SN Computer Science (2023) 4:35
https://doi.org/10.1007/s42979-022-01456-8

ORIGINAL RESEARCH

Projectional Editing of Software Product Lines Using Multi-variant Model Editors

Johannes Schröpfer[1] · **Thomas Buchmann**[1] · **Bernhard Westfechtel**[1]

Received: 30 September 2021 / Accepted: 12 October 2022 / Published online: 1 November 2022

Abstract

Model-driven software engineering (MDSE) as well as software product line engineering (SPLE) achieve productivity gains by raising the level of abstraction and fostering organized reuse. Consequently, the integrating discipline model-driven software product line engineering (MDSPLE) aims at combining the best of both worlds by creating multi-variant models which are (automatically) configured into single-variant models which are in turn adapted further (if required). Inherently complex multi-variant models call for urgently needed tools providing support for editing multi-variant models. In this paper, we present a framework for projectional multi-variant editors which make complexity manageable using a user-friendly representation. At all times, a domain engineer is aware of editing a multi-variant model which is necessary to assess the impact of changes on all model variants. Supporting a clear separation of product space (domain model) and variant space (variability annotations), our projectional multi-variant editors provide a novel approach to representing variability information which is displayed non-intrusively. Furthermore, the domain engineer may employ a projectional multi-variant editor to adapt the representation of the multi-variant domain model in a flexible way, according to the current focus of interest.

Keywords Model-driven development · Software product lines · Multi-variant model · Projectional editing · Ecore · Generic framework

Introduction

In *model-driven software engineering* (MDSE) [1], software systems are developed by creating high-level models which are analyzed, simulated, executed, or transformed into code. In this context, *models* are structured artifacts which are instantiated from metamodels. A *metamodel* defines the types of elements from which models are composed and the rules for their composition. For metamodels, the Object Management Group (OMG) has defined the *MOF* standard (*Meta Object Facility*) a subset of which is implemented as *Ecore* in the *Eclipse Modeling Framework* (*EMF*) [2].

Models may be represented in a variety of different ways, including diagrams, trees, tables, or human-readable text. Different kinds of editors may be employed to create and modify models. In the case of a textual representation, a *syntax-based editor* may be used which persists the text and derives the underlying model by an incremental parsing process. In the EMF ecosystem, the Xtext[1] framework is frequently used to generate syntax-based editors from language descriptions.

In contrast, *projectional editors* provide for commands operating directly on the model and project the model onto a suitable representation [3, 4]. A projectional editor may ensure syntactic correctness of models and enjoys further advantages concerning tool integration. In particular, since models are stored as instances of metamodels, unique identifiers may be assigned to model elements such that they may be referenced in a reliable way.

Communicated by Slimane Hammoudi and Luis Ferreira Pires.

This article is part of the topical collection "Model-Driven Engineering and Software Development" guest edited by Slimane Hammoudi and Luis Ferreira Pires.

✉ Johannes Schröpfer
Johannes.Schroepfer@uni-bayreuth.de

Thomas Buchmann
Thomas.Buchmann@uni-bayreuth.de

Bernhard Westfechtel
Bernhard.Westfechtel@uni-bayreuth.de

[1] Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany

[1] https://www.eclipse.org/Xtext.

Software product line engineering (SPLE) [5] is a discipline which is concerned with the systematic development of families of software systems from reusable assets. To this end, common and discriminating features of family members are captured in a variability model, e.g., a *feature model* [6]. In *domain engineering*, a variability model is developed along with a set of reusable assets which are used in *application engineering* to derive product variants.

Product variants may be constructed in different ways. In case of *positive variability*, they are composed from reusable modules. In case of *transformational variability*, product variants are constructed by applying a sequence of transformations. In case of *negative variability*, *multi-variant artifacts* are represented as superimpositions of annotated elements. An *annotation* constitutes a presence condition over features. A product variant is defined by a *feature configuration*, stating which features have to be included and excluded, respectively. To construct a *single-variant artifact*, all elements are removed from a multi-variant artifact whose annotations evaluate to false.

Model-driven software product line engineering (MDSPLE) combines MDSE with SPLE. Thus, SPLE is applied to models. While most SPLE approaches focus on source code rather than models, a number of MDSPLE tools have been developed, e.g., FeatureMapper [7], FAMILE [8], and SuperMod [9] all of which are based on EMF.

Contribution

This paper is a significant extended version of [10] and presents a *framework* for generating *projectional multi-variant model editors*. This framework is based on our previous work on projectional single-variant editors for models in the technological space of EMF. As described in [11], a projectional editor may be generated from a metamodel for domain models (an Ecore model defining classes, attributes, and references) and a syntax definition which maps model elements to a human-readable textual representation. In the work presented in this paper, we have extended the framework for single-variant model editors into a framework for building multi-variant editors. This extension is generic, i.e., it depends neither on the underlying metamodel nor on the syntax definition. Thus, no additional development effort is required to turn a single-variant editor into a multi-variant editor. The extensions compared to [10] comprise a much more in depth description of the editor and its underlying architecture as well as a complete new running example.

A projectional multi-variant editor which has been built with the help of our framework is characterized by the following properties:

1. So far, our projectional editors support *human-readable text* as the external representation of EMF models. In MDSE, human-readable text is becoming increasingly popular. One example constitutes the UML-based textual language *Action Language for Foundational UML* (*ALF*) [12] that also comes along with appropriate tool support in EMF [13]. The editor's design is extensible; further representations such as diagrams may be added in the future.
2. Projectional multi-variant editors are based on *negative variability* (probably the main-stream SPLE approach). Thus, engineers do not have to learn new languages. Rather, domain models are augmented with *annotations*.
3. For modeling variability, *feature models* (the most widespread notation in SPLE) are used. All annotations refer to features and attributes from a feature model for the software product line.
4. Projectional multi-variant editors are designed to support *domain engineering*. Since multi-variant models are the artifacts of domain engineering, a projectional multi-variant editor *directly operates* on a *multi-variant model*. Thus, all variants may be considered by the domain engineer during editing. Furthermore, each command has a uniquely determined semantics. These properties distinguish our approach from variation control systems [14] which are faced with view-update problems and limited awareness in filtered views.
5. Internally, annotations of model elements are stored in a separate *mapping model* [8]. Thus, existing domain metamodels may be reused. Furthermore, the relationships between features in the feature model and elements of domain models are captured in one single central data structure. This approach facilitates traceability and propagation of changes from the feature model to annotated domain models.
6. The mapping model is shielded from the user. Rather, annotations are displayed intuitively along with the model elements in a *single representation*. Therefore, the user does not have to deal with the internal concepts and structure of the mapping model. In contrast to approaches based on preprocessor directives, annotations are *separated clearly* from domain model elements in the representation of a multi-variant model.
7. Projectional multi-variant editors include commands for *projectional editing* of *annotations*. Thus, annotations are handled as structured objects rather than as text strings. Context-free correctness of annotations is guaranteed by the projectional editor.
8. To cope with complexity, projectional multi-variant editors provide several commands for *adapting the representation* of an annotated model to the current *focus of interest*. For example, annotations may be hidden completely or selectively. In this way, the representation of the model may be simplified. It should be noted, how-

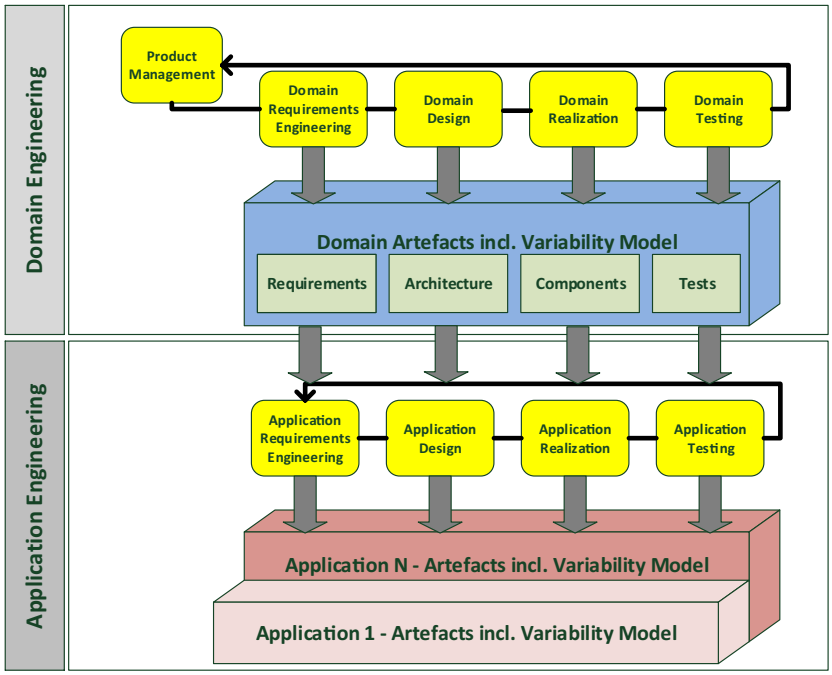

Fig. 1 Software product line engineering process distinguishing between domain engineering and application engineering [5]

ever, that all editing commands still refer to the underlying multi-variant model.

Overview

The rest of this paper is structured as follows: Section 2 explains the background of our research and related work. Section 3 describes the functionality and the user interface of projectional multi-variant editors. An example is given in Sections 4 and 5 outlines the model-based internal architecture underlying these editors. Section 6 details a specific aspect of the realization: the mapping between domain models and feature models. Finally, Section 7 concludes the paper.

Background and Related Work

As stated in the previous section, the main goal of software product line engineering is to (automatically) derive single applications from a common platform by employing organized reuse. A special development process is a crucial prerequisite in order to be successful. In the SPLE literature [5, 15], a typical development process has been established which distinguishes between the sub-disciplines *domain engineering* and *application engineering* as shown in Fig. 1. During domain engineering, the product domain is analyzed, capturing the results in a variability model (e.g., a feature model). Typically, *Feature-Oriented Domain Analysis* (*FODA*) [6] or one of its descendants (like FORM [16]) is used to analyze the domain. Furthermore, an implementation—the so called *platform*—is provided at the end of domain engineering. The platform comprises a set of reusable development artefacts that contain all commonalities and variabilities of the products being in the scope of the product line. In this regard, evolution of software product lines poses a significant field of research [17–19].

Typically, the variability is captured and expressed in a variability model. Different formalisms exist to describe commonalities and differences among members of the product line. As one representative, *feature models* [6, 20] use features as boolean properties of a software system which can be either present or absent in a specific product. Features are arranged in an AND/OR-tree. In the case of an AND-decomposition, all of its child features have to be selected when the parent is selected. In contrast, for an OR-decomposition, at least one child has to be selected. Additionally, XOR-decompositions enforce the selection of exactly one child. Depending on the respective variant of feature models, refining modeling constructs are provided, such as *requires* and *excludes* relationships [21] or *cardinality-based feature modeling* [22]. While feature models describe the variability of the entire product line, *feature configurations* describe the characteristics of individual products thereof.

Application engineering on the other hand deals with the construction of specific product variants by employing and exploiting the reusable assets developed in domain engineering. Deriving products can be achieved following three different approaches:

Positive Variability This approach fosters building variable artifacts around

	a common core which is shared by all variants. *Composition* techniques [23] are used to derive the final products.
Transformational Variability	This approach uses a sequence of transformations which are performed in a predefined order to construct products [24].
Negative Variability	This approach relies on a *superimposition* of all variants which is created in the form of a *multi-variant product* [25, 26]. Specific products are derived by removing all fragments of artifacts implementing features which are *not* contained in the desired product.

In all three cases, the application engineer binds the variability by creating a *feature configuration*. In a feature configuration, a selection state (selected or deselected) is assigned to each feature variable. A feature configuration is *consistent* if the provided selections conform to the constraints defined in the feature model. However, a consistent feature configuration cannot guarantee a consistent product in general [27].

For the reminder of this paper we want to put our emphasis on *negative variability*. Negative variability extends single- to multi-variant artifacts by annotating artifact elements. In contrast to positive and transformational variability, existing languages for single-variant artifacts may be reused. Thus, SPL engineers do not have to learn new languages and furthermore, existing implementation artifacts may be easily integrated into the platform. However, editing multi-variant artifacts poses a significant cognitive challenge: For example, editing source code written in the programming language C turns out to be difficult because preprocessor directives realizing annotations are intermingled with ordinary C code.

Therefore, dedicated *multi-variant editors* are required for making the complexity manageable. To date, quite a number of rather different approaches have been proposed and implemented. All of these approaches suffer from different shortcomings:

Virtual separation of concerns [25, 28] applies C-like preprocessor directives to Java code. A syntax-based editor supports separation of concerns by assigning colors to features and by eliding deselected program fragments. It is obvious that coloring works only for a small set of features and furthermore, preprocessor directives are still intermingled with ordinary code. Consequently, two different aspects of the software product line are mixed in one physical resource: features and their implementing source code artifacts. In addition, as soon as a product is derived, the traceability links between features and code are lost, as the preprocessor removes all preprocessor directives before passing the source code to the language compiler.

Model-driven tools such as FeatureMapper [7] and FAMILE [8] follow a different approach: Annotations are stored and visualized in a dedicated mapping model. While annotations are separated from models, the SPL engineer is exposed to an internal data structure which should be hidden from the user. Furthermore, it is hard to understand the relationships between model elements and annotations. Eventually, the mentioned tools support domain engineering only. As soon as a product is derived, the connection to the platform of the product line (including the mapping information) is lost.

In contrast to the approaches having been discussed above, *variation control systems* reduce complexity by *filtered editing* [14]. From a multi-variant artifact called *source*, a *view* is materialized in which variability has been resolved completely or partially. After editing of the view has been finished, the performed changes are propagated back to the source. Variation control systems are faced with two problems: Limited awareness of the context in which editing is performed and delineation of the scope of the change in the variant space. While the other tools mentioned above primarily display and edit artifacts of domain engineering, and provide (filtered) views of the multi-variant domain model, variation control systems operate on (partially configured) products. Thus, the software product line engineer works on specific variants (as in application engineering) and domain engineering is performed implicitly when a commit into the version repository is executed.

In contrast to the approaches mentioned above, our approach completely hides the mapping model from the user. The SPL engineer is empowered to use feature annotations in the concrete syntax in an intuitive way. Furthermore, the annotations may be hidden in the editor at any time. Sophisticated visualizations allow for different views of the multi-variant domain model ranging from fine-grained views showing single features only over partial feature configurations to a view of the complete multi-variant domain model.

In our approach, application engineering is a fully automated process where the final products are derived from the multi-variant domain model using a feature configuration. The derived artifacts comprise traceability links to the multi-variant domain model which is a significant aspect

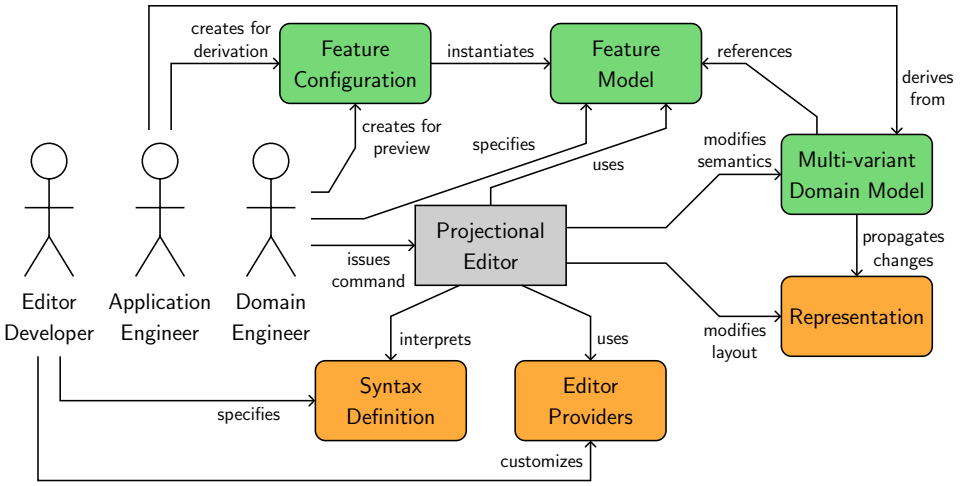

Fig. 2 Overview of roles and modeling artifacts within the editor framework. The application engineer's role is not considered in this paper

for evolution. If the derived products are modified and/or completed, changes can be propagated back into the multi-variant model employing the tracability links. In addition, this process requires a specification describing which variants are supposed to be affected by these remote changes (apart from the derived product variant itself).

In [29] the authors present a multi-view projectional editor for software product lines based on JetBrains MPS.[2] In contrast to our approach which is generic and allows for reusing existing EMF technology (modeling languages, model transformations, code generators), the PEoPL solution focuses on combining annotative and compositional editing of software product lines for a specific modeling language.

Functionality and User Interface

This section illustrates the functionality of the editor framework. Before considering the different kinds of annotations, the general functionality of the editors with respect to creating and modifying multi-variant models is described. Finally, some aspects regarding the integration of this framework are summarized.

The Framework in General

Figure 2 depicts the different roles of users and their use cases within the framework. The context of the framework is the Eclipse Modeling Framework, i.e., all metamodels of the considered models are instances of the Ecore metamodel. First, the editor developer configures the editor for the respective (domain-specific) language. The description of the lexical and contextfree syntax is specified by means of projection rules which map classes and structural features from the metamodel (abstract syntax) to concrete text. In addition, static semantics with respect to scoping and validation rules may be added programmatically by implementing generated classes that are initially empty.

The domain engineer specifies the feature model for the respective product line. By using commands within the projectional editor, the multi-variant domain model is built and modified. The editor visualizes the domain model as an abstract syntax tree—by interpreting the syntax definition specified by the editor developer—augmented with appropriate annotations which refer to the feature model. Projectional editor commands are provided to add and remove objects or to set attribute values and cross links—with respect to domain model elements as well as their annotations. Furthermore, the framework also persists the representation of the underlying model. While commands regarding the multi-variant domain model affect the representation respectively, the domain engineer can also modify the layout directly, e.g., by adding additional whitespace characters or line breaks. The domain engineer may create feature configurations to visualize filtered views of the multi-variant model including fully configured previews of single-variant products.

Eventually, the application engineer derives single variants from the multi-variant model. To this end, feature configurations are created. Note that the role of the application engineer is not considered in this paper. Nevertheless, feature configurations also play a significant role for the editor within domain engineering as (potentially partial) feature configurations can be used for defining arbitrarily filtered views of the multi-variant model.

All in all, we differentiate three different roles two of which are the typically considered roles in product line engineering. While the syntax definition, the editor providers, and the representation are specific to the editor framework, the multi-variant domain model, the feature model, and the feature configuration—that constitute the relevant artifacts

[2] https://www.jetbrains.com/mps/.

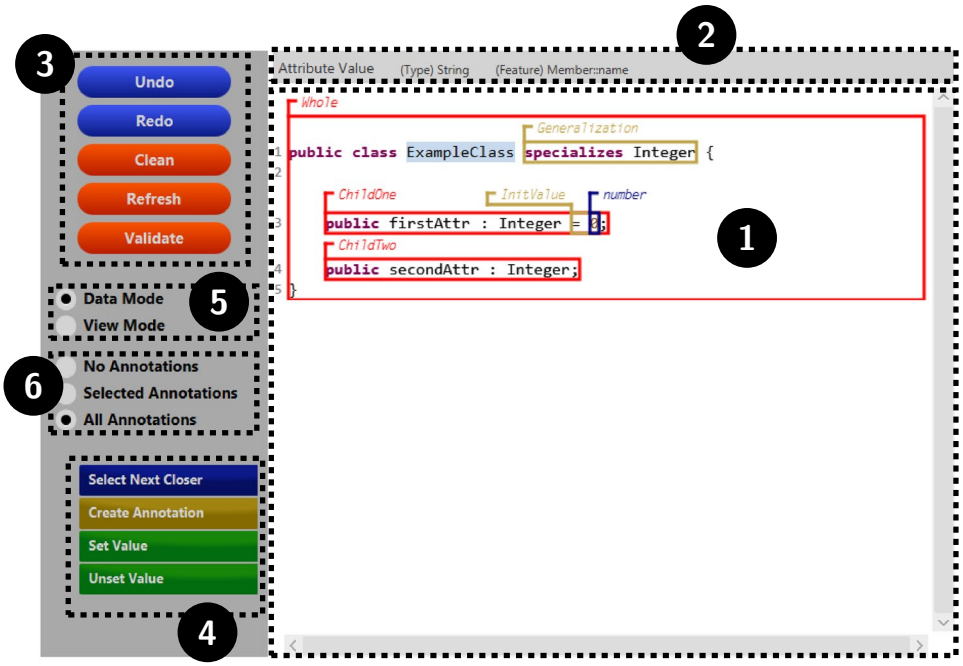

Fig. 3 Screenshot of the editor user interface for a small example ALF model

within the product line context—can be (re-)used outside the framework, as well.

The editor combines the different types of commands in one integrated view of the multi-variant domain model. The editor visualizes an arbitrary abstract syntax tree or an appropriate subtree. Figure 3 shows the editor for an example ALF model. The major part constitutes the main pane (cf. part 1) that visualizes the multi-variant domain model. The annotations are depicted by labels between the code lines. When selecting model elements in the main pane, the respective information about the model element is shown (cf. part 2). In the upper-left corner, the editor provides buttons for general commands, e.g., the common editor operations *Undo* and *Redo* as well as the validation process (cf. part 3) which do not depend on the currently chosen mode. In addition, mode-specific commands (depending on the mode that is chosen) may be executed by clicking buttons (cf. part 4) that depend on the current mode and the selected model element.

With respect to the models affected by the editor commands, the editor differentiates two modes (cf. part 5). Within the *data mode*, the domain model as well as the mapping model is modified before the representation is adjusted accordingly. The *view mode* offers purely representation commands, e.g., adding line breaks or whitespaces.

The annotations are displayed as labels between the lines that contain the respective model elements. With respect to the visualization of the annotations, three different modes are supported (cf. part 6). The domain engineer may choose whether *no annotations* are visible at all—that results in an editor without product line context—, whether *all annotations* of the represented domain model subtree are visible, or whether an arbitrary subset of all annotations is visible (*selected annotations*), e.g., annotations that contain a certain feature.

While domain model, mapping model, and the representation model are edited by means of commands within the editor, the feature model is built and modified using an extra editor. Currently, the generic EMF tree editor is used to this end. For future work, we plan a more adequate projectional editor with a simple and intuitive textual syntax. The respective metamodel for feature models is applied to our framework; thus, a projectional editor is used for both feature model and domain model which furthermore minimizes the dependencies to other tools.

Support for Domain Engineering

The editor always visualizes one representation model. Each representation refers to one subtree of a domain model. Therefore, one domain model can be represented by several representation models each of which corresponds to another subtree of the domain model. Eventually, the product line may be realized by several domain models. Thus, a product line of multiple domain models is represented by multiple representation models that are visualized by multiple editors providing different views of the range of domain models. In case of the example, there is exactly one domain model. This model consists of an ALF package which contains several classes. Each representation model refers to a class, i.e., each class is visualized by an own editor—analogously to Java

classes, for instance. The ALF class shown here contains two properties and one generalization.

Annotations contain links to features and attributes in the feature model within boolean expressions that are built by means of common logical operators (conjunction, ex- and inclusive disjunction, negation). Annotations are internally represented as subtrees. Projectional commands allow for adding and removing objects, restructuring expression subtrees, and setting links in the editor. Within the user interface, the annotations are located between the physical lines representing the domain model. For each physical line, the annotations referring to its elements are shown above the line.

Annotations can be created and modified using the respective editor commands. The framework provides three different kinds of annotations depending on the kind of the multi-variant model element the respective annotation refers to. As shown in Fig. 3, coloring is used to differentiate these different kinds. To this end, the following general classification is performed:

- *Visibilities of Objects and Values* A single object or value is annotated by means of a boolean expression that indicates its visibility. The element is present in the product if the expression evaluates to true for the respective feature configuration and vice versa. In the shown example, the class object is bound to the feature *Whole* and the properties are bound to the features *ChildOne* and *ChildTwo*, respectively. If the respective feature is set, the corresponding annotation evaluates to true and the object is visible in the derived product.
- *Visibilities of Optional Elements* This category captures optional model elements in general. Optional fragments may comprise numerous artifacts as objects, links, values, or keywords. For a single optional unit, the annotation is a boolean expression that indicates its visibility. The element is present in the product if the expression evaluates to true for the respective feature configuration and vice versa. In the example, the optional fragment for the superclass is bound to the optional feature *Generalization*. Furthermore, the initializer expression of the first property is optional and bound to the feature *InitValue*. The code fragments are visible in the editor—and thus, the child objects are contained in the domain model—if the respective feature is set.
- *Elementary Values of Attributes* In addition to annotations posing visibilities, values can be annotated with references to feature attributes (from the feature model). In this case, a value in the domain model is bound to a feature attribute. The multi-variant domain model stores the default value depending on the respective type of the attribute. When a product is derived, values for the feature attributes are provided by the feature configuration, as well. In the depicted model, the concrete integer value that constitutes the initial value of the first property is bound to the feature attribute *number*. In the multi-variant model, the default value 0 is stored.

For annotations describing visibilities of model elements, the principle of top-down propagation of annotations is applied. A model element is visible if and only if its annotation evaluates to true and all annotations that refer to (direct or transitive) container elements evaluate to true, as well.

Limitations of Integration

As outlined above, the context of this framework is the Eclipse modeling framework. Thus, we deal with EMF models with metamodels constituting instances of the Ecore (meta-)metamodel. As concrete syntax, merely textual syntax is provided so far. We assume that each product line implemented by this framework refers to one global feature model that is used to be referenced by annotations. A product line may comprise several domain models persisted by separated file resources.

For describing the abstract syntax of the multi-variant models, the framework requires one or more metamodels which are instances of the Ecore metamodel. Our restrictions of variability result from the assumption that the multi-variant model is a valid instance of the respective metamodel. In particular, values of single-valued properties cannot vary.

Note that no more technical restrictions with respect to the metamodels are present. Furthermore, the derived single variants can be reused by other modeling tools or code generators which are not aware of any product line context. Also the configuration of the editor is conceptually separated from the product line context. As a consequence, variability does not have to be taken into account when designing the (domain-specific) language with its syntax and semantics. In particular, artifacts from existing (single-variant) languages can be implemented. All in all, the editor framework may be flexibly applied to pretty arbitrary textual languages even without any product line context.

Example

This section describes an example use case of the framework. The projectional editor has been configured for the textual language ALF. It is used to implement the *Calculator* product line. Note that in this section, we only consider the role of the domain engineer and the related workflow. We assume that the respective ALF editor was already configured by an editor developer.

Figure 4 depicts the feature model of the *Calculator* product line in textual notation. The feature *Calculator*

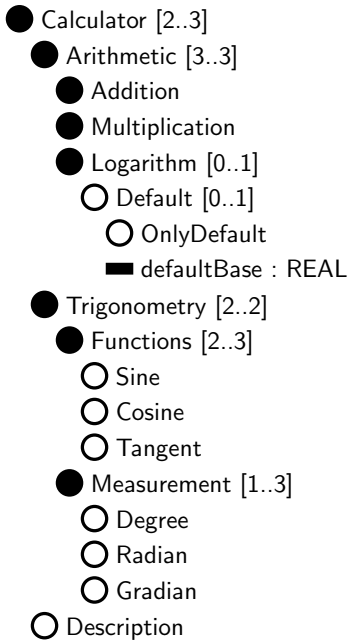

Fig. 4 Feature model of the *Calculator* example in tree notation. Mandatory features are marked by filled circles and optional ones by hollow circles. A filled rectangle represents a feature attribute. For each parent feature, the respective selection range referring to its child features is notated after the name of the feature

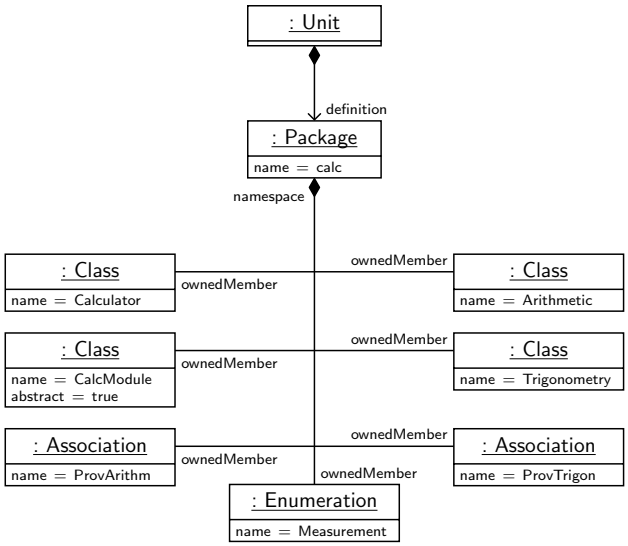

Fig. 5 The initial domain model of the example

constitutes the root feature. It contains the mandatory features *Arithmetic* and *Trigonometry* as well as the optional feature *Description*. The feature *Arithmetic* contains the usual operations represented by the mandatory child features *Addition*, *Multiplication*, and *Logarithm*. The value of a logarithm is specified by an expression and a base value (two parameters). The optional child feature *Default* provides operations for computing logarithms for a certain default base value (only one parameter). In addition, if only default logarithms are supposed to be considered, the optional feature *OnlyDefault* may be selected. For the default base value, the real attribute *defaultBase* is provided.

The feature *Trigonometry* consists of the mandatory features *Functions* and *Measurement*. Representing the provided functions, the child features *Sine*, *Cosine*, and *Tangent* are present which are all optional. The selection range of the parent feature *Functions* requires a selection of at least two child features. For the measurement of the angles, the optional features *Degree*, *Radian*, and *Gradian* are considered. In this case, the selection range of the parent feature *Measurement* requires a selection of at least one child feature which describes the behavior of an inclusive OR-group of features. Note that for simplification reasons, only a pretty small subset of possible operations is captured by the product line.

The product line comprises one ALF model as its single domain model. Fig. 5 shows the initial domain model the framework is applied to. The domain model has an ALF root package that contains several classes, associations, and an enumeration implementing the functionality. Each representation model refers to one packaged element; we focus upon the ALF class *Arithmectic*.

Within the editor, the initially empty class is extended with domain model elements which is presented below. After that, annotations are inserted to get a multi-variant domain model. Note that these two steps are separated only for presentation reasons. In the tool, domain model elements and annotations my be created and modified in an arbitrary order in one integrated view of the multi-variant domain model.

Figure 6 illustrates the succeeding steps of extending the initial ALF class. First, an object is inserted for the default base value of logarithms. Using projectional commands, the ALF property (step 1 → 2) with a real literal as its initial expression (step 2 → 3) is added. Next, the operations used for calculation are inserted (steps 3 → 4 → 5). For the sake of readability, only a subset of all operations is created; furthermore, the operation bodies are left empty. The class *Arithmetic* is supposed to inherit from the abstract supertype *CalcModule* which provides an operation that returns a description of the respective module within the calculation package. To this end, a generalization object has to be inserted (steps 5 → 6 → 7). In concrete syntax, the supertype of a class is an optional fragment. This option is enabled before the generalization object is added. Finally, layout commands are used to add some extra lines (step 7 → 8).

After building the domain model, it is augmented with annotations by means of the respective editor commands. Note that in general, domain model commands and annotation commands may be intermixed freely. Figure 7 shows

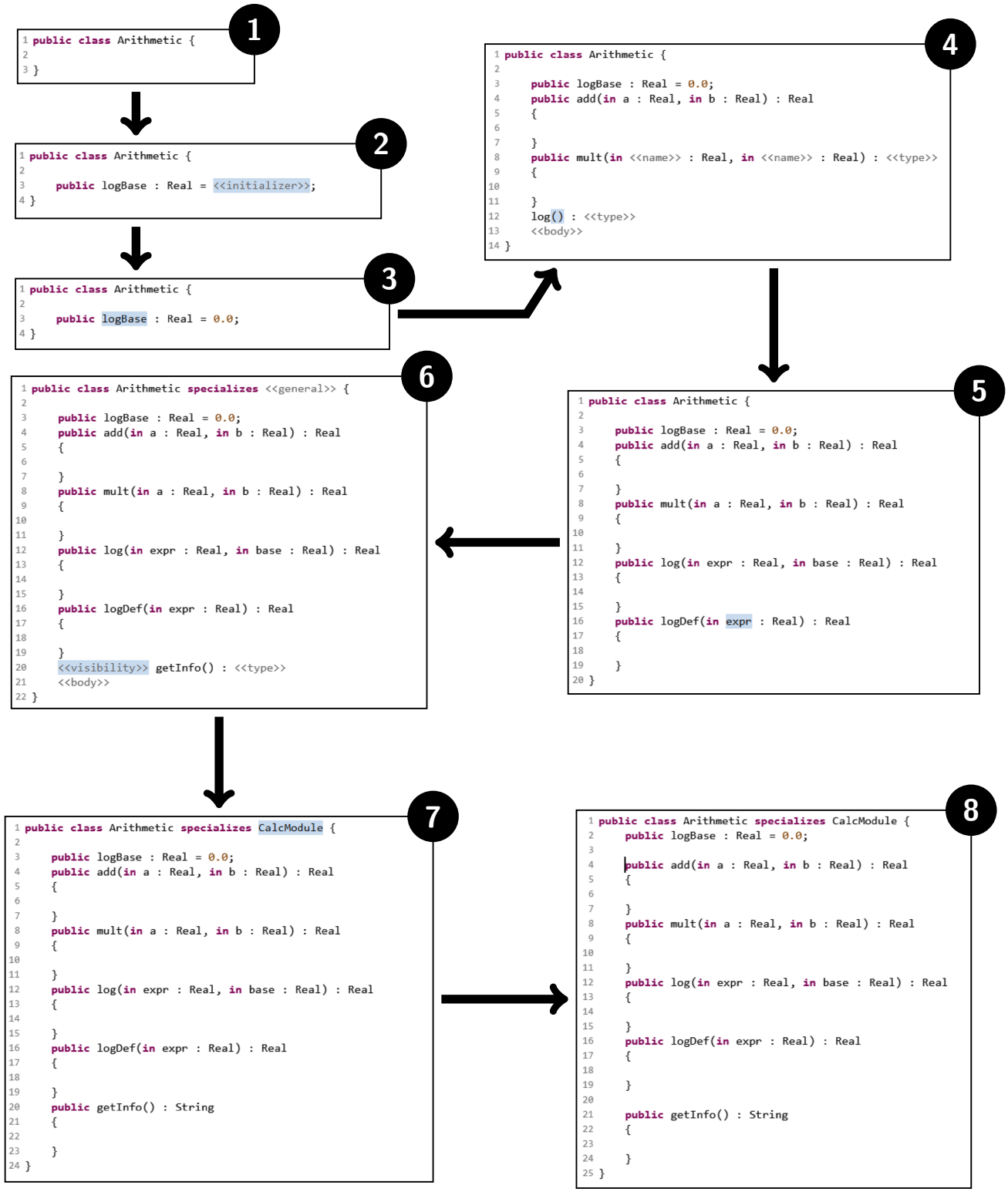

Fig. 6 The succeeding steps of adding domain model elements to the initial model. The different steps are performed by means of different projectional editor commands

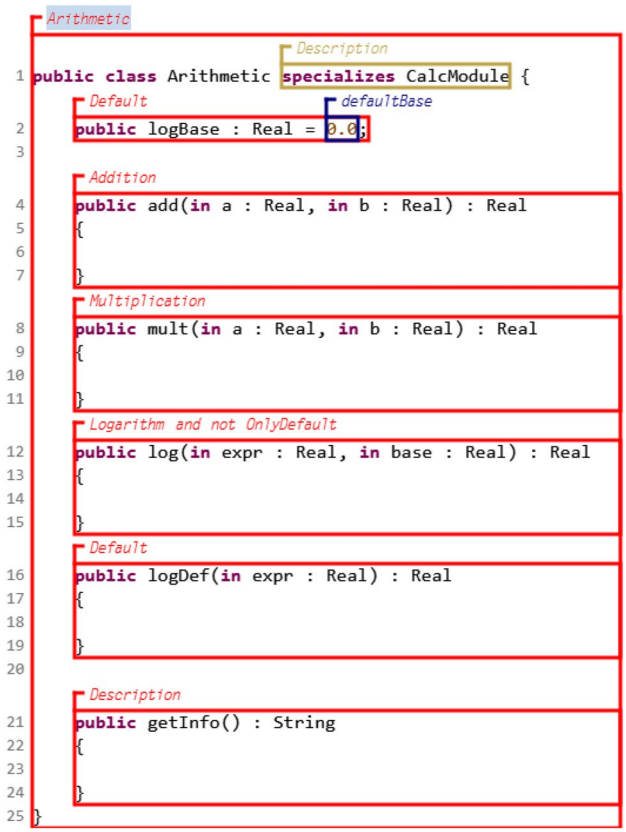

Fig. 7 The annotated ALF class *Arithmetic* in the domain model

the annotated model. For the whole class *Arithmetic*, the feature *Arithmetic* is referenced by the annotation expression. The operations *add* and *mult* are bound to the corresponding features, analogously. For computing the logarithm, two different operations are provided. The operation *log* has a parameter for the base and one for the expression; its annotation constitutes a conjunction of a reference to the feature *Logarithm* and a negation of a reference to the feature *OnlyDefault*.

In addition, the operation *logDef* only contains a parameter for the expression and employs the stored default value for the base; its annotation constitutes a reference to the feature *Default*. The same annotation is used for the property *defaultBase*. The real value within its expression is bound to the corresponding feature attribute *defaultBase*. Note that this value will be replaced with the value of the feature attribute specified in a feature configuration used to configure the domain model for application engineering.

Eventually, the annotations referring to the feature *Description* are added. First, the generalization relationship is linked. To this end, the optional fragment representing the inheritance—including the keyword *specializes* and the generalization object referencing the superclass—is annotated with the reference to the feature *Description*. Furthermore, the operation *getInfo()* is annotated analogously.

Architecture

This section describes the architecture of the framework. The technical context of our framework is the Eclipse Modeling Framework that comes along with the Ecore metamodel. Basically, each product line bases upon one global feature model. Furthermore, the traces connecting domain model elements with the respective elements from the feature model are stored within one global mapping model. Analogously to the mapping model providing links between domain model elements and their annotations in the product line context, one global correspondence model captures the connections between the models which are necessary for the internal functionality of the editor.

Overview

Figure 8 illustrates the architecture of the models and the corresponding editors as well as their dependencies. All models are based on the *ECORE* metamodel. The core editor (*EDITOR*) is a projectional editor without any product line context. The projectional editor with product line support (*PL-EDITOR*) is an extension of that editor. Each editor instance refers to exactly one domain model as an abstract syntax tree (*AST*); within the editor, either the complete model or a subtree is presented. There are not any assumptions about the respective metamodel; the domain metamodel (AST^M) may be arbirary.

The traces that link the domain model elements with artifacts from the feature model are persisted in the (product line) mapping model (*PL-MAPP*). In general, the mapping model contains mapping elements for domain model elements contained in several abstract syntax trees. Each annotation is contained in an adequate mapping element and comprises a subtree that constitutes the structure of the respective annotation expression. Both the metamodel of the feature models ($FEAT^M$) and the metamodel of the mapping models ($PL\text{-}MAPP^M$) are generic and fixed, internal artifacts of the framework. For creating and modifying the feature model, an extra editor is used. Currently, the default EMF tree editor is employed; future work will provide for a comfortable projectional editor.

While the mapping model connects the domain model with the feature model in the product line context, the correspondence model (*CORR*) serves as the central model that combines the models that are necessary for the internal editor functionality. The editor instances visualize the representation models (*REPR*) that are referenced by the correspondence model and connected to the abstract syntax

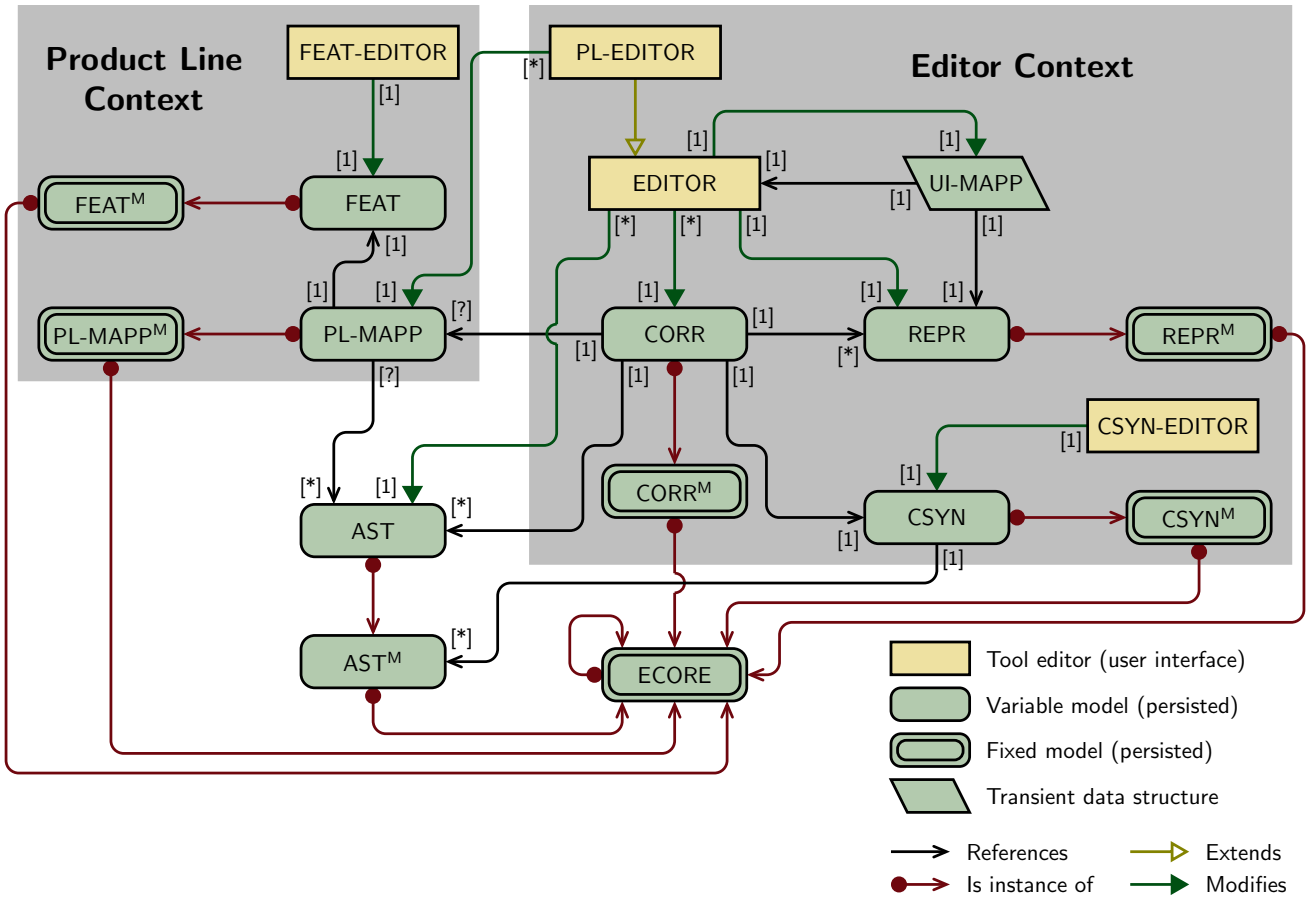

Fig. 8 Megamodel describing the architecture of the framework comprising the different models within one product line and their relations. In case of the dependencies *References* and *Modifies*, the UML multiplicities [1] (exactly one element) and [*] (arbitrarily many elements) are used; the multiplicity [?] describes optional single elements (in UML *0..1*) depending on the fact whether the extended editor is used (including product line support) or not [10]

trees as well as the model containing the concrete syntax definition (*CSYN*). In addition, in case of the extended editor within product line context, the correspondence model refers to the mapping model; consequently, internal editor correspondences and product line mappings are conceptually and physically separated. All the respective metamodels for the correspondence models ($CORR^M$), the representation models ($REPR^M$), and the syntax definitions ($CSYN^M$) are fixed, internal artifacts. When the representation model is used by the extended editor including product line support, it contains elements for annotations and their contents; furthermore, the correspondence model refers to these objects by means of respective correspondence objects for the expression objects within annotations.

Each editor instance refers to one representation model from which the presentation is derived. The representation model comprises model elements such as blocks, line, and cells; analogously, the editor pane visualizes geometrical objects (rectangles and labels) corresponding to the abstract model elements. To bundle the traces between representation elements in the representation model (*REPR*) and presentation elements created by the editor (*EDITOR*), an internal, transient data structure (*UI-MAPP*) is employed; these transient traces are not stored when the editor is closed. The concrete syntax definition model (*CSYN*) is constructed from a set of textually notated projectional rules that has been defined by the editor developer using an extra text editor (*CSYN-EDITOR*).

All in all, the architecture allows for a flexible information exchange between the involved models. For a representation element that is selected within the editor, the respective internal correspondence element is detected in order to get access to the represented domain model element, the referenced product line mapping element (if any) as well as the applied projection rule. The interconnected system of models establishes the logical basis for the functionality of the different editor commands with and without product line support.

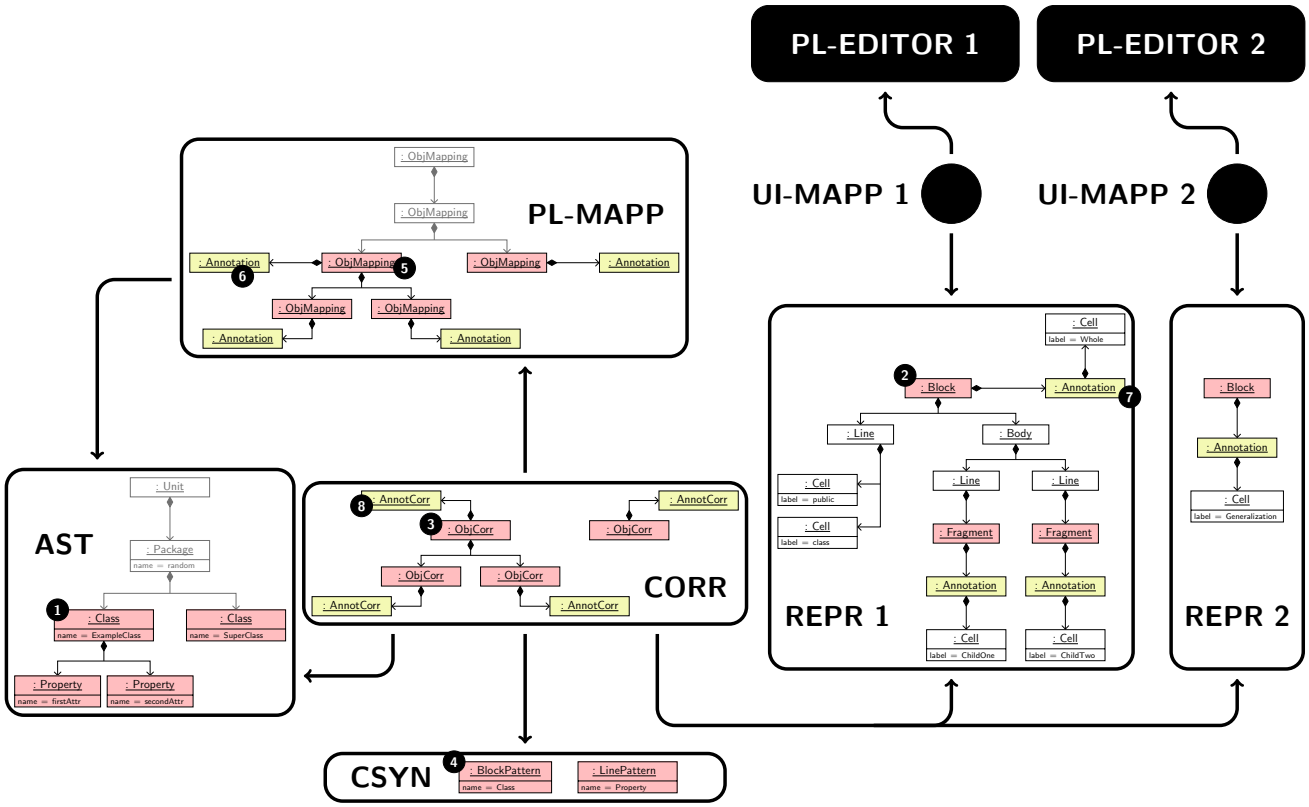

Fig. 9 An exemplary system of models with inter-model dependencies (cutout). It describes the simple example shown in Fig. 3. AST objects that are visible within the editors, their representation elements, and related elements have a red filling while annotations and their corresponding elements are marked by a yellow filling. AST objects which are not visualized by the editors and the respective mapping elements are grayed out. Diamond arrows stand for (direct or transitive) containment relations between model elements

Example Scenario

In order to depict an exemplary system of models according to this architecture, we refer to the very small example shown in Section 3. Figure 9 illustrates the system of the involved models for a cutout of the domain model which contains the classes *ExampleClass* and *SuperClass* as well as two properties; for the sake of readability, only inter-model references and no cross references between model elements are visualized. As each ALF class within the domain model (*AST*) is shown in an own editor instance, two representation models (*REPR 1* and *REPR 2*) are present. Both representation models possess blocks as their root elements which represent the ALF classes.

The correspondence model (*CORR*) provides an object correspondence for each object that is represented by a representation model and visualized in an editor. The correspondence element connects the representation element as well as the domain model element and links to the respective projection rule within the syntax definition model (*CSYN*) as well as the product line mapping element that is contained in the *PL-MAPP* model. In this case, the ALF class *Example* (1) is represented by a root block (2). This relationship is stored by an object correspondence (3) which links to the projectional rule for ALF classes (4) and the product line mapping element (5). Container objects of the ALF classes—e.g., the ALF package—are not presented in the editor; thus, neither representation elements nor correspondence elements are present for the respective container elements in the domain model and their mapping elements in the product line mapping model.

In general, the containment hierarchy of the domain model is also applied for the correspondence model and the representation models. For instance, in case of the class *ExampleClass* that contains the property *firstAttr* within the domain model, the representation element for the property is a transitive child element of the representation element for the class and the correspondence element for the property is contained in the correspondence element for the class. Furthermore, this containment structure is also visible in the mapping model; more details about the product line mapping models are provided by Section 6.

The annotations refer to features contained in the feature model. For the expressions, the product line mapping model contains respective elements which are children of the respective mapping elements. Besides objects representing

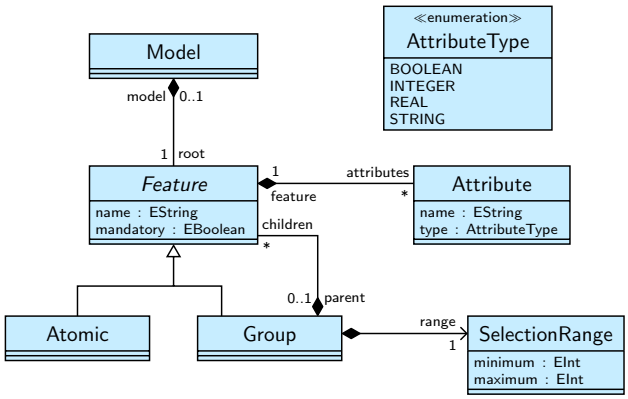

Fig. 10 The metamodel of feature models [10]

domain model elements, the representation models exhibit objects representing annotations and their cells. In this example, for the annotation *Whole* (6), a representation object (7) is provided as well as an annotation correspondence element (8). In case annotations are hidden in the editor, this circumstance is persisted by a boolean flag within the representation model.

Each representation model is presented by one specific editor where the traces between representation elements and presentation elements within the editor pane are persisted by a specific transient data structure (*UI-MAPP*). While the representation models contain geometrical objects for blocks, bodies, lines, cells, and annotations, the editor consists of corresponding graphical elements. In case of hidden annotations, there are representation objects within the model that do not have corresponding elements presented by the editor.

Mapping Model and Annotations

In this section, we describe some details about implementing variability in the domain models with our approach. After an overview of the metamodel for feature models, the product line mapping models are considered which serve as the connection between domain models and feature model.

Feature Models

The metamodel for feature models is depicted in Fig. 10. A feature model constitutes a tree of features. Each feature exhibits a name and can be mandatory which enforces that a valid feature configuration must mark the feature as selected. The tree of features is augmented with named feature attributes for which concrete values are set in the feature configuration. Each feature may possess arbitrarily many attributes for boolean, integer, real, and string values.

Each feature is annexed to a selection range that indicates how many child features have to be selected at least (minimum number) and can be selected at most (maximum number). The maximum number must be larger than or equal to the minimum number. Consequently, the minimum number must be larger than or equal to the number of mandatory child features. Analogously, the maximum number must be less than or equal to the number of all child features regardless whether mandatory or not. This concept of selection ranges constitutes a flexible mechanism that allows for realizing many restrictions including the semantics of OR- and XOR-groups which are commonly supported by feature models.

The tree structure of the feature model implicates several dependencies between the features; due to the containment relationship, the selection of a child feature enforces the selection of its enclosing parent feature. Furthermore, the selection ranges provide some restrictions among sibling features. Additional dependencies may be implemented precisely using an expression language with the common logical operators.

Mapping Models

The features and their attributes contained in the feature model are referenced by annotations that augment the domain model. Annotations are stored in the mapping model that connects feature model elements and domain model elements. As described in Section 5, the mapping model which captures annotations in logical expressions and is specific to the product line is conceptually separated from the correspondence model that constitutes the internal, central model of the editor in order to provide its functionality. Since annotations are visualized together with domain model elements within one integrated view, the domain engineer is not directly exposed to the internal structure of the mapping model.

The metamodel of the mapping model is depicted in Fig. 11. A mapping model is a tree that contains different *Mapping* instances as its elements. The annotations referring to domain model elements are stored as subtrees within the respective mappings. As described in Section 3, different kinds of annotations are provided by the editor. To this end, the mapping model contains different kinds of mappings depending on the domain model element the annotation refers to.

- All *ObjectMapping* instances correspond to objects within the domain model, i.e., arbitrary *EObject* instances. Annotations within object mappings represent annotations of objects in the domain model.
- As direct child objects of object mappings, *PropertyMapping* instances refer to their structural features

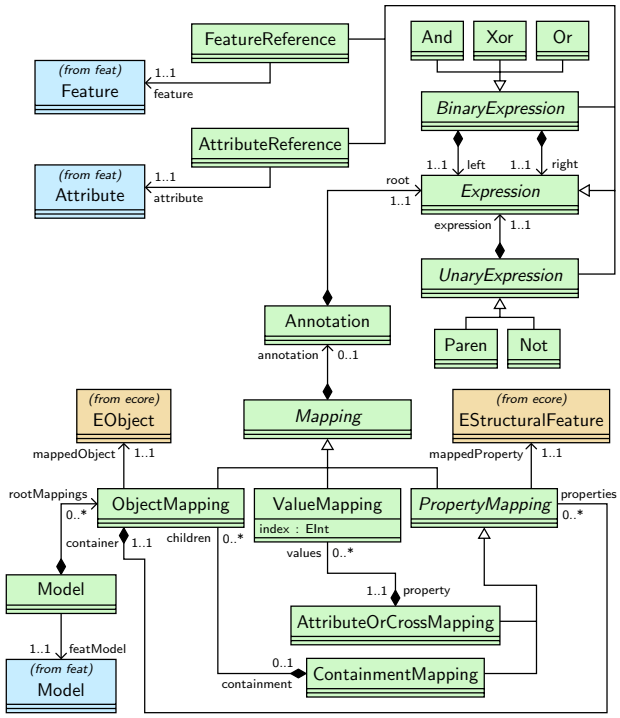

Fig. 11 The metamodel of product line mapping models [10]

(attributes and references), i.e., *EStructuralFeatures* instances that are provided by the respective *EClass* instances. Annotations within property mappings represent annotated options in the domain model.

- While object mappings correspond to objects, *ValueMappings* relate to values of attributes or links of cross references. Annotations within value mappings represent annotations of values of objects in the domain model.

The mapping elements and their containment relationships within the mapping model form a tree the structure of which reflects the corresponding containment hierarchy within the domain model. Direct child mappings of object mappings always constitute property mappings. The property mappings refer to the structural features that are provided by the object which is referenced by the container object mapping. The child objects—in case of containment references—or the values and links—in case of attributes and cross references—in the domain model are represented by object mappings and value mappings which are the direct child mappings of the respective property mappings. In order to distinguish the value mappings standing for a collection of different values or links of a multi-valued attribute or cross reference, value mappings exhibit index values describing the position in the respective collection in the domain model. Note that in case of multi-valued structural features, EMF provides ordered collections.

The analogous structure of domain model and mapping model leads to a simple strategy of locating mapping elements by employing the location of referenced elements in the domain model. With respect to modifying the mapping model, the principle of *lazy creation and deletion* is applied: When a new annotation is created by a respective editor command, only the mapping elements are constructed which are necessary to internally create and store the annotation; an annotation needs the mapping element the annotation is directly contained in as well as all recursively containing mappings up to the mapping referring to the domain model root. Furthermore, when an annotation is deleted, its referencing mapping element is not removed from the mapping model directly; rather, mapping elements are deleted if and only if the corresponding domain model elements are deleted, as well.

Annotations are stored as *Annotation* instances each of which contains a subtree that corresponds to the logical structure of the respective boolean expression. The *Expression* object that poses the root of the annotation expression is directly referenced by the *Annotation* object. Subexpressions may be nested arbitrarily respecting the operator precedences. References to the features and their attributes constitute the atomic expressions and therefore the leaf objects of the subtree.

Exemplary Mapping Model

A cutout of the internal structure of the mapping model from the *Calculator* example in Section 4 is shown in Fig. 12. The different annotations are subtrees of different mapping objects. A path of containment relations within the domain model leads to an alternating sequence of object mappings and containment mappings within the mapping model. For instance, the ALF class *Arithmetic* is contained in an ALF package. The object mapping for the package contains a containment mapping object for its containment reference *ownedMember* that also comprises the classes within the package. This containment mapping contains the object mapping for the class.

The root object mapping refers to the ALF unit element which is the root of the domain model. The annotations of objects are child elements of the respective object mappings. For instance, the annotation *Arithmetic* is contained in the object mapping referring to the ALF class of the same name. In case of annotations of optional elements, the property mappings that refer to the respective structural features are employed. In this example, the generalization dependency is annotated; therefore, the annotation is a child element of the containment mapping for the generalization containment reference within the object mapping for the surrounding ALF class. The annotation that references the feature attribute is contained in the value mapping that corresponds to the

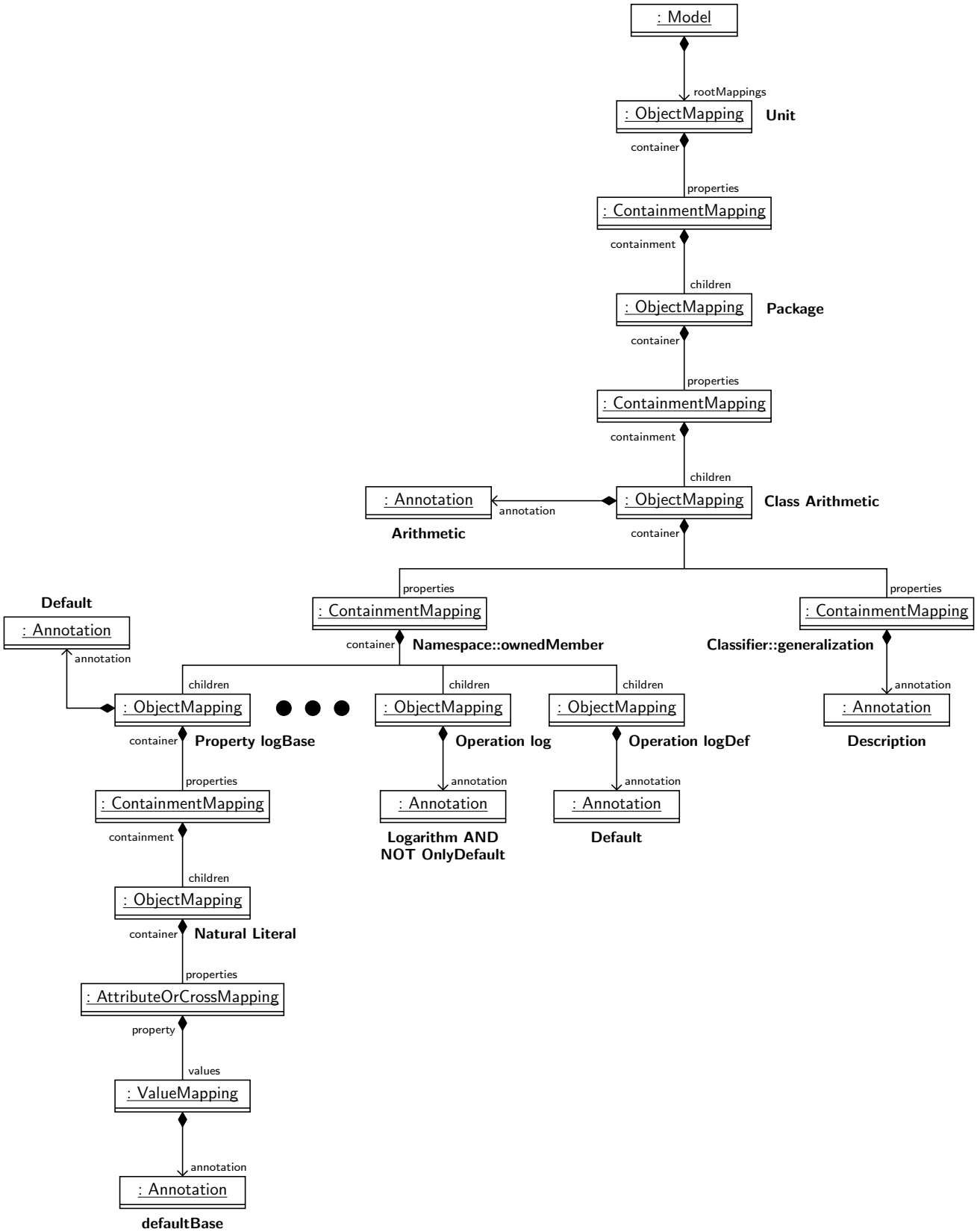

Fig. 12 The internal structure of the mapping model of the *Calculator* product line shown in Fig. 7. The referenced AST objects and the annotation expressions are notated next to the model objects

respective value. The value mapping is contained in the property mapping for the value within the object mapping which refers to the natural literal.

Note that all leaf mappings contain annotations. In this state, the mapping model does not comprise mapping objects for ALF parameters in the domain model as they do not exhibit annotations. If, for example, the annotation for the operation *log* was removed, the annotation object would be deleted without any impact on the mapping object. Rather, the object mapping would be deleted if the ALF operation was completely removed.

Conclusion

We presented a generic framework for constructing projectional multi-variant editors that are based on feature models for modeling variability. In this paper, we applied our framework to an exemplary product line including a concise domain model and a few features. Our approach employs negative variability by using feature expressions as annotations of domain model elements. The editor can be configured for arbitrary textual languages with respective metamodels. Both domain model and annotations are visualized within an integrated view including a clear optical separation between both kinds of artifacts. Within the editor, variability information may be shown and hidden in the editor in a pretty flexible way by means of respective commands.

Projectional multi-variant model editors have not purely been designed for model-driven engineering; they have also been realized with model-driven engineering. Therefore, projectional multi-variant model editors pose a complex use case for applying model-driven engineering. The internal architecture of multi-variant model editors (cf. Section 5) has been designed by means of megamodels describing the relationships between different models; to this end, we devised an extra notation for megamodels. One important design concept of the architecture is the separation of different concerns: The models that are necessary internally in order to assure the functionality of the editor are strictly separated from the models within the product line context.

The work presented here is still ongoing. Future work will provide for configuring multi-variant models by means of total or partial feature configurations. In this context, ensuring well-formedness of configured domain models constitutes a significant challenge that will be addressed along the lines of our previous work on FAMILE [8]. Configurations of feature and domain models is essential for application engineering. Nevertheless, it poses an important research area for domain engineering, as well. Total or partial feature configurations may be used to derive previews of configured products by coloring and eliding. In addition, these configured views could support filtered editing that is well-known in the context of variation control systems; since the underlying artifact is the multi-variant model and the editor commands ensure its correctness, the common view-update problems of variation control systems are not relevant for our approach.

All in all, our framework allows for a user-friendly domain engineering process. To evaluate our approach, we plan a more practical example product line with a more complex feature model including several dependencies between the features. By employing various metrics, we will analyse how the configured multi-variant model editors cope with larger feature models with numerous constraints, a higher number of annotated elements, more complex annotations, etc. As an appropriate and intuitive textual language in the world of modeling, we want to adhere to ALF as the underlying language for the projectional editor.

Funding Open Access funding enabled and organized by Projekt DEAL.

Availability of Data and Material, Code Availability Currently not available.

Declarations

Conflict of Interest The authors declare no competing interests.

References

1. Völter M, Stahl T, Bettin J, Haase A, Helsen S. Model-driven software development: technology, engineering, management. Hoboken: Wiley; 2006.
2. Steinberg D, Budinsky F, Paternostro M, Merks E. EMF eclipse modeling framework. 2nd ed. In: The Eclipse Series. Boston: Addison-Wesley; 2009.
3. Völter M, Siegmund J, Berger T, Kolb B. Towards user-friendly projectional editors, vol. 8706 of Lecture Notes in Computer Science. Berlin: Springer; 2014. p. 41–61. https://doi.org/10.1007/978-3-319-11245-9_3.
4. Berger T, Voelter M, Jensen HP, Dangprasert T, Siegmund J, Tichy M, Bodden E, Kuhrmann M, Wagner S, Steghöfer J, editors. Efficiency of projectional editing. In: Tichy M, Bodden E, Kuhrmann M, Wagner S, Steghöfer J, editors. Software engineering und software management 2018, Fachtagung des GI-Fachbereichs

Softwaretechnik, SE 2018, Ulm, Germany, 5-9 März 2018, vol. P-279 of LNI, 153–154 (Gesellschaft für Informatik, 2018); 2018. https://dl.gi.de/20.500.12116/16335.
5. Pohl K, Böckle G, van der Linden F. Software product line engineering—foundations, principles, and techniques. Berlin: Springer; 2005. https://doi.org/10.1007/3-540-28901-1.
6. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University; 1990.
7. Heidenreich F, Kopcsek J, Wende C, Schäfer W, Dwyer MB, Gruhn V. editors. Featuremapper: mapping features to models. In: Schäfer W, Dwyer MB, Gruhn V. editors. 30th International Conference on software engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, Companion Volume, pp. 943–944, ACM; 2008. https://doi.org/10.1145/1370175.1370199.
8. Buchmann T, Schwägerl, F, Störrle H, et al. editors. FAMILE: tool support for evolving model-driven product lines. In: Störrle H, et al. editors. Joint proceedings of co-located events at the 8th European conference on modelling foundations and applications (ECMFA 2012), CEUR WS, Technical University of Denmark (DTU), Kongens Lyngby; 2012. p. 59–62.
9. Schwägerl F, Westfechtel B. Integrated revision and variation control for evolving model-driven software product lines. Softw Syst Model. 2019;18(6):3373–420. https://doi.org/10.1007/s10270-019-00722-3.
10. Schröpfer J, Buchmann T, Westfechtel B, Hammoudi S, Pires LF, Seidewitz E, Soley R. editors. A framework for projectional multi-variant model editors. In: Hammoudi S, Pires LF, Seidewitz E, Soley R. editors. Proceedings of the 9th International Conference on model-driven engineering and software development, MODELSWARD 2021, Online Streaming, February 8–10, 2021, pp. 294–305, SCITEPRESS; 2021. https://doi.org/10.5220/0010310102940305.
11. Schröpfer J, Buchmann T, Westfechtel B, Hammoudi S, Pires LF, Selic B, editors. A generic projectional editor for EMF models. In: Hammoudi S, Pires LF, Selic B, editors. Proceedings of the 8th international conference on model-driven engineering and software development (MODELSWARD 2020). INSTICC, SciTePress; 2020. p. 381–92.
12. OMG. Action language for foundational UML (Alf). formal/2017-07-04. Needham: Object Management Group; 2017.
13. Guermazi S, et al. Executable modeling with fuml and alf in papyrus: tooling and experiments. In: Mayerhofer T, Langer P, Seidewitz E, Gray J, editors. Proceedings of the 1st international workshop on executable modeling co-located with ACM/IEEE 18th international conference on model driven engineering languages and systems (MODELS 2015), Ottawa, Canada, 27 September 2015, vol. 1560 of CEUR workshop proceedings, 3–8, CEUR-WS.org; 2015. http://ceur-ws.org/Vol-1560/paper1.pdf.
14. Linsbauer L, Schwägerl F, Berger T, Grünbacher P. Concepts of variation control systems. J Syst Softw. 2021;171:25. https://doi.org/10.1016/j.jss.2020.110796.
15. Clements P, Northrop L. Software product lines: practices and patterns. Boston: Addison-Wesley; 2001.
16. Kang KC, et al. Form: A feature-oriented reuse method with domain-specific reference architectures. Ann Softw Eng. 1998;5:143–68.
17. Quinton C, et al. Evolution in dynamic software product lines. J Softw Evol Process. 2021. https://doi.org/10.1002/smr.2293.
18. Michelon GK et al. Locating feature revisions in software systems evolving in space and time. In: Lopez-Herrejon, RE. editor. SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19–23, 2020, Volume A, pp. 14:1–14:11, ACM; 2020. https://doi.org/10.1145/3382025.3414954.
19. Mauro J, Nieke M, Seidl C, Yu IC ter Beek MH et al. editors. Anomaly detection and explanation in context-aware software product lines. In: Beek MH et al. editors. Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25–29, 2017, 18–21, ACM; 2017. https://doi.org/10.1145/3109729.3109752.
20. Batory DS, Obbink JH, Pohl K. editors. Feature models, grammars, and propositional formulas. In: Obbink JH, Pohl K. editors. Proceedings of the 9th International Software Product Line Conference (SPLC'05), Vol. 3714 of Lecture Notes in Computer Science, 7–20, Springer Verlag, Rennes, France; 2005.
21. Schobbens P, Heymans P, Trigaux J. Feature diagrams: a survey and a formal semantics. In: 14th international conference on requirements engineering (RE 2006). IEEE; 2006. p. 136–45.
22. Czarnecki K, Helsen S, Eisenecker UW. Formalizing cardinality-based feature models and their specialization. Softw Process Improv Pract. 2005;10(1):7–29.
23. Apel S, Kästner C, Lengauer C. FeatureHouse: language-independent, automated software composition. In: 31st international conference on software engineering (ICSE 2009). IEEE; 2009. p. 221–31.
24. Schaefer I, Müller P, Schaefer I. editors. A personal history of delta modelling. In: Müller, P, Schaefer I. editors. Principled software development—essays dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday, pp. 241–250. Springer; 2018. https://doi.org/10.1007/978-3-319-98047-8_15.
25. Apel S, Kästner C. Virtual separation of concerns—a second chance for preprocessors. J Object Technol. 2009;8(6):59–78. https://doi.org/10.5381/jot.2009.8.6.c5.
26. Buchmann T, Schwägerl F, Schaefer I, Thüm T. editors. Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In: Schaefer I, Thüm T. editors. 4th International Workshop on feature-oriented software development, FOSD '12, Dresden, Germany - September 24—25, 2012, pp. 37–44. ACM; 2012. https://doi.org/10.1145/2377816.2377822.
27. Heidenreich F. Towards systematic ensuring well-formedness of software product lines. Denver: ACM; 2009. p. 69–74.
28. Hunsen C, et al. Preprocessor-based variability in open-source and industrial software systems: an empirical study. Empir Softw Eng. 2016;21(2):449–82. https://doi.org/10.1007/s10664-015-9360-1.
29. Mukelabai M et al. Multi-view editing of software product lines with people. In: Chaudron M, Crnkovic I, Chechik M, Harman M. editors. Proceedings of the 40th International Conference on software engineering: companion proceedings, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018, pp. 81–84. ACM; 2018. https://doi.org/10.1145/3183440.3183499.