Product Line Engineering. In: SPLC (2008)
69. D'Souza, D., Gopinathan, M.: Conflict-tolerant features. In: CAV. LNCS, vol. 5123, pp. 227–239. Springer (2008)
70. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
71. Eclipse-Foundation: Atlas Model Weaver. Website, `http://www.eclipse.org/gmt/amw/`
72. Eclipse-Foundation: EMF Compare. Website, `http://www.eclipse.org/modeling/emft/?project=compare`
73. Eclipse-Foundation: Epsilon Project. Website, `http://www.eclipse.org/gmt/epsilon/`
74. Engels, G., Heckel, R., Küster, J., Groenewegen, L.: Consistency-preserving model evolution through transformations. In: UML Int. Conf. LNCS, vol. 2460, pp. 212–226. Springer (2002)
75. Fantechi, A., Gnesi, S.: Formal Modeling for Product Families Engineering. In: SPLC (2008)
76. Fantechi, A., Gnesi, S.: A behavioural model for product families. In: ESEC/SIGSOFT FSE. pp. 521–524 (2007)
77. Findler, R., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In: ICFP. pp. 94–104. ACM (1998)
78. Fischbein, D., Uchitel, S., Braberman, V.A.: A foundation for behavioural conformance in software product line architectures. In: ROSATEA. pp. 39–48 (2006)
79. Fisher, K., Mitchell, J.C.: A Delegation-based Object Calculus with Subtyping. In: FCT. LNCS, vol. 965, pp. 42–61. Springer (1995)
80. Fisler, K., Krishnamurthi, S.: Modular verification of collaboration-based software designs. In: ESEC / SIGSOFT FSE. pp. 152–163 (2001)
81. Fisler, K., Krishnamurthi, S.: Decomposing verification around end-user features. In: VSTTE. LNCS, vol. 4171, pp. 74–81. Springer (2005)
82. Fisler, K., Roberts, B.: A case study in using ACL2 for feature-oriented verification. In: Fifth International

Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04) (2004)
83. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: POPL. pp. 171–183. ACM Press (1998)
84. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley / ACM Press (2011), http://books.google.de/books?id=ri1muolw_YwC
85. Ganesan, D., Muthig, D., Knodel, J., Yoshimura, K.: Discovering organizational aspects from the source code history log during the product line planning phase–a case study. In: WCRE. pp. 211–220 (2006)
86. Ganesan, D., Muthig, D., Yoshimura, K.: Predicting return-on-investment for product line generations. In: SPLC. pp. 13–24 (2006)
87. Garlan, D., Barnes, J., Schmerl, B., Celiku, O.: Evolution styles: Foundations and tool support for software architecture evolution. In: WICSA/ECSA. pp. 131–140. IEEE (2009)
88. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in Alloy. In: Alloy Workshop. pp. 71–80 (2006)
89. Gheyi, R., Massoni, T., Borba, P.: Algebraic laws for feature models. J. UCS 14(21), 3573–3591 (2008)
90. pure systems GmbH: Variant management with pure::variants, technical whitepaper (2006)
91. Goldstein, I., Bobrow, D.: Extending object-oriented programming in Smalltalk. In: Conference on LISP and Functional Programming. pp. 75–81. ACM Press (1980)
92. Gomaa, H.: Designing Software Product Lines with UML. Addison-Wesley (2005)
93. Greenfield, J., Short, K.: Software Factories. Hungry Minds (2004)
94. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and model checking software product lines. In: FMOODS. LNCS, vol. 5051, pp. 113–131. Springer (2008)
95. Guelev, D.P., Ryan, M.D., Schobbens, P.Y.: Model-checking the preservation of temporal properties upon feature integration. STTT 9(1), 53–62 (2007)
96. Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.: Delta-oriented Architectural Variability using MontiCore. In: Workshop on Software Architecture Variability (SAVA) (2011)
97. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Delta Modeling for Software Architectures. In: Workshop on Model-based Development of Embedded Systems (MBEES) (2011)
98. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.: Hierarchical variability modeling for software architectures. In: SPLC (2011)
99. Hanssen, G., Faegri, T.: Process fusion: An industrial case study on agile software product line engineering. Journal of Systems and Software 81(6) (2008)
100. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
101. Harhurin, A., Hartmann, J.: Towards consistent specifications of product families. In: FM. LNCS, vol. 5014, pp. 390–405. Springer (2008)
102. Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding standardized variability to domain specific languages. In: SPLC. pp. 139–148. IEEE (2008)
103. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to models. In: ICSE. pp. 943–944. ACM (2008)
104. Heidenreich, F., Wende, C.: Bridging the Gap Between Features and Models. In: Aspect-Oriented Product Line Engineering (2007)
105. Heidenreich, F.: Towards systematic ensuring well-formedness of software product lines. In: Workshop on Feature-Oriented Software Development. pp. 69–74. ACM (2009)
106. Heider, W., Rabiser, R., Dhungana, D., Grünbacher, P.: Tracking evolution in model-based product lines. In: MAPLE. pp. 59–63. Software Engineering Institute, Carnegie Mellon (2009)
107. Heider, W., Rabiser, R., Grünbacher, P.: Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. STTT (October 2012), (in this issue)
108. Hendrickson, S.A., van der Hoek, A.: Modeling Product Line Architectures through Change Sets and Relationships. In: ICSE (2007)
109. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP. pp. 52–76. Springer (2009)
110. Hetrick, W., Krueger, C., Moore, J.: Incremental return on incremental investment: Engenios transition to software product line practice. In: OOPSLA. pp. 798–804 (2006)
111. Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., Demonceau, L.: A Code Tagging Approach to Software Product Line Development. STTT (October 2012), (in this issue)
112. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology 7(3), 125–151 (2008)
113. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. In: ESOP. LNCS, vol. 2305, pp. 6–20. Springer (2002)
114. Höfner, P., Khédri, R., Möller, B.: Algebraic view reconciliation. In: SEFM. pp. 149–158. IEEE Computer Society (2008)
115. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: MoDELS. pp. 151–165 (2007)
116. John, I., Knodel, J., Schulz, T.: Applied Software Product Line Engineering, chap. Efficient Scoping with CaVE: A Case Study, pp. 421–445. CRC Press (2010)
117. Jonsson, B., Margaria, T., Naeser, G., Nyström, J., Steffen, B.: Incremental requirement specification for evolving systems. Nordic J. of Computing 8, 65–87 (March 2001)
118. Jörges, S.: Genesys: A Model-Driven and Service-Oriented Approach to the Construction and Evolution of Code Generators. Ph.D. thesis, Technische Universitt Dortmund (2011)
119. Jörges, S., Lamprecht, A.L., Margaria, T., Schaefer, I., Steffen, B.: A Constraint-based Variability Modeling Framework. STTT (October 2012), (in this issue)
120. Jörges, S., Margaria, T., Steffen, B.: Genesys: service-oriented construction of property conform code generators. ISSE 4(4), 361–384 (2008)

121. Kahsai, T., Roggenbach, M., Schlingloff, B.H.: Specification-based testing for software product lines. In: SEFM. pp. 149–158. IEEE Computer Society (2008)
122. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
123. Kang, S., Lee, J., Kim, M., Lee, W.: Towards a formal framework for product line test development. In: CIT. pp. 921–926. IEEE Computer Society (2007)
124. Kästner, C., Apel, S.: Type-Checking Software Product Lines - A Formal Approach. In: ASE. pp. 258–267. IEEE (2008)
125. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D.S.: Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In: TOOLS. pp. 175–194 (2009)
126. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA (2000)
127. Keck, D.O., Kühn, P.J.: The feature and service interaction problem in telecommunications systems. a survey. IEEE Trans. Software Eng. 24(10), 779–796 (1998)
128. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press (2008)
129. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: ECOOP. LNCS, vol. 1241, pp. 220–242. Springer (1997)
130. Kim, C., Batory, D., Khurshid, S.: Reducing combinatorics in testing product lines. In: AOSD (2011)
131. Kishi, T., Noda, N.: Formal verification and software product lines. Commun. ACM 49(12), 73–77 (2006)
132. Kniesel, G.: Type-Safe Delegation for Run-Time Component Adaptation. In: ECOOP. LNCS, vol. 1628, pp. 351–366. Springer (1999)
133. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: SIGSOFT FSE. pp. 137–146 (2004)
134. Krueger, C.: New methods in software product line practicel. Communications of the ACM 49(12), 37–40 (2006)
135. Kubczak, C., Jörges, S., Margaria, T., Steffen, B.: eXtreme Model-Driven Design with jABC. In: CTIT Proc. of the Tools and Consultancy Track of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA). vol. WP09-12, pp. 78–99 (2009)
136. Kuhlemann, M., Batory, D.S., Kästner, C.: Safe composition of non-monotonic features. In: GPCE. pp. 177–186 (2009)
137. Lamancha, B.P., Usaola, M.P., Velthius, M.P.: Software product line testing - a systematic review. In: ICSOFT. pp. 23–30 (2009)
138. Lamprecht, A.L., Margaria, T., Schaefer, I., Steffen, B.: Comparing Structure-oriented and Behavior-oriented Variability Modeling for Workflows. In: Moschitti, A., Scandariato, R. (eds.) 1st International Workshop on Eternal Systems (EternalS'11). Communications in Computer and Information Science (CCIS), vol. 225. Springer (2011)
139. Lamprecht, A., Margaria, T., Steffen, B.: Seven Variations of an Alignment Workflow - An Illustration of Agile Process Design and Management in Bio-jETI. In: Bioinformatics Research and Applications. LNBI, vol. 4983, p. 445456. Springer, Atlanta, Georgia (2008)
140. Lamprecht, A.L., Naujokat, S., Margaria, T., Steffen, B.: Synthesis-Based Loose Programming. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC) (Sep 2010)
141. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: ESOP. LNCS, vol. 4421, pp. 64–79. Springer (2007)
142. Larsen, K.G., Nyman, U., Wasowski, A.: Modeling software product lines using color-blind transition systems. STTT 9(5-6), 471–487 (2007)
143. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS. pp. 203–210. IEEE Computer Society (1988)
144. Lauenroth, K., Pohl, K., Toehning, S.: Model checking of domain artifacts in product line engineering. In: ASE. pp. 269–280 (2009)
145. Lehman, M.: Programs, life cycles, and laws of software evolution. IEEE Information Processing Letters 68(9), 1060–1076 (1980)
146. Lerner, B.: A model for compound type changes encountered in schema evolution. ACM Trans. Database Syst. 25(1), 83–127 (2000)
147. Li, H.C., Fisler, K., Krishnamurthi, S.: The influence of software module systems on modular verification. In: SPIN. LNCS, vol. 2318, pp. 60–78. Springer (2002)
148. Li, H.C., Krishnamurthi, S., Fisler, K.: Interfaces for modular feature verification. In: ASE. pp. 195–204 (2002)
149. Li, H.C., Krishnamurthi, S., Fisler, K.: Verifying cross-cutting features as open systems. In: SIGSOFT FSE. pp. 89–98 (2002)
150. Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features using three-valued model checking. Autom. Softw. Eng. 12(3), 349–382 (2005)
151. van der Linden, F.: Software product families in europe: The esaps & cafè projects. IEEE Software 19(4), 41–49 (2002)
152. van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., Obbink, H.: Software product family evaluation. In: SPLC. pp. 110–129 (2004)
153. Liu, J., Basu, S., Lutz, R.R.: Compositional model checking of software product lines using variation point obligations. Autom. Softw. Eng. 18(1), 39–76 (2011)
154. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In: ECOOP. LNCS, vol. 3586, pp. 169–194. Springer (2005)
155. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models. In: Software Composition, LNCS, vol. 4954. Springer (2008)
156. M. Krsek, J. van Zyl, R.R., Clohesy, B.: Experiences of large banks: Hurdles and enablers to the adoption of software product line practices in large corporate organisations. In: SPLC. pp. 161–169 (2008)
157. Margaria, T., Steffen, B.: Business Process Modelling in the jABC: The One-Thing-Approach. In: Cardoso,

J., van der Aalst, W. (eds.) Handbook of Research on Business Process Modeling. IGI Global (2009)
158. Margaria, T., Steffen, B.: Continuous Model-Driven Engineering. IEEE Computer 42(10), 106—109 (Oct 2009)
159. Margaria, T., Steffen, B., Kubczak, C.: Evolution support in heterogeneous service-oriented landscapes. J. Braz. Comp. Soc. 16(1), 35–47 (2010)
160. Margaria, T., Steffen, B., Reitenspieß, M.: Service-Oriented Design: The Roots. In: ICSOC, pp. 450–464 (2005)
161. Mattsson, M., Bosch, J.: Frameworks as components: a classification of framework evolution. In: Nordic Workshop on Programming Environment Research. pp. 63–174. Ronneby, Sweden (1998)
162. McGregor, J.: The evolution of product line assets. Tech. rep., CMU/SEI-2003-TR-005 ESC-TR-2003-005 (2003)
163. McGregor, J.D.: Testing a software product line. In: PSSE. LNCS, vol. 6153, pp. 104–140. Springer (2007)
164. McVeigh, A., Kramer, J., Magee, J.: Using resemblance to support component reuse and evolution. In: SAVCBS. pp. 49–56 (2006)
165. Mende, T., Beckwermert, F., Koschke, R., Meier, G.: Supporting the grow-and-prune model in software product lines evolution using clone detection. In: CSMR. pp. 163–172. IEEE CS (2008)
166. Mens, T., D'Hondt, T.: Automating support for software evolution in UML. Autom. Softw. Eng. 7(1), 39–59 (2000)
167. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: IWPSE. pp. 13–22. IEEE Computer Society (2005)
168. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE. pp. 243–253. IEEE (2007)
169. Mikhajlov, L., Sekerinski, E.: A Study of the Fragile Base Class Problem. In: ECOOP. LNCS, vol. 1445, pp. 355–383. Springer (1998)
170. Mohan, K., Ramesh, B., Sugumaran, V.: Integrating software product line engineering and agile development. IEEE Software 27(3), 48–55 (2010)
171. Muccini, H., van der Hoek, A.: Towards testing product line architectures. Electr. Notes Theor. Comput. Sci. 82(6) (2003)
172. Müller-Olm, M., Steffen, B., Cleaveland, R.: On the Evolution of Reactive Components: A Process-Algebraic Approach. In: Proceedings of the Second Internationsl Conference on Fundamental Approaches to Software Engineering. pp. 161–175. FASE '99 (1999)
173. Muschevici, R., Clarke, D., Proença, J.: Feature Petri nets. In: FMSPLE. Technical Report, University of Lancaster, U.K. (2010)
174. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of StateCharts specifications. In: ICSE. pp. 54–64 (2007)
175. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: SPLC (2008)
176. Noda, N., Kishi, T.: Design verification tool for product line development. In: SPLC. pp. 147–148 (2007)
177. Noor, M.A., Rabiser, R., Grünbacher, P.: Agile product line planning: A collaborative approach and a case study. Journal of Systems and Software 81(6), 868–882 (2008)
178. Nuseibeh, B., Easterbrook, S., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software 58(2), 171–180 (2001)
179. Nyman, U.: Modal Transition Systems as the Basis for Interface Theories and Product Lines. Ph.D. thesis, Department of Computer Science, Aalborg University (2008)
180. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Computer 33(3), 78–85 (2000)
181. Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: SPLC. pp. 196–210. Springer (2010)
182. Padmanabhan, P., Lutz, R.R.: Tool-supported verification of product line requirements. Autom. Softw. Eng. 12(4), 447–465 (2005)
183. Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic Partial Components: A solution to support variability in architectural components. In: WICSA/ECSA (2009)
184. Plath, M., Ryan, M.D.: Plug-and-play features. In: FIW. pp. 150–164 (1998)
185. Pleuss, A., Botterweck, G.: Visualization of Variability and Configuration Options. STTT (October 2012), (in this issue)
186. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
187. Pohl, K., Metzger, A.: Software product line testing. Commun. ACM 49(12), 78–81 (2006)
188. Poppleton, M.: Towards Feature-Oriented Specification and Development with Event-B. In: REFSQ. pp. 367–381 (2007)
189. Post, H., Sinz, C.: Configuration lifting: Verification meets software configuration. In: ASE. pp. 347–350 (2008)
190. Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams. Software and System Modeling 3(3), 221–234 (2004)
191. Rabiser, R., Grünbacher, P., Dhungana, D.: Supporting product derivation by adapting and augmenting variability models. In: SPLC. pp. 141–150. IEEE (2007)
192. Rabiser, R., O'Leary, P., Richardson, I.: Key activities for product derivation in software product lines. Journal of Systems and Software 84(2), 285–300 (2011)
193. Satyananda, T.K., Lee, D., Kang, S.: Formal verification of consistency between feature model and software architecture in software product line. In: ICSEA. p. 10 (2007)
194. Schaefer, I., Bettini, L., Botterweck, G., Clarke, D., Costanza, C., Pathak, A., Rabiser, R., Trujillo, S., Villela, K.: Survey on diversity awareness and management. Tech. rep., Deliverable 2.1 of the EternalS Coordination Action (FP7-247758) (2011)
195. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: AOSD. ACM Press (2011)

196. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: MAPLE (2009)
197. Schaefer, I.: Variability modelling for model-driven development of software product lines. In: VaMoS. pp. 85–92 (2010)
198. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented Programming of Software Product Lines. In: SPLC. LNCS, vol. 6287, pp. 77–91. Springer (2010)
199. Schaefer, I., Lamprecht, A.L., Margaria, T.: Constraint-oriented Variability Modeling. In: Rash, J., Rouff, C. (eds.) 34th Annual IEEE Software Engineering Workshop (SEW-34). IEEE CS Press (2011), to appear
200. Schmid, K., John, I.: A customizable approach to full-life cycle variability management. Journal of the Science of Computer Programming, Special Issue on Variability Management 53(3), 259–284 (2004)
201. Schmid, K., John, I., Kolb, R., Meier, G.: Introducing the PuLSE approach to an embedded system population at testo ag. In: ICSE. pp. 544–552 (2005)
202. Schmid, K., Rabiser, R., Grünbacher, P.: A comparison of decision modeling approaches in product lines. In: VaMoS. pp. 119–126. ACM (2011)
203. Schobbens, P., Trigaux, J., Heymans, P., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)
204. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature diagrams: A survey and a formal semantics. In: RE. pp. 139–148. IEEE (2006)
205. Segura, S., Benavides, D., Cortés, A.R., Trinidad, P.: Automated merging of feature models using graph transformations. In: GTTSE. LNCS, vol. 5235, pp. 489–505. Springer (2007)
206. Sinnema, M., Deelstra, S.: Classifying variability modeling techniques. Information and Software Technology 49(7), 717–739 (2006)
207. Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM TOSEM 11(2), 215–255 (2002)
208. Smith, R., Ungar, D.: A simple and unifying approach to subjective objects. ACM TOPLAS 2(3), 161–178 (1996)
209. Steffen, B., Margaria, T., Braun, V., Kalt, N.: Hierarchical Service Definition. Annual Review of Communications of the ACM 51, 847–856 (1997)
210. Steffen, B., Margaria, T., Braun, V.: Coarse-granular model checking in practice. In: Proceedings of the 8th international SPIN workshop on Model checking of software. pp. 304–311. SPIN '01 (2001)
211. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Hardware and Software, Verification and Testing, Lecture Notes in Computer Science, vol. 4383, pp. 92–108. Springer Berlin / Heidelberg (2007)
212. Svahnberg, M., Bosch, J.: Evolution in software product lines: two cases. Journal of Software Maintenance: Research and Practice 11(6), 391–422 (1999)
213. Taivalsaari, A.: On the notion of inheritance. ACM Computing Surveys 28(3), 438–479 (Sep 1996)
214. Tartler, R., Sincero, J., Dietrich, C., Schröder-Preikschat, W., Lohmann, D.: Revealing and Repairing Configuration Inconsistencies in Large-Scale Software Systems. STTT (October 2012), (in this issue)
215. Thaker, S., Batory, D.S., Kitchin, D., Cook, W.R.: Safe composition of product lines. In: GPCE. pp. 95–104 (2007)
216. Thang, N.T.: Incremental Verification of Consistency in Feature-Oriented Software. Ph.D. thesis, Japan Advanced Institute of Science and Technology (2005)
217. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE. pp. 254–264 (2009)
218. Trujillo, S., Batory, D., Diaz, O.: Feature oriented model driven development: A case study for portlets. In: ICSE. pp. 44–53. IEEE CS (2007)
219. Ungar, D., Smith, R.B.: Self: The power of simplicity. ACM SIGPLAN Notices 22(12), 227–242 (1987)
220. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE Trans. Software Eng. 36(3), 309–322 (2010)
221. Van Limberghen, M., Mens, T.: Encapsulation and Composition as Orthogonal Operators on Mixins: a Solution to Multiple Inheritance Problems. Object Oriented Systems 3(1), 1–30 (1996)
222. Viega, J., Tutt, B., Behrends, R.: Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Tech. Rep. CS-98-03, UVa Computer Science (1998)
223. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: SPLC. pp. 233–242 (2007)
224. Wehrheim, H.: Slicing techniques for verification re-use. Theor. Comput. Sci. 343(3), 509–528 (2005)
225. Wong, P.Y.H., Albert, E., Muschevici, R., Proenca, J., Schäfer, J., Schlatte, R.: The ABS Tool Suite: Modeling, Executing and Analysing Distributed Adaptable Object-Oriented Systems. STTT (October 2012), (in this issue)
226. Yoshimura, K., Ganesan, D., Muthig, D.: Defining a strategy to introduce a software product line using existing embedded systems. In: EMSOFT. pp. 63–72 (2006)
227. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: Workshop on Product Familiy Engineering. pp. 129–139 (2003)