Softw Syst Model (2014) 13:1495–1527
DOI 10.1007/s10270-012-0305-5

REGULAR PAPER

Mapping feature models onto domain models: ensuring consistency of configured domain models

Thomas Buchmann · Bernhard Westfechtel

Received: 27 June 2011 / Revised: 14 September 2012 / Accepted: 13 November 2012 / Published online: 19 December 2012

Abstract We present an approach to model-driven software product line engineering which is based on feature models and domain models. A feature model describes both common and varying properties of the instances of a software product line. The domain model is composed of a structural model (package and class diagrams) and a behavioral model (story diagrams). Features are mapped onto the domain model by annotating elements of the domain model with features. An element of a domain model is specific to the features included in its feature annotation. An instance of the product line is defined by a set of selected features (a feature configuration). A configuration of the domain model is built by excluding all elements whose feature set is not included in the feature configuration. To ensure consistency of the configured domain model, we define constraints on the annotations of inter-dependent domain model elements. These constraints guarantee that a model element may be selected only when the model elements are also included on which it depends. Violations of dependency constraints may be removed automatically with the help of an error repair tool which propagates features to dependent model elements.

Keywords Model-driven software product line engineering · Feature models · Domain models · Feature mappings · Dependency constraints

Communicated by Prof. Alfonso Pierantonio.

T. Buchmann (✉) · B. Westfechtel
Applied Computer Science I, University of Bayreuth,
95440 Bayreuth, Germany
e-mail: thomas.buchmann@uni-bayreuth.de

1 Introduction

Model-driven software product line engineering [40] aims at increasing the productivity of software engineers in two ways. First, *software product line engineering* [14,44,52] deals with the systematic development of products belonging to a common system family. Rather than developing each instance of a product line from scratch, reusable software artifacts are created such that each product may be composed from a library of components. Second, *model-driven software engineering* [23,51] puts strong emphasis on the development of high-level models rather than on the source code. Ideally, software engineers operate only on the level of executable models such that there is no need to inspect or edit the actual source code (if any). By combining both disciplines, productivity may be improved by both reuse and code generation.

This paper contributes to the discipline of model-driven software product line engineering by presenting an integrated environment which is based on feature models and executable domain models. A *feature model* [36] is used to define the capabilities and the variation points of a product line. A *domain model* is developed for the product line which covers all product variants. The domain model defines both structure and behavior. The structural model consists of package and class diagrams. The behavioral model is composed of a set of story diagrams (resembling UML interaction overview diagrams) implementing the operations of classes defined in class diagrams. Feature model and domain model are connected by *feature annotations*, which map domain model elements to the sets of product variants in which they occur. An instance of the product line is defined by a feature configuration, which is composed of all features selected for the product variant to be built. Using the feature configuration, a variant of the domain model is configured by assembling all domain elements whose feature sets are included in the feature configuration.

The core contribution of our work consists in the *consistent mapping* of the *feature model* to the *domain model*. Our goal is to ensure the consistency of configured domain models with respect to the underlying metamodel. To this end, we define *dependency constraints* on the mappings of feature models to domain models. These constraints demand that a depending model element may be included only into a configuration of the domain model when the model elements on which it depends are selected, as well. Mapping constraints are defined as conditions on the feature annotations of inter-dependent model elements. Constraint violations may be removed by an error repair tool which propagates features to dependent elements. In addition, tool support also covers other kinds of errors such as, e.g., the use of undeclared features in feature annotations.

In this paper, the concept of dependency constraints is applied to domain models composed of package, class, and story diagrams. The constraints are defined in terms of the underlying metamodels. However, the concept as such is *general* and not specific to the metamodels covered in our current tool support. For a new metamodel, the notion of dependency has to be defined with respect to this metamodel, resulting in a set of metamodel-specific constraints.

Software product lines can either be constructed around a common core or a set of superimposed variants is filtered and unused artefacts are removed during product derivation. While the first one is called positive variability, the latter is referred to as negative variability. Table 3 compares different approaches based on these concepts. Our approach is based on negative variability, as the domain model created with Fujaba contains all variants. A specific product is created by removing all unused model artefacts from the domain model.

The rest of this paper is structured as follows: Sect. 2 provides an overview of our approach; Sect. 3 develops the theory underlying the consistent mapping of the feature model onto the domain model; Sect. 4 describes the tools which we implemented; Sect. 5 introduces a case study which was used to apply and evaluate our approach; Sect. 6 discusses achievements and limitations; Sect. 7 compares related work; and Sect. 8 concludes the paper.

2 Overview

2.1 Tool chain

For model-driven software product line engineering, we implemented an environment called *MODPL* which is visualized in Fig. 1. As far as possible, we tried to reuse existing and widespread tools. For feature modeling, we used the *FeaturePlugin* developed by Czarnecki and Antkiewicz [1]. FeaturePlugin is a stand-alone tool which supports cardinality-based feature modeling [17]. The tool provides a tree editor for creating a feature model and an interactive

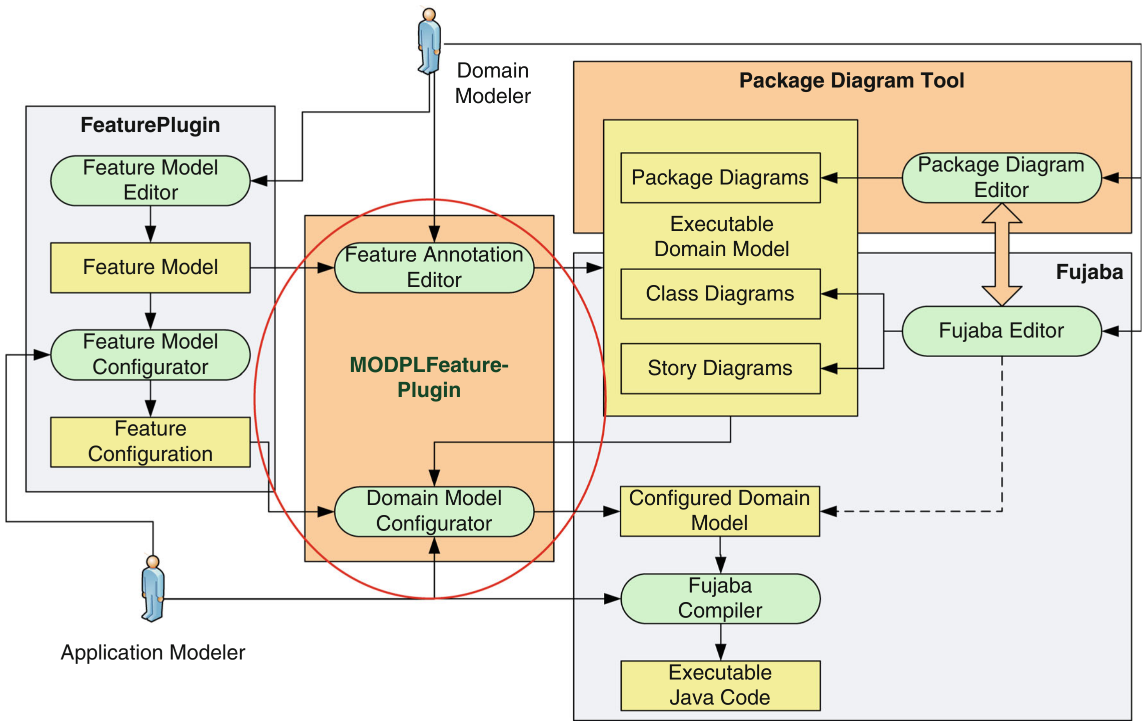


Fig. 1 The MODPL environment for model-driven engineering of software product lines

configurator for deriving a consistent feature configuration from the model.

For domain modeling, we selected Fujaba [61], which supports class diagrams for structural modeling and story diagrams for behavioral modeling. Fujaba was chosen because the domain model is executable: from the domain model, the Fujaba compiler generates fully functional Java code. In contrast to most other tools, Fujaba does not merely generate implementation frames for method bodies. Rather, complete executable Java code is generated from a story diagram (see Sect. 2.2.2) acting as a behavioral model of a method defined in a class diagram. A yearly workshop series (Fujaba Days) has been established which provides a platform for presenting and discussing tool developments in and applications of Fujaba. The reader is referred to [47] for an overview of the Fujaba tool suite being developed at multiple sites. The web page http://www.fujaba.de gives detailed information on Fujaba, including a comprehensive list of publications.

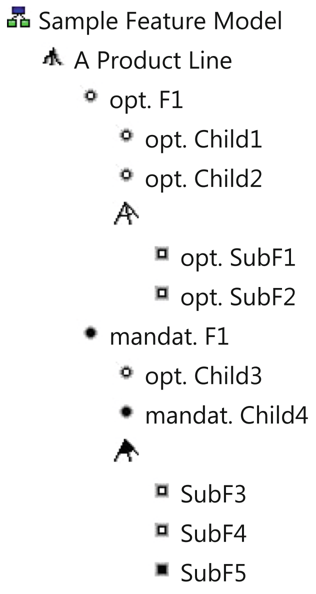
Apart from these tools, the overall MODPL environment includes further tools implemented by us (orange rectangles in the diagram). Since Fujaba, like other common UML tools, does not provide adequate support for modeling-in-the-large, we developed a *package diagram editor* [8,12] which is based on the UML2 package concept. The main focus of the current paper (red ellipsis in the figure) lies on *MODPLFeaturePlugin* [5,6], which provides the bridge connecting feature models to domain models.

The tool chain is used as follows: in *domain engineering*, the domain modeler creates a feature model describing common and varying properties of instances of the product line with the help of FeaturePlugin. Furthermore, he uses the package diagram editor and the Fujaba editor to create a domain model as a superimposition of all product variants. Domain model elements are connected to features by means of the feature annotation editor. In *application engineering*, the application modeler creates a feature configuration, which includes all features of the product variant to be constructed. Subsequently, he uses the domain model configurator to configure the domain model by dismissing all domain model elements which do not belong to the specified product variant. Finally, the Fujaba compiler converts the configured domain model into executable Java code.

2.2 Models

2.2.1 Feature model

A *feature model* consists of one or more *feature diagrams*, which are organized hierarchically. From the original approach presented in [36], several variants have emerged, including *cardinality-based feature modeling* [17], implemented in an Eclipse plug-in called *FeaturePlugin* [1].



```
Sample Feature Model
  A Product Line
    opt. F1
      opt. Child1
      opt. Child2
      
        opt. SubF1
        opt. SubF2
    mandat. F1
      opt. Child3
      mandat. Child4
      
        SubF3
        SubF4
        SubF5
```

Fig. 2 A simple example of a feature model created with FeaturePlugin

Figure 2 shows a feature diagram created with FeaturePlugin. Features are organized into a tree. A Product Line constitutes the *root feature*, which is decomposed into *subfeatures* opt. F1 and mand. F1. These features are *optional* (unfilled icon) and *mandatory* (filled icon), respectively: if the parent feature is selected, an optional/mandatory subfeature may/must be selected, as well. opt. F1 is decomposed into optional subfeatures opt. Child1 and opt. Child2. Furthermore, there is a *feature group* grouping two subfeatures opt. SubF1 and opt. SubF2. An unfilled fork denotes an *exclusive-or selection*, i.e., exactly one of the child features (unfilled squares) has to be selected. Feature mandat. F1 is refined into an optional child feature opt. Child3 and a mandatory child feature mandat. Child4. Furthermore, there is a feature group with *inclusive-or semantics* (filled fork), i.e., at least one feature of the group must be selected. Since SubF5 is mandatory, it is always selected when its parent feature is selected. SubF3 and SubF4 are optional and may or may not be selected.

The feature diagram defines *constraints* on feature selections: a child feature may be selected only if its parent has been selected, exactly one feature of an exclusive-or group has to be selected, etc. Further, "context sensitive" constraints may be expressed such as "feature f_1 excludes/requires f_2". By means of these constraints, *feature interactions* may be specified.

In cardinality-based feature modeling, *cardinalities* may be specified for feature groups (represented by forks); this (meta-)feature is not shown in the feature diagram of Fig. 2. The cardinality of a feature group defines the minimum and the maximum number of elements that may be selected in

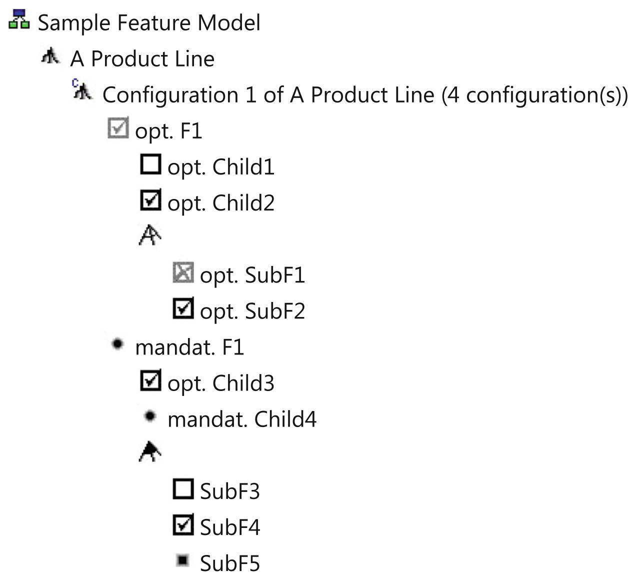

Sample Feature Model
A Product Line
Configuration 1 of A Product Line (4 configuration(s))
opt. F1
opt. Child1
opt. Child2
opt. SubF1
opt. SubF2
mandat. F1
opt. Child3
mandat. Child4
SubF3
SubF4
SubF5

Fig. 3 A sample configuration created with FeaturePlugin

a feature configuration.[1] In addition, cardinality-based feature modeling allows to specify cardinalities for individual features. In this case, a feature may be *instantiated* multiple times in a feature configuration.

In our approach, we allow for cardinalities attached to (inclusive-or) feature groups, but we do not make use of feature instantiations. Thus, a *feature configuration* consists of a subset of all features which are defined in a feature model. Configuring a feature model is reduced to a selection process and does not comprise feature instantiations. A configuration is called *consistent* if it satisfies the constraints defined in the feature model. The tool FeaturePlugin supports the construction of consistent configurations, as will be shown below.

In Fig. 3, a consistent feature configuration is displayed which is based upon the features defined in the feature model. Selected features are depicted by hooked squares (opt. Child2). Optional features which have not been selected are shown as empty squares (opt. Child1). In exclusive-or groups, excluded features are displayed as crossed squares (opt. SubF1). The selection of a subfeature implies the selection of its parent feature (if any) in order to maintain consistency. In Fig. 2, opt. F1 was selected automatically because opt. Child2 is part of the current configuration. Furthermore, if a feature in an exclusive-or group is selected, all competing features in the same group are excluded automatically. In the sample configuration, opt. SubF1 and opt. SubF2 are excluded and selected, respectively.

[1] Cardinalities may be specified only for inclusive-or groups. An exclusive-or group always has the cardinality [1:1].

2.2.2 Domain model

The domain model consists of models of different levels of granularity. *Package diagrams* are used on a coarse-grained level to model the basic structure of the software system. Packages are refined into *class diagrams*. Finally, methods of (passive) classes are specified by story diagrams. We assume the reader to be familiar with UML2 package and class diagrams [43]; story diagrams are described below.

A *story diagram* [22] realizes exactly one method of some class, and it may use classes, attributes and methods from multiple class diagrams. Story diagrams are close to interaction overview diagrams in UML2. However, they were designed with a different purpose, namely, to provide an executable behavior rather than a description of interactions among objects. Within story diagrams, activities and transitions are used to model the control flow. Fujaba supports two different kinds of activities: (1) statement activities, allowing the modeler to use source code fragments that are merged 1:1 with the generated code and (2) story activities. A story activity contains a story pattern and describes a graph transformation rule where both the left and the right hand side of the rule are shown in one diagram. Additionally, collaboration calls can be used on objects and constraints can be specified that are evaluated at runtime. A graph transformation rule is only applied in case the constraints are satisfied. On the level of control structures, story diagrams support sequences, branches, and loops. Furthermore, a story activity can be marked as a *"for each"* activity: While an ordinary story activity is executed only once (on *one* match of the story pattern), a "for each" activity is carried out on *each* match of the pattern.

Figure 4 shows a simple example of a story diagram containing some of the constructs mentioned above. It represents the implementation of method process of class B. The first activity checks whether the attribute processed has a given value. If the attribute assertion fails, the method ends. Otherwise, the next activity is processed. This activity checks if no object of the type X is associated with the current object. If this condition holds, an object of the type Y is created and associated with the this object. Finally, the attribute processed is modified and the stop activity is reached.

Story diagrams constitute the added value of Fujaba compared to other UML CASE tools since they allow for the generation of fully executable source code. Their expressive power comes from the story patterns, which describe complex operations with a graphical and intuitive notation. Some more advanced examples of story diagrams will follow in Sect. 4, where we employ story diagrams internally for model-driven development of the MODPLFeaturePlugin tool. For a formal definition of the syntax and semantics of story diagrams, the reader is referred to [61]. In this paper, we will be concerned only with the abstract

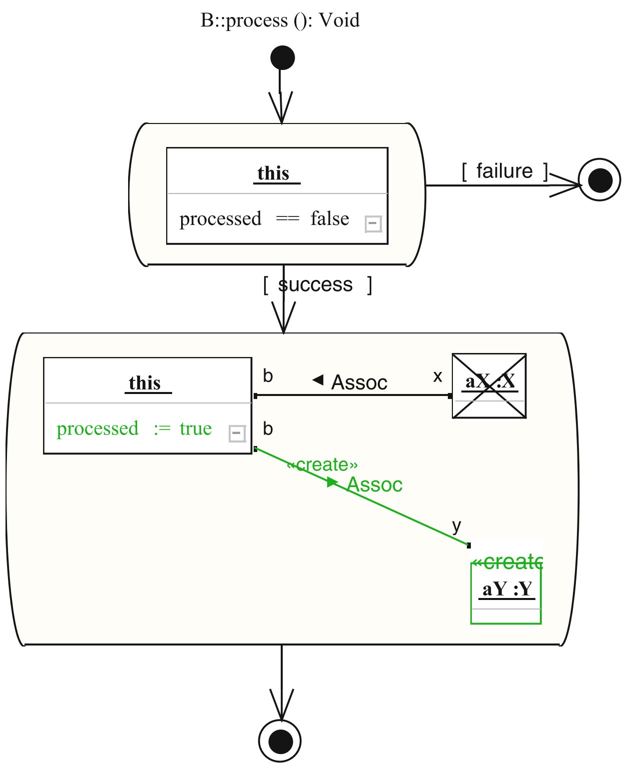


Fig. 4 A simple example of a story diagram

syntax of story diagrams, which will be introduced later (in Sect. 3).

3 Mapping features to models

This section deals with the *mapping* of elements contained in the feature model to domain model elements. This mapping defines for each model element in which product variants it will be visible. To this end, each model element may be decorated with a *feature annotation*—a set of features from the feature model. From a feature configuration, which defines the set of features of the product variant to be built, and the annotated domain model a *configured domain model* is constructed. The configured model is composed of all domain model elements whose feature set is included in the feature set of the configuration. If a domain element is annotated with some feature f which is not part of the feature configuration, it is specific to an excluded feature. As a consequence, it is not inserted into the configured domain model.

The configured domain model has to be *consistent*, i.e., it needs to be a valid instance of the underlying metamodel. Unfortunately, inadequate feature annotations may easily result in inconsistent configured domain models. For example, an association might be selected whose ends were excluded from the configured domain model. To ensure consistency of the configured domain model, *constraints* on the

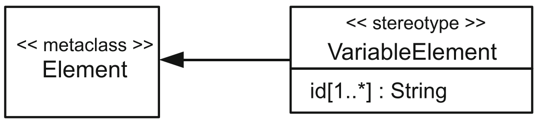


Fig. 5 UML profile to map features to a domain model element

feature annotations of inter-dependent model elements have to be enforced. In our example, the feature set of the association must include the feature sets of the association ends. In this way, it is guaranteed that an association is selected only with its association ends.

In this section, we will formally define the mapping of features to model elements as well as the consistency rules to be applied to this mapping. To this end, the *UML profile* mechanism is used to extend the domain model with feature annotations. Furthermore, the consistency rules are defined with the help of *OCL constraints*. In this way, the definitions are kept independent of tool-specific peculiarities. Furthermore, we separate the formal definition of the mapping and the consistency rules clearly from their implementation, which will be described in Sect. 4.

3.1 Mapping definition

In order to map features to domain model elements, we defined a profile. This profile consists of a *stereotype* called VariableElement which contains a set of strings denoting features from the feature model (Fig. 5). The stereotype extends the metaclass Element.

In the following, the term *feature annotation* is used when the stereotype VariableElement is assigned to a model element in order to establish a mapping between this element and corresponding features in the feature model. A model element which has no annotation is treated in the same way as a model element being annotated with an empty feature set.

Formally, feature annotations may be defined by a function mapping domain elements to sets of features.

Definition 1 (*Feature annotation function*) Let F and DM denote a set of features (from the feature model) and a domain model (consisting of a set of model elements), respectively. A *feature annotation function* is a function

$$f_a : \mathrm{DM} \rightarrow 2^F \tag{1}$$

The function f_a assigns to each model element $e \in \mathrm{DM}$ a (potentially empty) set of features $f_a(e) \subseteq F$.[2] $f_a(e)$ is called the *feature annotation* of e.

From the feature annotation function, a *feature annotation relation* $R_a \subseteq \mathrm{DM} \times F$ may be derived as follows:

$$(e, f) \in R_a \Leftrightarrow f \in f_a(e). \tag{2}$$

[2] 2^F denotes the powerset of F.

The feature annotation relation is an $m : n$ relation, i.e., a given domain model element may be decorated with multiple features, and a given feature may be used to decorate multiple domain model elements.

A model element annotated with a set of features $f_1, \ldots, f_n$ is specific to *all* of these features. The model element is included into a domain configuration (a configuration of the domain model) if and only if the underlying feature configuration includes all of the features $f_1, \ldots, f_n$. Thus, the feature annotation of some model element controls its *visibility*. This behavior is formalized by a domain configuration function.

Definition 2 (*Domain configuration function*) Let F and DM denote a set of features (from the feature model) and a domain model, respectively. A *domain configuration function for* DM is a function

$$f_{\mathrm{dc}} : 2^F \rightarrow 2^{\mathrm{DM}} \tag{3}$$

For some feature configuration $C \subseteq F$, the value of f_{dc}—the *domain configuration* DC for C—is defined as follows:

$$f_{\mathrm{dc}}(C) = \{e \in \mathrm{DM} | f_{\mathrm{a}}(e) \subseteq C\} \tag{4}$$

Note that elements e with $f_{\mathrm{a}}(e) = \emptyset$ are *globally visible*, i.e., they are contained in each domain configuration.

3.2 Mapping constraints

Mapping constraints are conditions which refer the mapping of the feature model onto the domain model (formalized by the feature annotation function). Table 1 lists the most important constraints violations which are taken care of by the MODPLFeaturePlugin. The first column denotes different types of inconsistencies, the second column describes the respective actions for handling inconsistencies.

Feature not declared This inconsistency is *avoided* by offering only features from the feature model when a model element is going to be annotated.
Feature not used If there is no model element which has been annotated with a given feature f, this may indicate that f has not been implemented yet. In this case, a *warning* is displayed.

Table 1 Handling of inconsistencies

Inconsistency	Action
Feature not declared	Avoidance
Feature not used	Warning
Feature annotation not satisfiable	Error
Dependency violation	Repair

Feature annotation not satisfiable A feature annotation is not satisfiable if it contains mutually exclusive features f_i and f_j from the feature model. As a consequence, the annotation is flagged with an *error*.
Dependency violation A model element e_1 is said to depend on a model element e_2 when the latter must be present whenever the former is part of a domain configuration. A dependency violation occurs when e_1 is visible in some configuration but e_2 is excluded. Inconsistencies of this type are handled by *repair actions*.

In the following, we will focus on the most interesting of these inconsistencies, namely *dependency violations*. Supposing that the annotated domain model is syntactically correct, all errors in a configured domain model result from the removal of annotated elements (this is quite similar to the deletion of model elements). In order to avoid syntactical errors resulting from removing annotated elements, we have to make sure that depending model elements are annotated in a "sufficiently restrictive way". This leads to the following rule:

If model element e_1 depends on another model element e_2, e_1 may be visible only when e_2 is visible, as well.

This rule is formalized as follows:

Definition 3 (*Dependency constraint*) Let DM be a domain model with elements e_1, e_2 such that e_1 depends on e_2. Furthermore, let $\mathrm{DC} \subseteq \mathrm{DM}$ denote some domain configuration. For each DC, the following condition must hold:

$$e_1 \in \mathrm{DC} \Rightarrow e_2 \in \mathrm{DC}. \tag{5}$$

We may rephrase this constraint by referring to the sets of domain configurations including e_1 and e_2, respectively. Let us denote these sets by $\mathcal{DC}_1$ and $\mathcal{DC}_2$, respectively. Using these sets, Condition 5 is rewritten as follows:

$$\mathcal{DC}_1 \subseteq \mathcal{DC}_2. \tag{6}$$

These definitions are expensive to check: for each domain configuration including e_1, it has to be tested whether e_2 is included, as well. Therefore, we rephrase the dependency constraint in terms of feature annotations:

$$f_{\mathrm{a}}(e_1) \supseteq f_{\mathrm{a}}(e_2). \tag{7}$$

Please note that the set inclusion is performed in different directions in Conditions 6 and 7, respectively. The more features are assigned to some model element e, the smaller is the set of domain configurations in which e occurs.

The dependency constraint has to hold for all configured domain models. Therefore, the rule above has to be "instantiated" for the respective metamodels. In the following sections, we specify consistency rules written in OCL [42] defining dependency constraints for UML class diagrams and Fujaba's story diagrams. In the OCL rules, the generic notion of dependency is instantiated in a metamodel-specific way. For example, a component depends on its container, and an applied occurrence depends on the corresponding declaration.

Since OCL has been designed for readers who are not familiar with mathematical notation, OCL constraints tend to be verbose. Nevertheless, we decided to use OCL because it is also used for consistency constraints within the UML specification [43]. Furthermore, OCL has been applied frequently in tools for model-driven software engineering. Altogether, it is a widely adopted specification language for constraints.

3.3 Dependency constraints for class diagrams

In this section we present constraints for annotating elements of UML class diagrams. We do not strive for completeness; rather, we have selected a representative set of rules. Before presenting the OCL constraints, we start with a motivating example. Please note that the rules specified in this section differ from "regular" consistency constraints for class diagrams in terms that they take into account our stereotype which is used for feature annotations. Furthermore, the constraints presented here are transformed into automatic repair actions. While other tools which use OCL for validation purposes are able to only detect errors, our approach automatically repairs these errors without any user interaction.

Example 1 (Class diagram constraints) Figure 6 shows a simple class diagram where class B is annotated with the feature unselectedFeature. When deriving a configuration which does not contain unselectedFeature, the following syntactical errors arise:

- Source or target of generalizations are missing.
- Both associations have only one member end.
- The classes X and Y both have a role end of a nonexisting type.
- The implementation of print_K_from_A() leads to compilation errors, since class C is no longer derived indirectly from class A, and therefore cannot access the property k.
- Within the method implementation of print_I_from_B(), an access to property i contained in class B is not possible, since B does not exist.

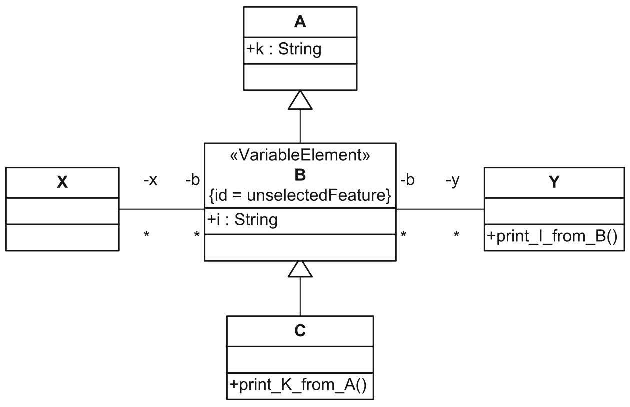


Fig. 6 Simple class diagram with a tagged class

Figure 7 shows a cutout of the UML Superstructure as far as it is relevant for the OCL constraints to be presented below. Element constitutes the root of the class hierarchy.

Constraint 1 (Owned elements) For the constraint VisibilityOfOwningElement defined in Fig. 8, we first introduce a function which returns the feature identifiers of an element. The function featureIDs() will be reused in all definitions to be presented below. Starting from the current element, all attached stereotypes are retrieved. From these, the stereotype named VariableElement is selected, whose set-valued id attribute is read.[3] The function returns the empty set if no stereotype named VariableElement is attached to the current element.

The constraint VisibilityOfOwningElement requires that the feature set of the current element includes the feature set of its owning element. In this way, it is guaranteed that if an owned element is visible, its container is visible, as well. If the current element does not have an owner, the constraint is always true because the feature set of the current element is compared against the empty set.

In the UML specification, the composition defined on Element is refined for certain subclasses, e.g., for Class. Table 2 shows a few examples of subclasses of Element and their dependent elements.

Constraint 2 (Typed elements) If a typed element is visible, its type must be visible, as well. Therefore, it is required that the feature set of the type is included by the feature set of the typed element (Fig. 9). This rule ensures that the types used, e.g., as property types or parameter types of operations are "sufficiently visible".

Constraint 3 (Generalizations) A generalization is a directed relationship from a specific to a general classifier which is owned by the specific classifier. Thus, Constraint 1 ensures that the generalization is visible only when its source (the specific classifier) is visible. In addition, the current constraint (Fig. 10) guarantees that the visibility of the generalization is also constrained by the visibility of the target (the general classifier). To this end, the feature set of the generalization must contain the feature set of the general classifier.

[3] In OCL, nested navigations result in bags rather than in sets. The predefined function asSet() is used to convert a bag into a set, removing potential duplicates (not occurring here).

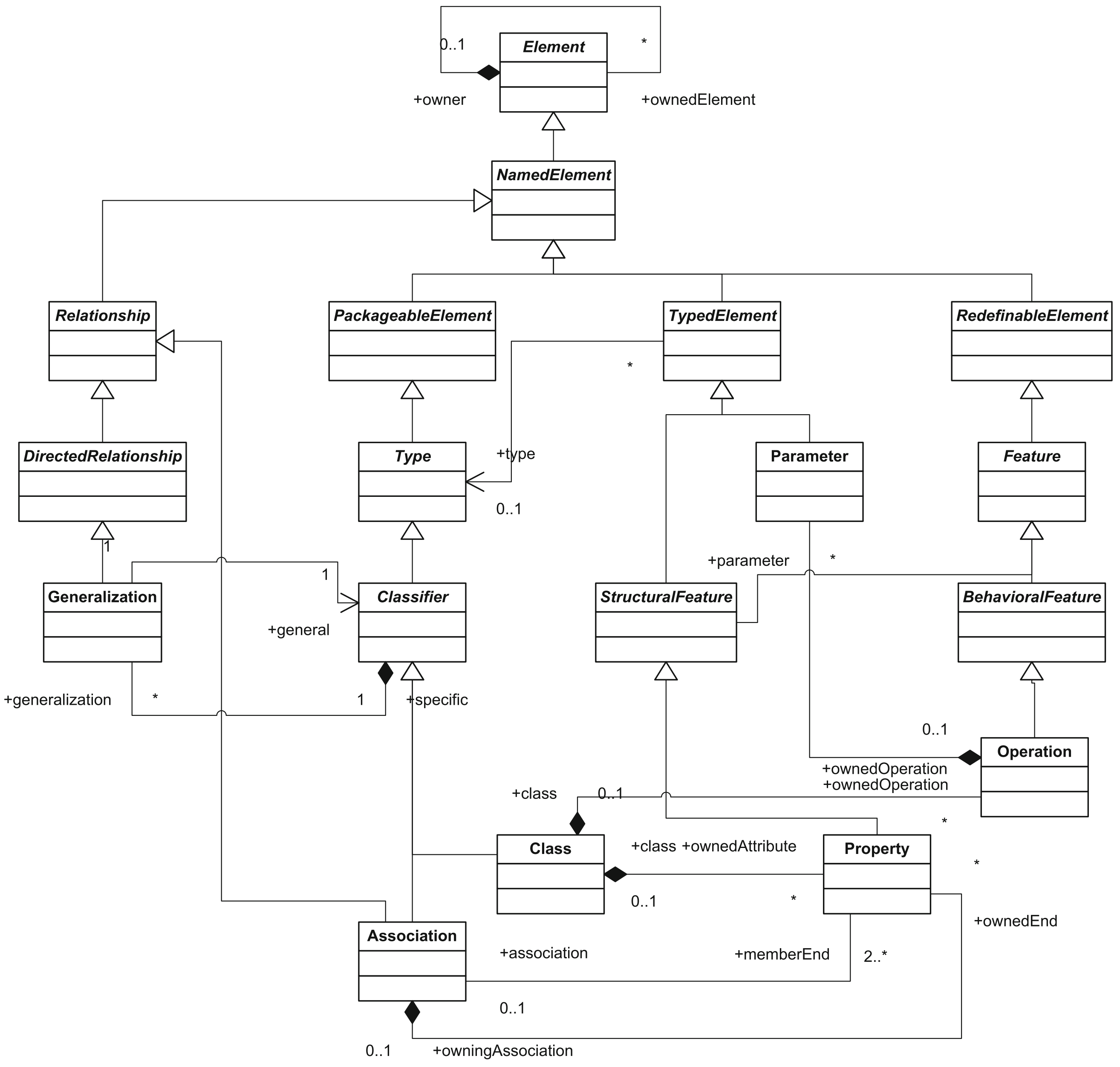


Fig. 7 Cutout showing relevant parts of the UML Superstructure

```
context Element

def: featureIds() : Set(String) =
  self.stereotype->select
    (name = 'VariableElement').id->
      asSet()

inv VisibilityOfOwningElement:
  self.featuresIds()->includesAll
    (self.owner.featureIds()->asSet())
```

Fig. 8 Owned elements

Table 2 Types of owned elements

Class	Owned elements (type)
Association	Contained rules (Constraint), outgoing imports (ElementImport or PackageImport), generalizations to super-associations (Generalization), non-navigable or *n*-ary roles (Property)
Class	Nested classes (Class), contained rules (Constraint), outgoing imports (ElementImport or PackageImport), generalizations to superclasses (Generalization), defined operations (Operation), defined attributes (Property)

```
context TypedElement

inv VisibilityOfElementType:
  self.featureIds()->includesAll
    (self.type.featureIds()->asSet())
```

Fig. 9 Typed elements

```
context Generalization

inv VisibilityOfGeneralClassifier:
  self.featureIds()->includesAll
    (self.general.featureIds()->
      asSet())
```

Fig. 10 Generalizations

```
context Association

inv VisibilityOfAllAssociationEnds:
  self.featureIds()->includesAll
    (self.memberEnd.featureIds()->
      asSet())
```

Fig. 11 Association ends

Constraint 4 (Association ends) Each association requires at least two association ends. The constraint of Fig. 11 ensures that *all* association ends are visible when an association is selected. To this end, the set of features annotating the association must contain the union of the feature sets of all association ends. Please note that this constraint could be relaxed for n-ary associations by requiring only that at least two association ends are visible when the association is selected. However, in Fujaba only binary associations are supported. Therefore, the relaxed constraint—which is more difficult to handle, e.g., in repair actions—would not make a difference for Fujaba class diagrams.

3.4 Dependency constraints for story diagrams

The following constraints are required for Fujaba's story diagrams. We will focus on constraints referring to story patterns. Further constraints are defined on the control flow level, e.g., a control flow may be visible only if its ends are visible.

Figure 12 shows a simplified cutout of the Fujaba metamodel. Please note that Fujaba is based on the UML, but still has its own extensions for behavioral modeling. The behavior of each operation can be specified by exactly one *story diagram*. Each story diagram consists of a set of activities of different kinds, e.g., statement activities or story patterns. A *story pattern* is composed of *instance specifications*. Both *objects* and *links* are instances of classifiers. Each link connects a source to a target object. An *attribute expression* is owned by an object of some story pattern. It consists of a left-hand side (an attribute of the referenced object), an operator (either a relational operator or the assignment operator), and a right-hand side specifying an attribute value. The attribute expression is connected to the property which is instantiated on its left-hand side.

Constraint 1 applies to owned elements in behavioral models, as well. For example, a story pattern is visible only when its enclosing story diagram is visible, and an object is visible only when its enclosing story pattern is visible.

In the following, we will define a few constraints on elements of story patterns in order to illustrate that the same kind of reasoning as for class diagrams may be applied to story diagrams.

Constraint 5 (Instances) If an instance specification is visible, its corresponding classifier must be visible, as well. Thus, the feature set of the instance specification must include the feature set of its classifier (Fig. 13).

Constraint 6 (Attributes) Analogously, if an attribute expression is visible, the property from which its target attribute is instantiated must be visible, as well (Fig. 14).

Constraint 7 (Links) If a link is visible, both its source and its target have to be visible, as well (Fig. 15). Otherwise, the configuration process may produce dangling links.

Constraint 8 (Visibility of attributes) Within story diagrams, attributes declared in the class diagram can be used. Inheritance ensures that attributes from superclasses can be used in respective method implementations of subclasses. Thus, syntax errors can occur if the inheritance hierarchy is broken due to filtering.

The constraint which excludes this type of error is shown at the bottom of Fig. 16. The context of the constraint is the class AttributeExpression and refers to the target of the attribute expression. First, the *owning class* of the target attribute is determined by navigating the property and the class association ends. Second, the *object class*—the class used in the declaration of the object in the story diagram—is retrieved by navigating from the attribute expression to its corresponding object and from this object to its class.[4] Third, all intermediate parent classes between the object class and the owning class are retrieved.

These definitions are used in the condition following the keyword in. The condition is an implication with the premise that the owning class differs from the object class. In the first operand of the conjunction in the conclusion, it is required that the feature set of the attribute expression includes the feature sets of all intermediate parents. Likewise, it is required that this condition also holds for all generalizations emanating from these classes or immediately from the object class

[4] Here, oclAsType() narrows the type from Classifier to Class.

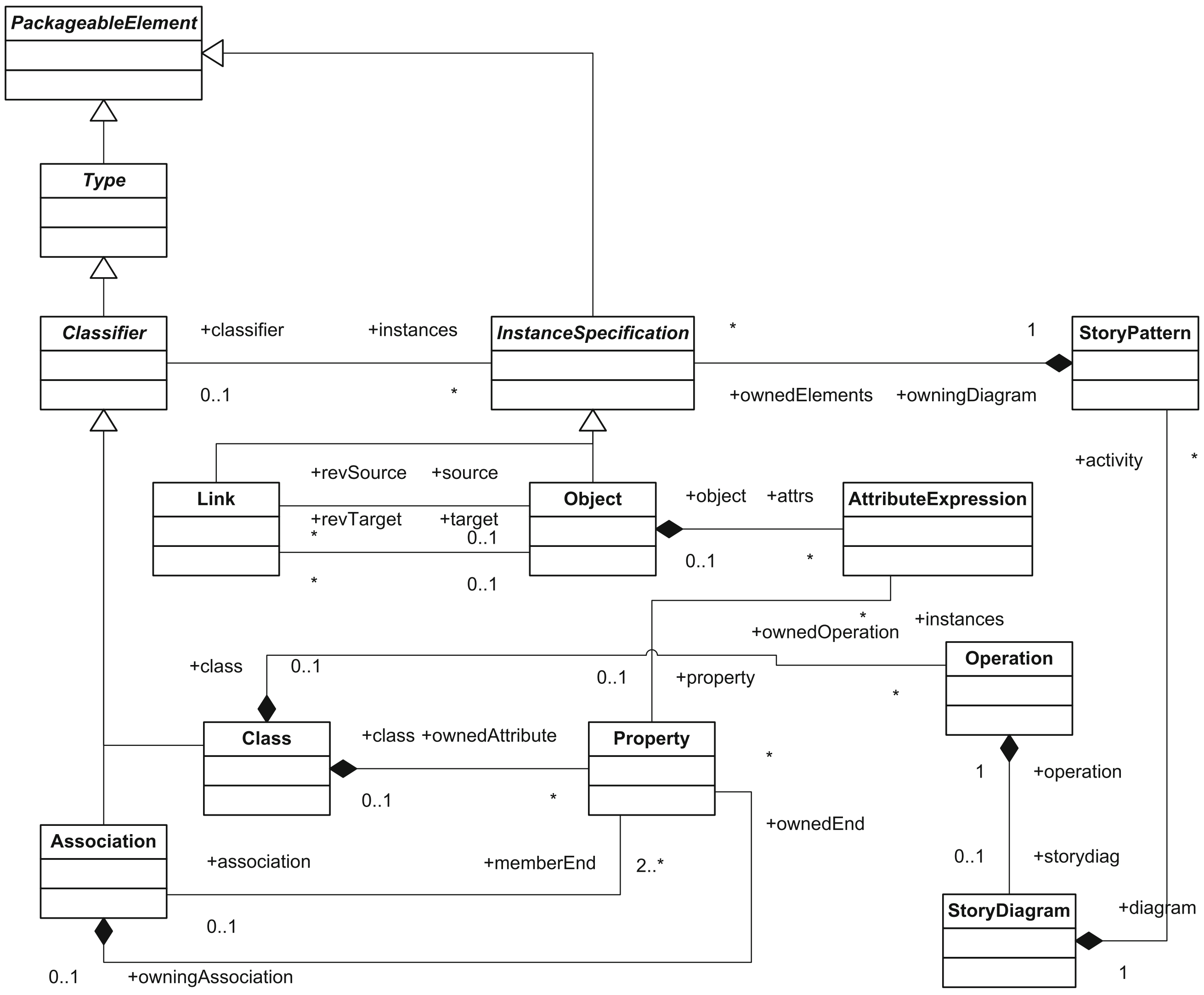


Fig. 12 Cutout showing relevant parts of the Fujaba meta model

```
context InstanceSpecification

inv VisibilityOfClassifier:
  self.featureIds()->includesAll
    (self.classifier.featureIds()->
      asSet())
```

Fig. 13 Instances

```
context AttributeExpression

inv VisibilityOfProperty:
  self.featureIds()->includesAll
    (self.property.featureIds()->
      asSet())
```

Fig. 14 Attributes

```
context Link

inv VisibilityOfEnds:
  self.featureIds()->includesAll
    (self.source.featureIds()->
      asSet())
  and
  self.featureIds()->includesAll
    (self.target.featureIds()->
      asSet())
```

Fig. 15 Links

and ending at either an intermediate parent or the owning class. Altogether, this constraint guarantees that the paths from the object class to the owning class are not filtered away when the attribute expression is visible.

In the upper part of Fig. 16, the required auxiliary function is defined in the context of Classifier. In the definition, the function allParents() from the UML 2.3 standard is used which returns the transitive closure over the generalizations starting from a given classifier. From the full transitive clo-

```
context Classifier

def: allIntermediateParents
       (parent : Classifier) :
       Set(Classifier) =
  self.allParents()->select
    (p | p.allParents()->includes
       (parent))

context AttributeExpression

inv AttributeVisible:
  let owningClass : Class =
        self.property.class,
      objectClass : Class =
        self.object.classifier.
          oclAsType(Class),
      parentClasses : Set(Classifier) =
        objectClass.
          allIntermediateParents
            (owningClass)
  in
  owningClass <> objectClass implies
    self.featureIds()->includesAll
      (parentClasses.
         featureIds()->asSet())
    and
    self.featureIds()->includesAll
      (parentClasses->including
        (objectClass).
          generalization->select
            (g |
              parentClasses->including
                (owningClass)->includes
                  (g.general)).
            featureIds()->asSet())
```

Fig. 16 Visibility of attributes

sure, those classes are selected which are located below the parent classifier.

Constraint 8 is considerably more complex than the other constraints defined above. All other constraints are derived from a single association in the underlying metamodel. In contrast, Constraint 8 is derived from a context-sensitive rule dealing with inheritance of properties: when an attribute expression is connected to a property, the property must be defined in the declared class of the object owning the attribute expression or in a (transitive) superclass of the object class.

3.5 Repair actions

Checking the dependency constraints presented in the previous sections is helpful since violations indicate that configured domain models may be inconsistent. In addition, the dependency constraints may be used proactively for *automatic repair actions*: if a dependency constraint is violated, the feature set of the dependent element is extended until it includes all features of the master element. This repair action is called *feature propagation*. Below, feature propagation is defined in a generic way, following the approach at the end of Sect. 3.2.

Definition 4 (*Feature propagation*) Let DM be a domain model with elements e_1, e_2 such that e_1 depends on e_2. Furthermore, let $f_a(e_1)$ and $f_a(e_2)$ denote their respective feature annotations (feature sets). Finally, the feature difference set Δ_f is calculated as follows:

$$\Delta_f = f_a(e_2) \setminus f_a(e_1) \tag{8}$$

If $\Delta_f \neq \emptyset$, *feature propagation* extends the feature set of e_1 as follows:

$$f_a(e_1) := f_a(e_1) \cup \Delta_f. \tag{9}$$

Definition 4 refers to a single pair of model elements. Feature propagation on the whole domain model involves a *fixed point iteration*: feature sets are gradually extended until all dependency constraints are satisfied.

Algorithm 1 (Feature propagation)
Let $g = (V, E)$ be a dependency graph for a domain model DM which is constructed as follows:

- $V = \{e | e \in DM\}$ is a set of vertices consisting of all elements of DM.
- $E \subseteq V \times V$ is a set of directed edges such that $(e_1, e_2) \in E \Leftrightarrow e_2$ depends on e_1.

The following procedures propagate features to dependent elements until all dependency constraints are satisfied:

```
procedure propagate()
begin
    for each $e \in V$ do
        propagate($e$);
    end
end
procedure propagate($e_2$ : Element)
begin
    for each $e_1 \in \{e | (e_2, e) \in E\}$ do
        $\Delta_f := f_a(e_2) \setminus f_a(e_1)$;
        if $\Delta_f \neq \emptyset$ then
            $f_a(e_1) := f_a(e_1) \cup \Delta_f$;
            propagate($e_1$);
        end
    end
end
```

Feature propagation was implemented along the lines of Algorithm 1[5] (see Sect. 4.1.3 for further details). Let us briefly discuss partial correctness, termination, and efficiency of this algorithm.

Partial correctness The outermost loop in *propagate*() calls *propagate*(e) on each element of the domain

[5] The edge direction coincides with the direction of feature propagation.

model. Recursive calls to $propagate(e_1)$ are performed as long as dependency constraints are violated. If the algorithm terminates, the domain model does not contain dependency violations any more.

Termination A recursive call $propagate(e_1)$ is performed only if $\Delta_f \neq \emptyset$, i.e., a proper extension of the feature annotation of e_1 precedes the recursive call. Since the base feature set F is finite, the feature annotation of some element e_1 may be extended only a bounded number of times. Therefore, the algorithm terminates.

Efficiency The parameterized procedure $propagate(e_2 : Element)$ is called in the outermost loop $|V|$ times. For each element e_1, the number of recursive calls on e_1 is bounded by $|F|$ since each recursive call is guarded by a proper extension of $f_a(e_1)$. Altogether, the number of calls is bounded by $|V|(1 + |F|)$.

Example 2 (Class diagram propagations) Figure 17 shows the application of the repair actions to the sample diagram given in Fig. 6 (Example 1). By applying Rule 1 (i.e., the propagation rule derived from Constraint 1), all owned elements of class B are also decorated with the stereotype VariableElement and the respective tagged value. This includes the generalization to class A, the property i, and the navigable member ends x and y of both associations. Applying Rule 3, the visibility of the generalization from class C to class B is constrained. The visibilities of the opposite ends of both associations are constrained according to Rule 2. Finally, Rule 4 is used to constrain the visibilities of both associations.

If the resulting annotated domain model is configured without selecting unselectedFeature, class C would have no outgoing generalization in the configured model. Further repair actions are offered on configured models to deal with *broken inheritance hierarchies*: In an automatic mode, the filtered class (B) is replaced by its superclass (A) while in an interactive mode, the user can decide if the inheritance to the superclass should be introduced or not.

Example 3 (Story diagram propagations) The top of Fig. 18 displays an unannotated story diagram for the method print_k_from_A() provided by class C. We assume that feature propagation has already been applied to the class diagram, resulting in the final state shown in Fig. 17. At the bottom, the result of applying feature propagation rules to the story diagram is displayed. According to Rule 5, object b is annotated because its class B has been annotated. For the same reason, both links are annotated, as well. Please note that the link between x and b has to be annotated also for another reason: According to Rule 7, a link is annotated if its ends have been annotated. The attribute expression with left-hand side i receives an annotation from its property declared in the class diagram (Rule 6). Finally, the attribute expression with left-hand side k is annotated because of a broken inheritance hierarchy (Rule 8).

4 Tool support

In the previous section, we have dealt with the mapping of feature models onto domain models at a conceptual level. The current section describes the support tools which we developed based on these concepts. As already illustrated in Fig. 1, our core contribution consists of a tool called *MODPLFeaturePlugin* [5,6]. This tool is displayed in a more detailed way in Fig. 19. It consists of two major parts:

- An extension for the Fujaba editor, enabling the user to annotate domain model elements with features and automatically applying the consistency rules presented in Sect. 3. By invoking the automatic propagation of features to dependent domain model elements, the user has full design time control of the effects of assigning features to model elements.
- A set of configuration mechanisms: (1) a graphical visualization of the chosen product configuration directly in the domain model editor, (2) the generation of a configured model, and (3) the direct generation of configured and executable source code.

4.1 MODPL feature editor

4.1.1 Implementation of feature annotations

In the previous section, we defined a UML profile for annotating domain model elements with features (Fig. 5). Unfortunately, it was not possible to implement the mapping of the feature model onto the domain model in this way. The Fujaba metamodel supports UML stereotypes, but unfortunately no tagged values can be associated to them. Therefore, we used *annotations* (structured comments) to realize the mapping between features and domain model elements. This is illustrated in Fig. 20, which shows the class diagram obtained from feature propagation in Example 2 (Fig. 17).

4.1.2 Handling inconsistencies

Following the structure of Table 1, inconsistencies are handled in our tool as follows:

Feature not declared The tool allows to annotate model elements with declared features only. To this end, a feature model is loaded. As soon as the user invokes the dialog to annotate a model element, he can only choose

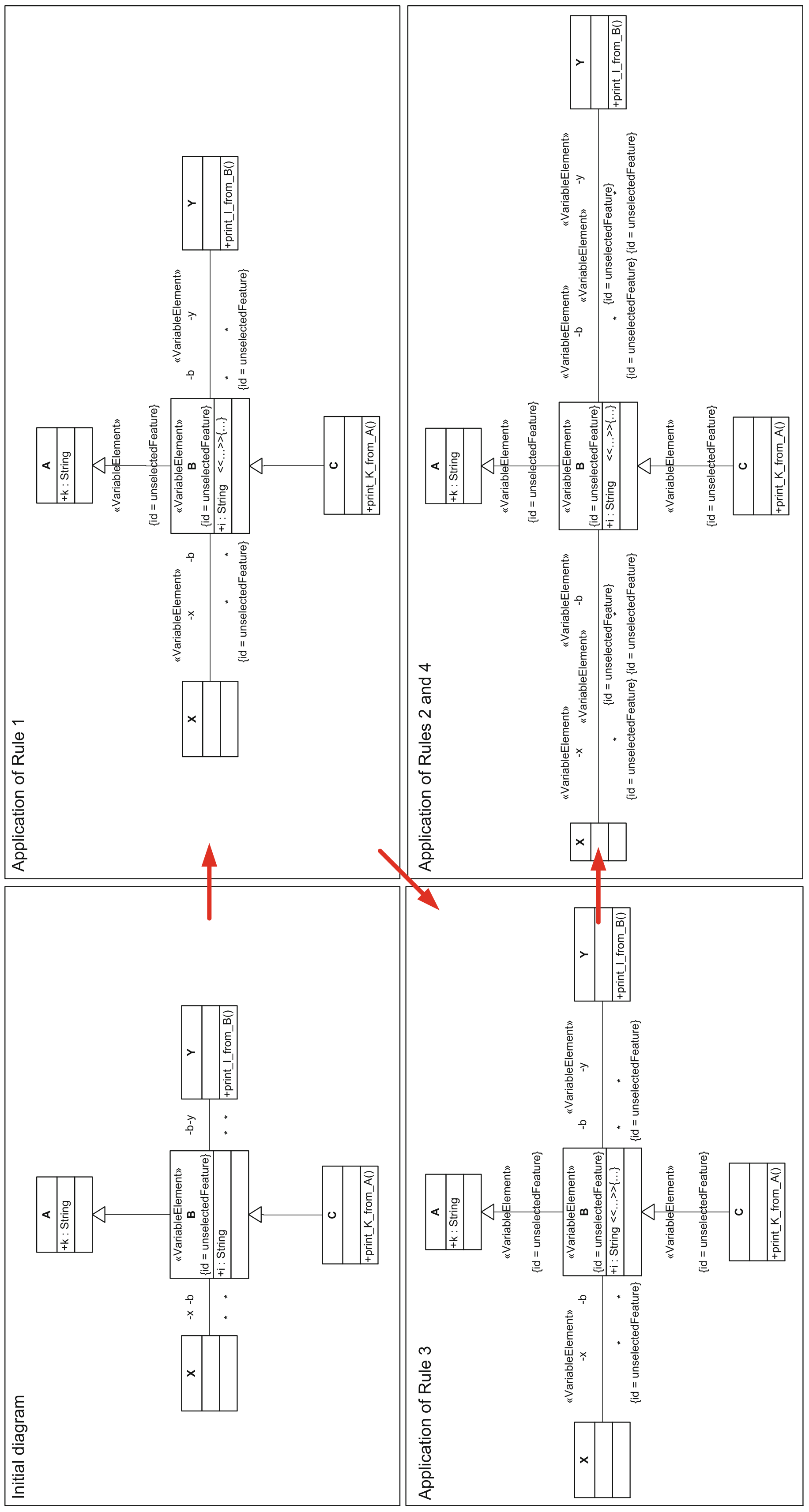


Fig. 17 Application of the propagation rules

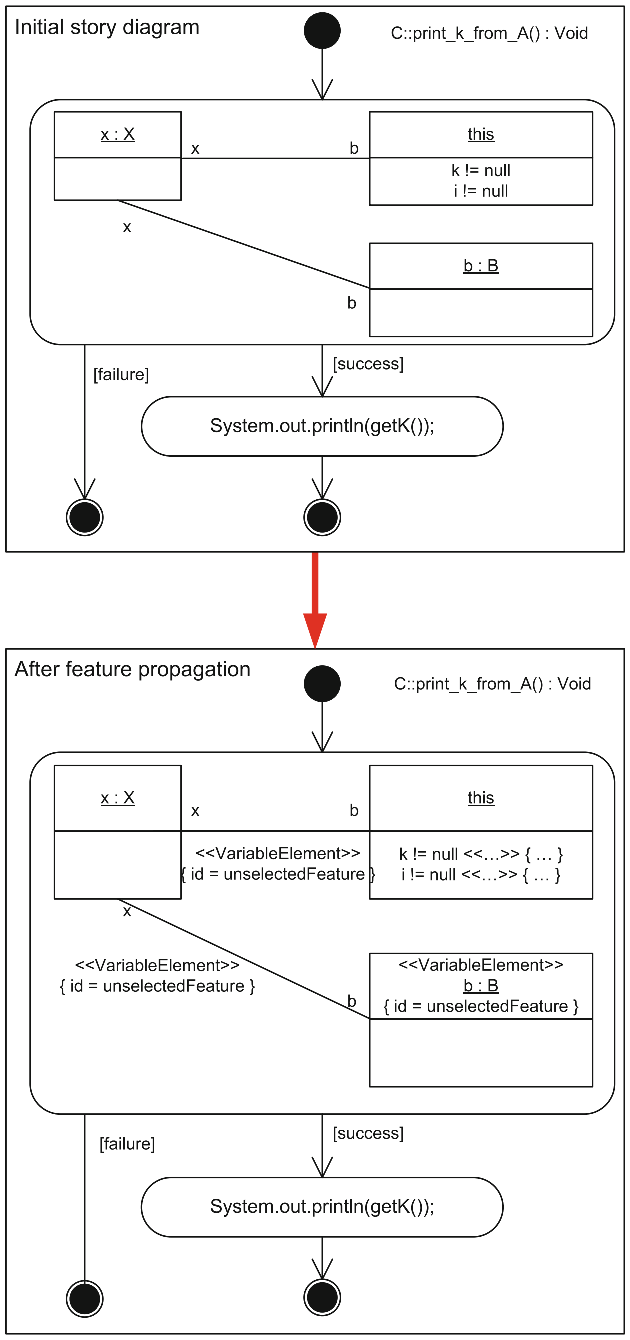

Initial story diagram
C::print_k_from_A() : Void
x : X
this
k != null
i != null
b : B
x
b
[failure]
[success]
System.out.println(getK());
After feature propagation
C::print_k_from_A() : Void
<<VariableElement>>
{ id = unselectedFeature }
k != null <<...>> { ... }
i != null <<...>> { ... }
<<VariableElement>>
b : B
{ id = unselectedFeature }

Fig. 18 Application of the consistency rules for story diagrams

features from the already loaded feature model (Fig. 21). Thus, inconsistencies resulting from misspelled feature ids are avoided.

Feature not used Features that are not assigned to a domain model element indicate a potentially incomplete domain model. Thus, our tool checks if each feature contained in the feature model is assigned to at least one domain model element. If this is not the case, a warning is issued and the corresponding feature is marked in the feature model.

Feature annotation not satisfiable If mutually exclusive features are assigned to the same domain model element, it will not be contained in any configured domain model. To this end, the corresponding domain model element and the assigned feature annotations are highlighted to indicate the problem to the user. The user has to solve this conflict manually by removing one of the mutually exclusive feature annotations.

Dependency violation Certain model elements depend on each other. In the previous section we presented a set of constraints which have to hold for valid annotated domain models. Feature annotations performed by the user are automatically propagated to dependent model elements in order to avoid syntactical errors. These propagations are performed either on demand (through a command in the model editor) or automatically (during configuration of the model or generation of configured source code). The annotations which have been added automatically are maintained separately from annotations assigned manually by the user, i.e., the set of annotations is partitioned into two disjoint subsets. By default, automatically added annotations are hidden at the user interface. However, they may be displayed at any time upon the user's request. Additionally, automatically added annotations may also be removed in case the user annotations need to be changed. The feature annotations added by automatic feature propagation are displayed with a grey background, whereas manual annotations are presented with a white background color (Fig. 20).

4.1.3 Implementation of feature propagation

The tool *MODPLFeatureEditor* was developed in a model-driven way. The consistency rules, which are specified in Sect. 3 formally using OCL, were implemented using story diagrams. In our implementation, we chose to provide *repair actions*, i.e., propagation mechanisms that enforce the existence of feature annotations on depending model elements. These propagation mechanisms are implemented with Fujaba's story diagrams, which operate directly on Fujaba's abstract syntax graph to enforce a propagation of feature tags to dependent model elements. The result of the propagation mechanism is a model that conforms to the constraints of Sect. 3.

A separate *validation* of the model has not been implemented, because from the user's point of view a syntactically correct model is required. A validation points the user to errors in the model only, while the automatic repair actions propagate feature tags to the corresponding model element, which has to be done manually by the user otherwise.

Feature propagation is performed automatically when the domain model is configured or configured source code is

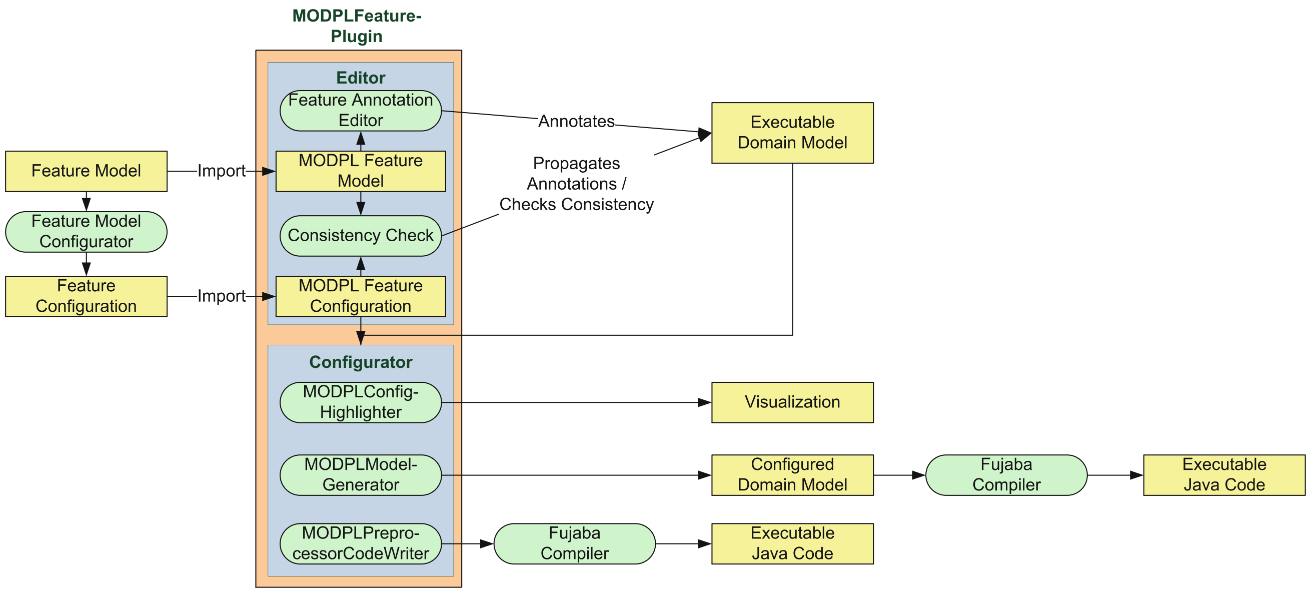


Fig. 19 Overview: tool functionality

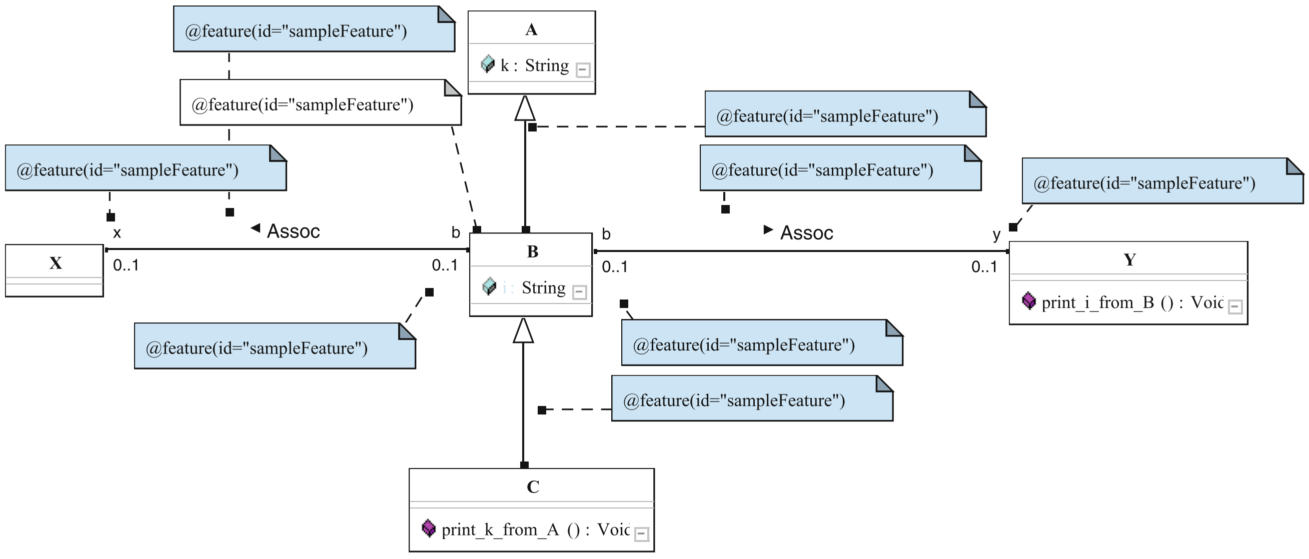


Fig. 20 Applying consistency rules

generated. Alternatively, the user may invoke feature propagation in the model editor.

Please notice that validation could have been performed directly on the basis of the OCL constraints of Sect. 3 (by integrating an OCL validation framework with Fujaba). However, OCL is a functional language, and the evaluation of OCL expressions does not have side effects. Repair actions do change the state of the mapping (between feature model and domain model) and thus cannot be implemented with a validation framework. Instead, we transformed the OCL constraints into repair actions in a systematic, yet manual way.

Below, we present two examples of story diagrams for implementing feature propagation. Please note that the story diagrams show the implementations of the repair actions which are used internally in the tool. Users working with our tool do not require any knowledge about repair actions or about the Fujaba meta-model. As the repair actions provided by our tool implementation refer to the Fujaba meta-model, they can be reused for any domain model created with Fujaba.

Example 4 (Propagation from a class) Figure 22 shows a story diagram defining a method that propagates feature tags assigned to a class to dependent elements.[6] All methods used

[6] Please note that this figure shows the actual implementation of the method based upon Fujaba's metamodel. This metamodel uses slightly different names for metaclasses.

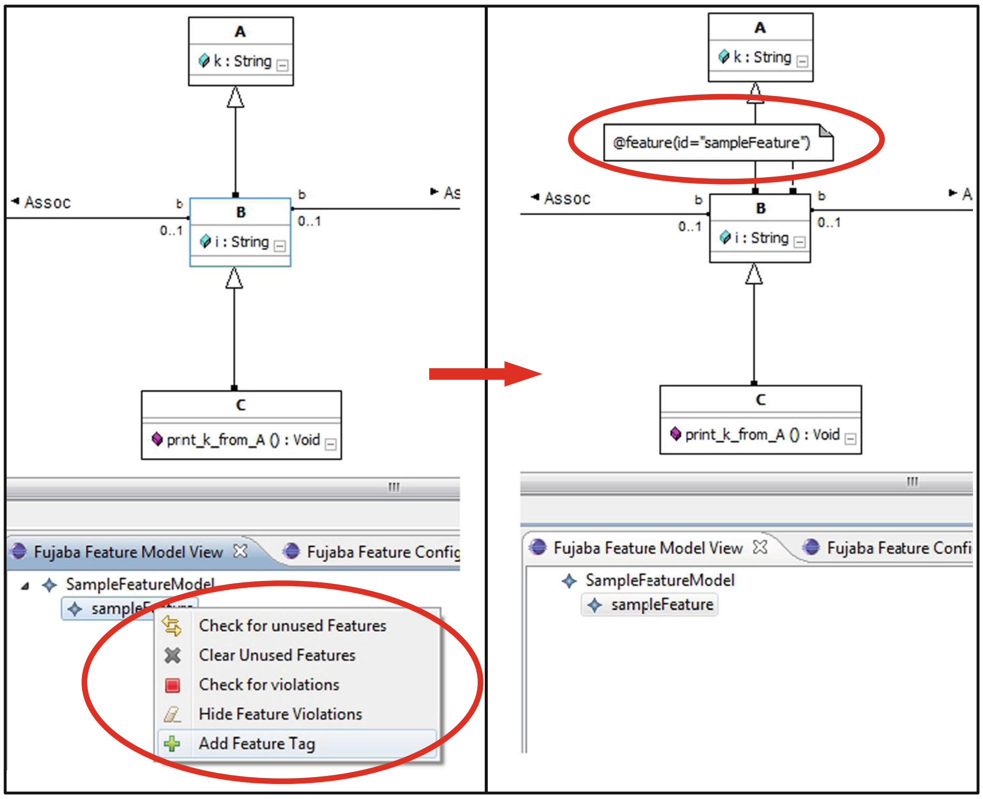


Fig. 21 Annotating model elements

to propagate feature tags to dependent model elements belong to the class TagPropagator which is part of our Fujaba plug-in named *MODPLFeaturePlugin*. An instance of this class is created for each project opened in the Fujaba IDE and holds references to all feature tags which have been added automatically by invoking the repair actions. These references are used for highlighting automatically added tags (see Fig. 20).

The story patterns depicted by rounded rectangles with a double frame represent *"for each"* activities, which are terminated when all instances of the patterns have been exhausted (outgoing end transition). For example, the first pattern iterates over all attributes that belong to the corresponding class and propagates the feature tags associated with the class to them. Propagation is performed by a method call, which is represented by an arrow labeled with a number (determining the order in the case of multiple calls) and the text of the call. In the pattern, the method propagate(Element source, Element target) is called which propagates feature tags from source to target in order to satisfy Constraint 1. Analogously, feature propagation is performed to other types of owned elements: methods (II), association ends (III), and outgoing generalizations (VI).

Pattern (IV) propagates features from a class to declarations in class diagrams in which it is used as a type (Constraint 2). (V) performs feature propagation to incoming generalizations (Constraint 3). (VII) handles instances of the class used in story diagrams (Constraint 5). (VIII) is concerned with the file which is used to store the generated source code of the class. Finally, in (IX) the method propagateAlongInheritanceHierarchy() is called in order to satisfy Constraint 8 (visibility of attributes; see next example).

Example 5 (Propagation to attributes) The method propagateAlongInheritanceHierarchy() ensures that attributes of superclasses being used in story patterns are visible (Constraint 8). To this end, features are propagated along the paths of classes and generalizations from the superclass owning the attribute to the subclass used in the declaration of the object owning the attribute expression in a story pattern. As a result of this propagation, the connecting paths are not filtered away in configurations of domain models in which the attribute expression is selected.

The corresponding story diagram is shown in Fig. 23. In some patterns, *path expressions* are used which define derived associations. Navigation is performed on association ends; navigations are composed with the dot notation. * denotes a transitive closure. In the case of a "for each" activity, an outgoing transition labeled with each time is used to enter the body of a loop. The body ends when the "for each" activity is reached again.

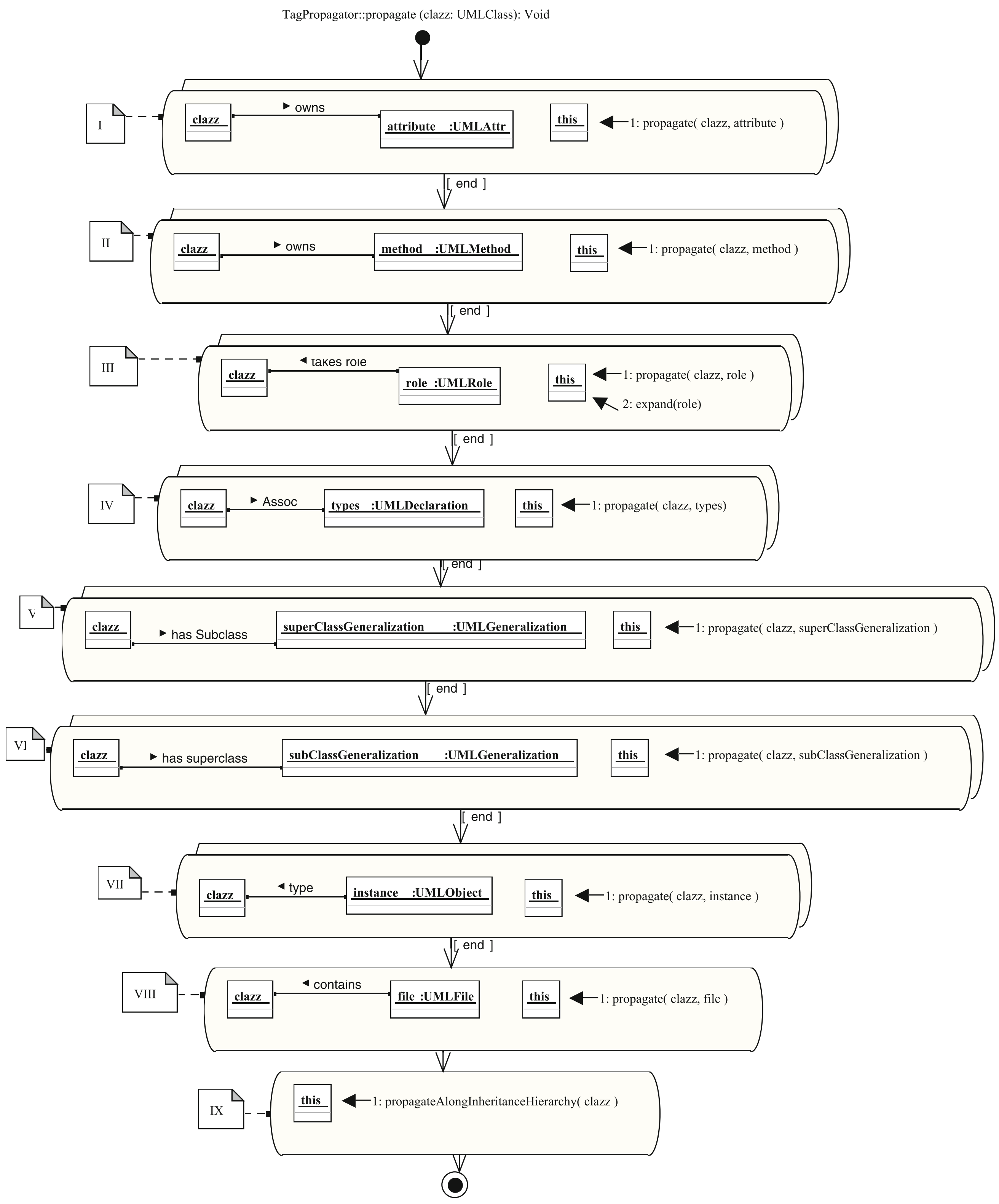

TagPropagator::propagate (clazz: UMLClass): Void
I
clazz
▶ owns
attribute :UMLAttr
this
1: propagate(clazz, attribute)
[end]
II
clazz
▶ owns
method :UMLMethod
this
1: propagate(clazz, method)
[end]
III
clazz
◀ takes role
role :UMLRole
this
1: propagate(clazz, role)
2: expand(role)
[end]
IV
clazz
▶ Assoc
types :UMLDeclaration
this
1: propagate(clazz, types)
[end]
V
clazz
▶ has Subclass
superClassGeneralization :UMLGeneralization
this
1: propagate(clazz, superClassGeneralization)
[end]
VI
clazz
▶ has superclass
subClassGeneralization :UMLGeneralization
this
1: propagate(clazz, subClassGeneralization)
[end]
VII
clazz
◀ type
instance :UMLObject
this
1: propagate(clazz, instance)
[end]
VIII
clazz
◀ contains
file :UMLFile
this
1: propagate(clazz, file)
IX
this
1: propagateAlongInheritanceHierarchy(clazz)

Fig. 22 Method implementation that propagates feature tags associated with classes to respective dependent elements

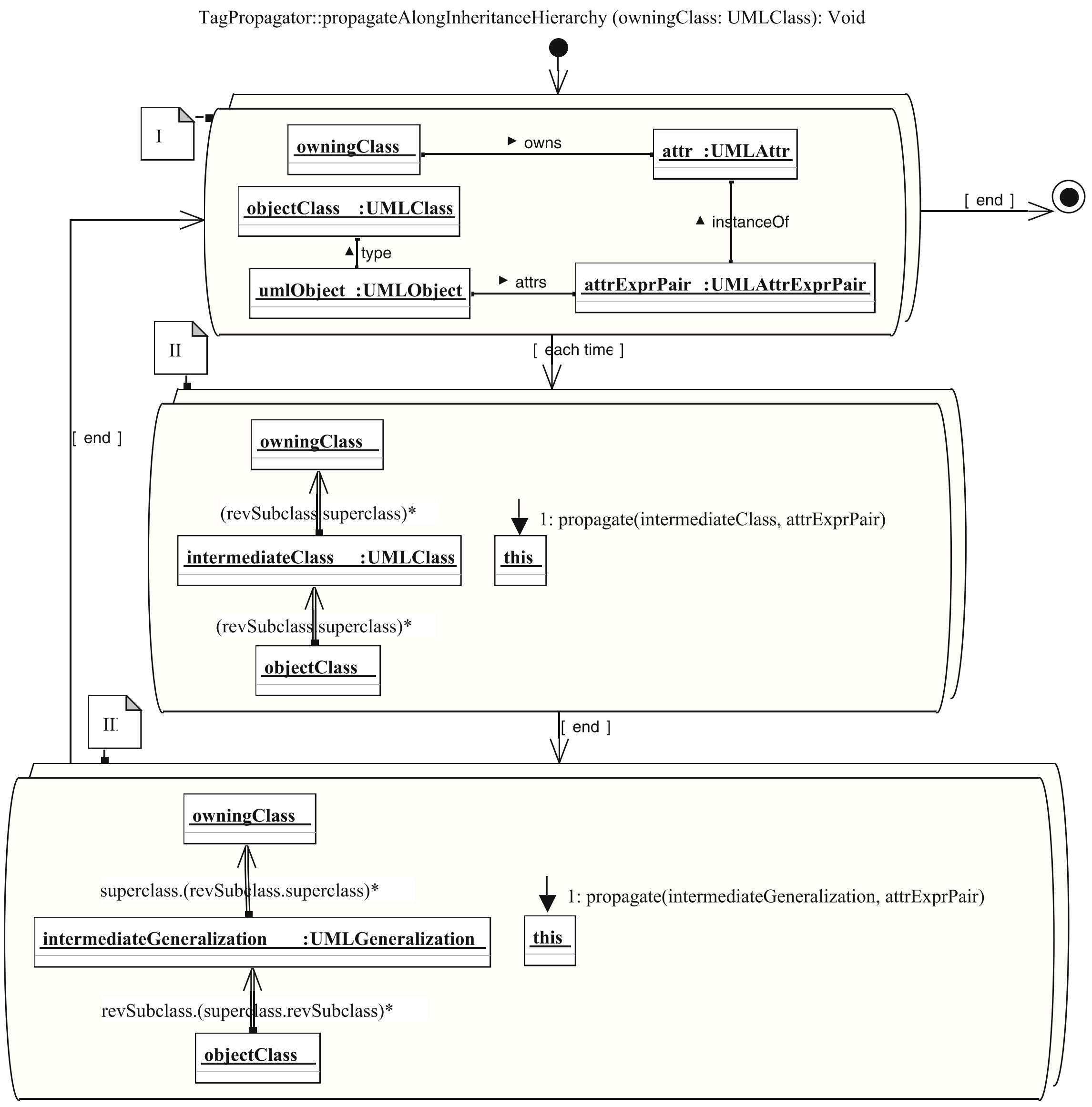

TagPropagator::propagateAlongInheritanceHierarchy (owningClass: UMLClass): Void
I
owningClass
owns
attr :UMLAttr
objectClass :UMLClass
instanceOf
type
umlObject :UMLObject
attrs
attrExprPair :UMLAttrExprPair
[end]
[each time]
II
[end]
owningClass
(revSubclass.superclass)*
1: propagate(intermediateClass, attrExprPair)
intermediateClass :UMLClass
this
(revSubclass.superclass)*
objectClass
[end]
III
owningClass
superclass.(revSubclass.superclass)*
1: propagate(intermediateGeneralization, attrExprPair)
intermediateGeneralization :UMLGeneralization
this
revSubclass.(superclass.revSubclass)*
objectClass

Fig. 23 Method implementation that propagates feature tags according to Constraint 8

Parameter owningClass determines the start of propagation. In Pattern (1), an attribute expression attrExprPair is searched which is an instance of some attribute attr of the owning class. Furthermore, the attribute expression must be part of an object umlObject whose declared class objectClass differs from the owning class.[7]

For each matched attribute expression, features are propagated from all classes and generalizations being located on a path from the object class to the owning class. Pattern (2) handles intermediate classes. For each intermediateClass which is a transitive parent of objectClass and a transitive child of owningClass, the features of intermediateClass are propagated to attrExprPair.[8] Pattern (3) works analogously for intermediate generalizations.

4.2 MODPL configurator

In order to start the configuration process, a corresponding feature configuration is required. FeaturePlugin allows to

[7] Fujaba's graph matching algorithm internally uses injective matches. As a consequence, the Fujaba runtime environment ensures that "owningClass" and "objectClass" are different instances of the metaclass UMLClass. Therefore, the UMLObject "umlObject" is not an instance of "owningClass". In this case, the object class must be a subclass of the owning class.

[8] The association ends revSubclass and superclass used in the implemented metamodel correspond to the association ends generalization and general from the UML2 standard, as depicted in Figs. 7 and 12.

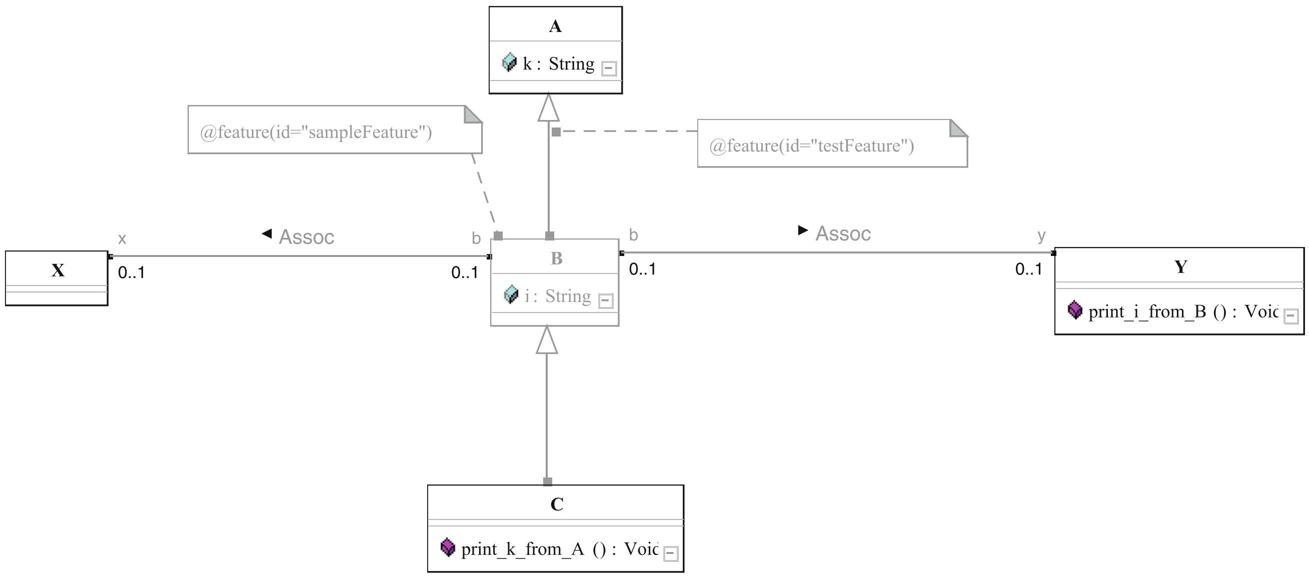


Fig. 24 Visualizing a configuration

easily create feature configurations by selecting and deselecting distinct features from the feature model.

4.2.1 Visualization

To provide an overview of the selected configuration, MODPLFeaturePlugin is able to *visualize* it directly in the Fujaba model editor. To this end, all model elements which are not contained in the current configuration are displayed in light grey color, whereas the contained model elements are displayed in their regular colors. Figure 24 shows a visualization of a configuration which does not comprise the feature sampleFeature.

4.2.2 Generating a configured model

Using model transformations, a tagged domain model can be transformed into a *configured model* based upon a specific feature configuration. To this end, a configuration created with FeaturePlugin is used to determine the model elements which are part of the target model. Afterwards the new model is built in memory and stored to a file for (possible) further editing. After editing is completed, the ordinary Fujaba compiler may be employed to generate Java code for the product instance to be built.

Changing the configured model is part of *application engineering*. Depending on the respective engineering process, the changes may or may not be propagated back to the feature model and domain model, which are developed in *domain engineering*. Permitting deviations from configured models provides for more flexibility. On the other hand, the domain model and the feature model may erode gradually if more and more development effort has to be invested in application engineering.

4.2.3 Generating executable code

Besides visualizing a configuration or generating a configured model, our tool also enables the user to generate configured and executable code directly. To this end, a *preprocessor* for the Fujaba model compiler was developed. Similar to a compiler preprocessor for textual languages, this preprocessor removes model elements which are not part of the chosen configuration. The Fujaba code generator is based upon the *chain-of-responsibility* pattern [24]. Thus, a chain of code writers exists and each token of the abstract syntax graph is passed to the first element of the chain during the code generation process. In case the current code writer is responsible for generating code for the token, the respective code fragment is generated, otherwise the token is passed to the next element of the chain. This design allowed an easy integration of our preprocessor: it is the first item in the chain. The preprocessor checks for each token whether it is visible in the current configuration. In this case, the token is passed to the next element in the chain in order to start the regular code generation process. Otherwise an empty string is returned and the chain ends.

Thus, *direct generation* of executable code again involves the use of the Fujaba compiler, which now is preceded by a preprocessor being part of our tool MODPLFeaturePlugin. In the case of direct generation, there is no need for the explicit creation of a configured domain model. Direct generation may be employed in a stringent top-down process, in which application engineering is reduced to a pure configuration process. Furthermore, direct generation may also be employed as a final step after a series of changes to configured domain models has been performed and these changes have been propagated back to the feature model and the domain model. In this way, it is guaranteed that the generated code conforms to the annotated domain model and the feature configuration of the respective product instance.

5 Case study

We applied the described approach successfully to a non-trivial project located in the domain of software configuration management. The project serves as a complex case study and was developed over several years. For a comprehensive description, please refer to the PhD thesis [20] or the corresponding journal article [11]. Further information is given in conference and workshop papers [7,9,10].

5.1 Domain

Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. A wide variety of SCM tools and systems has been implemented, ranging from small tools such as RCS [48] over medium-sized systems such as CVS [49] or Subversion [15] to large-scale industrial systems such as Adele [21] and ClearCase [54].

The current state of practice when developing SCM systems is characterized as follows:

1. SCM systems are *large*. For example, even the code base of the GNU CVS project comprises 300,000 lines of code. CVS is still a rather small tool compared to a commercial system for large enterprises such as ClearCase.
2. SCM systems are *similar*. For example, almost all commercial and open source systems are based on *version graphs*, which are used to manage the evolution of software objects.
3. The underlying *models* are defined only implicitly by the program code, i.e., the model is *hard-wired* into the respective system.
4. SCM systems are *hard to adapt* to modified requirements. For example, although Subversion provides similar functionality as CVS (at least from a bird's eye view), the developers of Subversion decided to start over from scratch.

Since all of the aforementioned systems share some common properties, common building blocks in a software product line could allow the configuration of these systems. The development of this product line is by far a non-trivial task. More information about the feature model and the domain model of our SCM product line can be found in [11].

5.2 Approach

These observations have motivated us to launch a project dedicated to the development of a *mod*ular and *mod*el-driven product line for *SCM* systems (*MOD2-SCM*) [9,20]. A product line reduces the effort of developing an SCM system by configuring an SCM system from a set of reusable components. Model-driven software engineering makes the underlying models explicit, which eases communication, reasoning, and change. Furthermore, the development effort is reduced by replacing the implementation of programs with the development of models at a higher level of abstraction.

To analyze the domain, we used the FORM method [37], which proposes to organize the feature model into a hierarchy of abstraction layers (capability layer, operating environment layer, domain technology layer, and implementation technique layer). The current feature model comprises 147 different features distributed over these layers.

Based upon the results of the domain analysis, an executable and configurable domain model was developed with the help of Fujaba. The domain model realizes a significant subset of the features specified in the feature model (102 features). Currently, the domain model comprises 136 classes distributed over 36 different packages. The product line contains in its current state different variants of product models (e.g., file system items, use case diagrams, EMF models), version models (e.g., set, sequence, tree), and delta storage mechanisms (e.g., forward deltas, backward deltas, mixed deltas). Furthermore, different persistence and locking mechanisms have been implemented.

Figure 25 illustrates the MOD2-SCM approach by presenting (simplified) cutouts of the feature model and the domain model. The left-hand side displays the feature model and a typical feature configuration for a CVS-like SCM system. For the history, the feature Branches has been selected, versions are stored with mixed (forward and backward) deltas, synchronization is optimistic, and version identifiers are maintained locally for each versioned item.

On the right-hand side, cutouts of the domain model are shown, which covers all features of the product line in a single model. On the top, the architecture of the domain model is represented by a package diagram. Product model, history, and storage are realized by respective packages, which are connected to the feature model by feature annotations. Below the package diagrams, a few class diagrams are displayed, focusing on the package for directed deltas and its imported interfaces. The abstract class DeltaStorage is refined into three subclasses, corresponding to the subfeatures of DirectedDeltas in the feature model. Methods of classes are realized by story diagrams, which, however, are not represented in the figure.

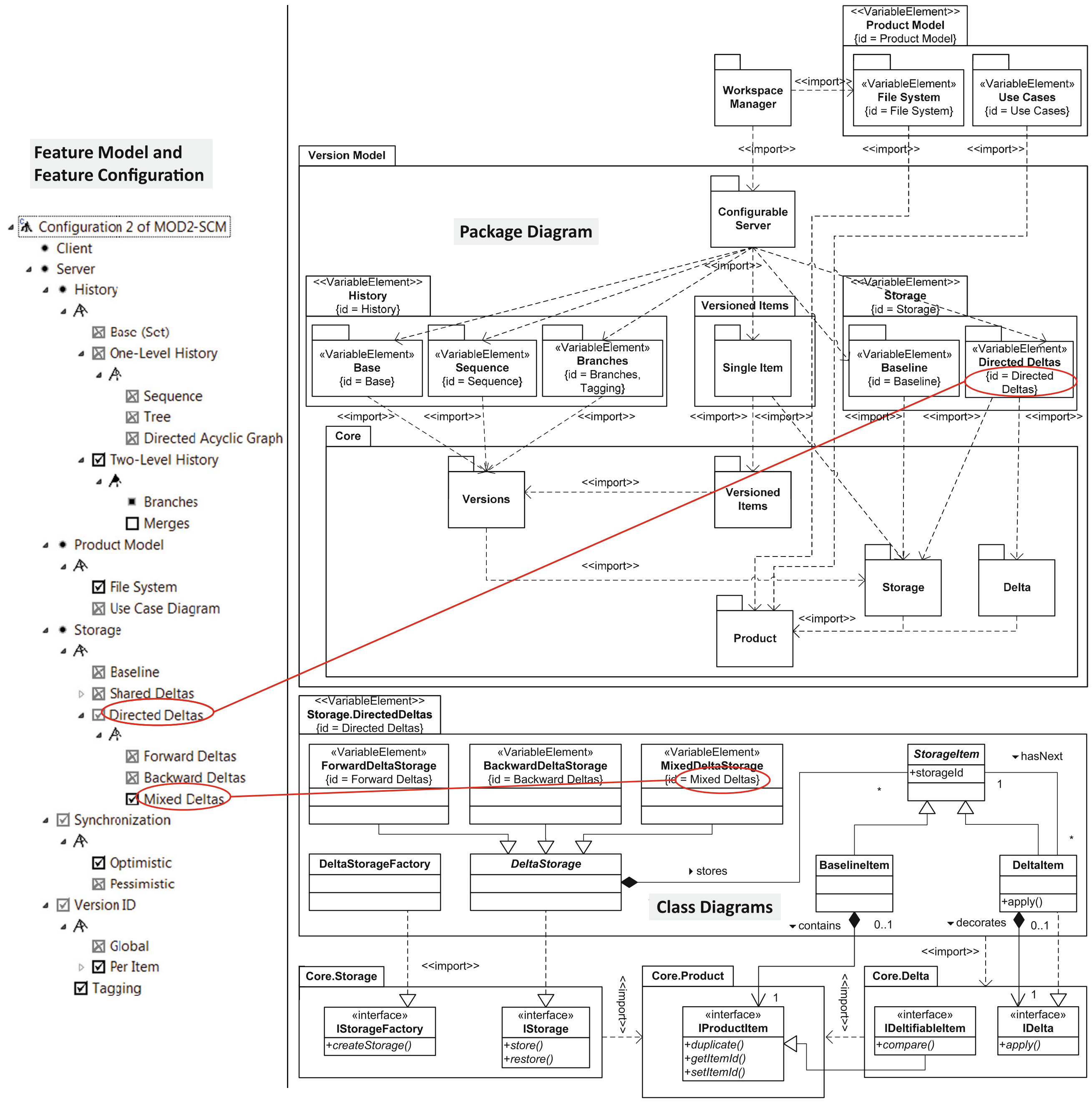


Fig. 25 Cutouts of feature model and domain model in MOD2-SCM

5.3 Experiences

The case study demonstrates that our approach to model-driven software product line engineering is feasible. From a single domain model, different variants of version control systems may be configured, supporting different product and version models as well as different mechanisms for storage and synchronization. The feature model is used for both planning and documentation of the product line. The mapping of features to elements of the domain model provides for traceability by recording which elements contribute to the realization of a certain feature. Furthermore, the mapping is used to configure the domain model based on a configuration of the feature model.

Figure 25 illustrates the application of our approach to the case study in the SCM domain. Please notice that the domain model is a thoroughly planned and designed artifact and not just an accidental superimposition of variants. By the mapping from the feature model to the domain model, it is documented clearly which elements realize which features.

For example, the package History realizes the corresponding feature in the feature model, and its subpackages realize respective subfeatures. Similarly, the package DirectedDeltas is used to realize the feature of the same name. Within the package, subclasses of the abstract class DeltaStorage realize different variants of directed deltas. Altogether, the domain model is based on a carefully designed *model architecture*.

The domain model for a product line has to cover all product variants in a single model. Thus, developing a domain model is an inherently complex task. In particular, the modeler needs to be assisted in establishing a consistent mapping of features to domain model elements. Our tool MODPLFeaturePlugin assists the user in various ways in defining the mapping (avoidance of errors, validations resulting in error messages and warnings, as well as repair actions). In particular, the repair actions may be used to propagate features automatically to dependent elements.

For example, in Fig. 25 the package DirectedDeltas has been decorated with the corresponding feature from the feature model, while most of its contained classes have not been decorated at all. Without feature propagation, these classes would be universally visible. Automatic feature propagation ensures that all owned elements of the package are also decorated with the feature DirectedDeltas. In this way, the effort to be invested by the modeler may be reduced significantly. Furthermore, accidental mistakes in the potentially error-prone process of annotating model elements with features may be detected and removed.

However, sophisticated tool support alone does not guarantee the success of model-driven product line engineering. In general, software product line engineering requires careful planning and design. In particular, potential feature interactions have to be considered thoroughly. This requires a model architecture which is composed of *loosely coupled components*. If the feature model declares features to be *orthogonal* (i.e., they may be selected independently), the model architecture has to ensure that the realizations of such features are not coupled inadvertently. The package diagram editor [12] which was developed by us and which is part of our tool chain supports the user during this tedious task.

For example, the MOD2-SCM architecture has been designed such that the product model, the version model, and the storage model may be combined in an orthogonal way. The model architecture in Fig. 25 is designed such that there are no mutual dependencies among the packages realizing these features. In particular, the version model does not depend on the storage model and vice versa. To make this work, the methods for adding a version to the version graph and for storing this version (potentially using deltas) have been decoupled.

By means of conscious design of the domain model and automatic feature propagation, the number of annotated elements has been kept very small. Altogether, only 54 domain model elements had to be annotated manually by the user (19 packages, 6 classes, 7 methods, and 22 elements of story diagrams).[9] 50 out of 54 domain elements carry only a single feature, 4 elements required the assignment of two features. Thus, the domain model does not suffer from the well-known "conditional compilation" syndrome which has often been observed on the source code level (the source code is cluttered with preprocessor options to such a degree that it is hardly readable any more).

It should be noted that considerable work had to be invested into the design of the feature model and the domain model to make the overall approach feasible. In particular, *feature interactions* had to be minimized. For example, in an early version of the domain model the delta storage package had a dependency on the history package: when storing a new version as a delta, the version graph was accessed to retrieve the predecessor version. Instead of adding a constraint to the feature model, the domain model was refactored in order to remove the unwanted feature interaction. Thus, the perceived simplicity of the solution results from a careful design rather than from the simplicity of the problem to be solved (which in fact is not simple at all).

Please notice that minimization of feature interactions does not mean that these interactions have been eliminated completely. In [20], feature interactions are analyzed in detail: 15 essential interactions are identified which have been written as implications ("requires" constraints).

Our *model-driven* approach to software product line engineering has proved successful inasmuch as all methods defined in class diagrams were realized as Fujaba story diagrams rather than hand-coded in Java. However, the story diagrams still contain statement activities, which are fragments of Java code. Thus, the behavioral model depends on the target language for code generation; a platform-independent behavioral modeling language would be greatly appreciated.

We analyzed the story diagrams both qualitatively and quantitatively; the results are presented in [13]. Essentially, this analysis showed that the expressive power of story patterns was not exploited to the degree we expected (i.e., story patterns were rather small on the average). This may be due to the modular development approach, in which concerns are separated clearly among different classes such that each class is responsible only for a small part of the overall domain model. Furthermore, our analysis revealed limitations concerning the expressiveness of control structures. On the positive side, story diagrams provide a graphical and executable description of the behavior of a method, which is probably

[9] In the actual implementation, mandatory features are not used for annotations. This reduces the number of annotations, but has a negative impact on traceability. Even with mandatory features, the number of annotations would be small compared to the size of the domain model.

easier to understand (for readers being fluent in Fujaba) than textual program code.

6 Discussion

In the following, we discuss achievements and limitations of the approach presented in the previous sections. In the course of our discussion, we identify several issues some of which are addressed by our current work on a next generation of tool support for model-driven software product line engineering.

6.1 Feature models

In our work, we decided to rely upon *cardinality-based feature modeling* [17], which we consider one of the most expressive approaches to feature modeling. In MODPL-FeaturePlugin, we use cardinality-based feature modeling in a slightly restricted way: cardinalities may be defined for inclusive-or groups, but not for individual features. Furthermore, feature attributes are not taken into account. Due to these restrictions, configuring a feature model constitutes a *selection process*: From all available features, a certain subset is selected which satisfies the constraints of the feature model. The domain model is configured by filtering all domain model elements which are annotated with features not being part of the feature configuration.

If the restrictions mentioned above are removed, configuring a feature model may no longer be formalized as a selection process. Rather, an *instantiation process* is required: a feature configuration is a tree of feature instances, where features with cardinalities may be instantiated as often as constrained by their cardinalities. Furthermore, the feature configuration contains attribute values. This means that information is added during the feature configuration step. In contrast, MODPLFeaturePlugin maps the feature model onto the domain model without being able to take such information into account, i.e., the mapping is defined before the instantiation of the feature model.

6.2 Domain models

Our approach to mapping feature models onto domain models is *general* inasmuch as it may be applied to *arbitrary domain metamodels*. In Table 1, several types of inconsistencies of the mapping were introduced which are handled by our tool MODPLFeaturePlugin in different ways. Most of these inconsistencies are generic, i.e., they are completely independent of the underlying domain metamodel (feature not declared, feature not used, feature annotation not satisfiable). Only dependency violations need metamodel-specific rules (Sect. 3.3). In this paper, we have defined dependency constraints for class and story diagrams. However, the concept of dependency constraints as such is general and not specific to the metamodels covered in our current tool support. For a new metamodel, the notion of dependency has to be defined with respect to this metamodel, resulting in a set of metamodel-specific constraints.

Currently, all dependency constraints have to be defined (and implemented) manually. Some dependency constraints could be derived automatically from the metamodel (e.g., a component depends on its container). However, other constraints are less evident and have to be defined manually (consider, e.g., Constraint 8, which deals with the visibility of attributes in story patterns).

6.3 Feature annotations

In MODPLFeaturePlugin, domain model elements may be annotated with *feature sets*. As explained in Sect. 4.1.1, a feature may be added to a feature set by selecting a feature from the feature model (Fig. 21). Thus, domain model elements may be annotated with features in a comfortable way with little effort.

Our tool distinguishes between *manual* and *automatic annotations*, which are managed separately (Fig. 20). In our case study, only a small fraction of domain model elements had to be annotated manually. In addition, almost all of these annotations consist of a single feature.

Altogether, feature annotations keep manageable as long as only a small fraction of domain model elements need to be annotated manually and these annotations are simple. The MODPL feature editor makes management of annotations easier by convenient commands for annotating model elements and automatic feature propagation. Our experiences from the MOD2-SCM project indicate that the overall approach is feasible if the domain model is designed carefully for variability.

6.4 Feature interactions

Ideally, features defined in the feature model are *orthogonal*, i.e., they may be selected independently. For example, in MOD2-SCM the history model is orthogonal to the storage model. However, in many applications of software product line engineering features interact. Thus, tools for software product line engineering need to support *feature interaction*.

In the feature model, feature interaction may be expressed by the feature tree as well as by context-sensitive constraints extending the feature tree. Here, MODPL "inherits" the functionality of FeaturePlugin, which implements cardinality-based feature modeling. In the domain model, feature interaction may be expressed by feature annotations. So far, MODPLFeaturePlugin supports feature sets, which means that a model element is visible only when all features from its feature set are selected.

Thus, with respect to feature annotations MODPLFeaturePlugin provides a simple approach which is easy to operate. By and large, this approach proved powerful enough to represent feature interactions in the MOD2-SCM project. In a few cases, however, it was necessary to express that a domain model element is included only when a certain feature is not selected. Negative annotations were simulated by extending the feature model with a negative feature. They could be handled easily without simulation by distinguishing between positive and negative feature sets and applying dependency constraints and propagation rules to both of them. This extension would retain the simplicity of feature annotations.

In a more general solution, the visibility of a domain model element would be determined via a boolean expression. In this way, more complex feature interactions may be represented in the annotations of domain model elements, supplementing global feature interactions defined in the feature model. While we are currently implementing this extension, we are also aware of the fact that complex feature interactions may render software product line engineering infeasible. In fact, minimizing feature interactions was one of the major drivers behind the MOD2-SCM project.

The approach presented in this paper can also be applied to problems requiring arbitrary complex feature interactions (by using constraints and annotations). It is obvious that a thorough analysis and the avoidance of unnecessary complexity yield a better result. Otherwise the user is faced with the “conditional compilation syndrome”.

6.5 Consistency control

6.5.1 Consistency

The main contribution of this paper consists in tool support for ensuring consistency of configured domain models. Here, our main focus lies on *dependency constraints*: when a domain model element is included in a configured domain model, all elements on which it depends are included, as well. Thus, each domain model element will have its required context in the configured domain model.

The dependency constraints specified in Sect. 3.2 contribute to the *syntactic consistency* of configured domain models. More precisely, we are referring to the *abstract syntax* of domain models (the dependency constraints are based on the underlying metamodel). Furthermore, we may distinguish between context-free and context-sensitive syntax.

Context-free syntax refers to the composition of the spanning containment tree. Context-free correctness is ensured partially by Constraint 1: if a component is visible, its container has to be visible, as well. In addition, dependency constraints in the opposite direction may be defined: if a container is included into a configured domain model, all mandatory components have to be included, too. For example, a dependency constraint of this type ensures that a story diagram has a (mandatory and unique) start node in each configuration of the domain model.

The notion of *context-sensitive syntax* subsumes all constraints which relate model elements at different locations of the containment tree. Among others, context-sensitive constraints deal with relationships between *declarations* and *applications*. For example, an association end (application) references a class (declaration). Constraints 2–8 all belong to this category.

The degree to which consistency of configured domain models is ensured depends on the dependency constraints which are defined on the underlying metamodel. If the overall domain model is consistent, all configured domain models are consistent with respect to those constraints which are addressed by feature propagation. Thus, a 100 % consistent domain model does not imply that all configured domain models are 100 % consistent, as well.

Not all constraints may be sensibly written as dependency constraints. For example, a story diagram must form a connected graph, where all activity and decision nodes are located on a directed path from the start node to an end node. In a configured domain model, this constraint may be violated if one or more control flows are not visible under the respective feature configuration. Connectivity in each configured domain model could be enforced by the following dependency constraint: if the story diagram is visible, each of its control flows has to be visible, as well. However, this is a sufficient condition which severely restricts variability.

6.5.2 Repair actions

Feature propagation enforces the satisfaction of dependency constraints. In MODPLFeaturePlugin, this is achieved by *adding features* to the dependent model elements. However, this type of repair action does not constitute the only way to satisfy dependency constraints. Instead of restricting the visibility of a dependent element, the visibility of a master element may be extended by *removing features* from the master element which are not present at the dependent element. In our current work, we are extending repair actions such that they may operate in both ways. Which repair actions are applied, may be configured by the user.

6.6 Variability

Two complementary approaches to supporting variability have been proposed in the literature:

- In the case of *filtering*, the domain model is a *union* of all product variants [16,29,31]. A configuration of the domain model is created by removing all elements which are not visible under the respective feature configuration.

- In the case of *composition* (model weaving), the domain model comprises the *intersection* of all product variants [3,55]. The domain model is configured by adding all model fragments which are selected in the respective feature configuration.

Composition is applied, e.g., in aspect-oriented approaches, which have become popular due to the separation of concerns. On the other hand, the domain model is just a collection of fragments, which are composed only at configuration time. Thus, problems concerning the interaction of model fragments may become apparent only late in development.

Our own approach is based on filtering. Thus, the modeler views and edits the complete model, which is stripped from invisible fragments at configuration time. If used in an undisciplined way, filtering may suffer from the *conditional compilation syndrome* observed in programming (the source code is cluttered with preprocessor directives to such an extent that it is neither readable nor maintainable). Thus, the modeler has to tame variability such that the overall domain model is kept manageable.

In the current paper, we have addressed *modeling-time configuration* of the domain model. In addition to modeling-time configuration, we also studied *run-time configuration* in the MOD2-SCM project. To this end, we built a tool which may configure an SCM server at run time. In this case, the domain model is not filtered at all; it is merely annotated to support traceability with respect to the feature model. Under these prerequisites, there is no way around designing a domain model which simultaneously covers all product variants.

Since we use only one domain model which comprises all variants of the software product line, a "natural" limitation of the variability is given by consistency constraints of the underlying UML metamodel. For example, a class needs to have exactly one name, an association end needs to be typed with exactly one target class, and it needs to have exactly one multiplicity. If more flexibility is required, the domain model is no longer consistent with the underlying metamodel. Editing inconsistent domain models would require radically different tool support [53].

6.7 Process

The MODPL environment illustrated in Fig. 1 provides a set of tools which may be orchestrated in different ways, resulting in different *processes*.

As commonplace in software product line engineering, we distinguish between the disciplines domain engineering and application engineering. In *domain engineering*, the feature model and the configurable domain model are developed. Ideally, *application engineering* is reduced to a configuration

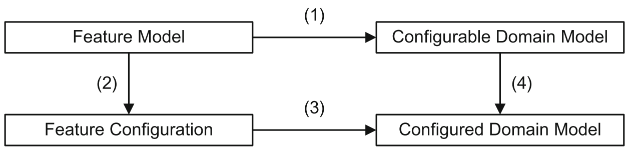

$$\begin{tikzcd}
\boxed{\text{Feature Model}} \arrow[r, "(1)"] \arrow[d, "(2)"'] & \boxed{\text{Configurable Domain Model}} \arrow[d, "(4)"] \\
\boxed{\text{Feature Configuration}} \arrow[r, "(3)"] & \boxed{\text{Configured Domain Model}}
\end{tikzcd}$$

Fig. 26 Model dependencies

process. In this case, source code may be generated directly from the feature configuration and the configurable domain model (shown at the bottom of Fig. 19). Otherwise, the configured domain model still needs to be edited (dashed line in Fig. 1), and code is generated from the configured domain model.

In a *phase-oriented process*, models are created in an order which is induced by their dependencies (Fig. 26). In an *incremental process*, models may be developed in an intertwined manner. MODPL allows for intertwined development of feature model and configurable domain model (1), but incremental change propagation has not been realized for the other cases (2–4).

Evolution of the *feature model* is handled as follows: adding a feature does not affect existing feature annotations in the domain model. Similarly, renaming a feature has no effect since immutable unique identifiers are used in feature annotations rather than mutable names. Finally, deleting a feature makes feature annotations including this feature invalid; these annotations may be repaired automatically by removing all nonexisting features.

7 Related work

Expressing variability in software product line engineering can be performed basically in two different ways: (1) negative variability and (2) positive variability. While approaches using negative variability are mainly based on filtering a set of all superimposed variants (e.g., similar to preprocessor directives in programming languages), the latter approach composes software artifacts based on selected features around a common kernel. In the following subsections we will discuss various approaches from both categories and compare them to our work presented in this article. In particular, we will provide a short overview about each approach and compare it the one presented in this paper in terms of methodology (positive / negative variability), domain model type, error detection and automatic error correction. Table 3 provides an overview about the different approaches. Each approach is discussed in detail in the following subsections.

7.1 MATA

MATA [55] describes a compositional approach to model-driven engineering of software product lines based on UML aspect models and graph transformations. In [55], the authors

Table 3 Comparison of the approaches discussed in this section

	PL approach	Domain model	Error detection	Error correction
MATA	Positive var.	UML	Yes	No
FeatureHouse	Positive var.	Arbitrary text-based	Yes	Yes[a]
fmp2rsm	Negative var.	UML	Yes	Yes
FeatureMapper	Negative var.	Ecore-based	Yes[b]	No
Pure::variants	Negative var.	Text files and EA models	Yes	No
PLiBS	Negative var.	UML	Yes	No
PLUS	Negative var.	UML	No	No
CIDE	Negative var.	Arbitrary languages[c]	Yes	Yes[d]
VML*	Neg. + pos. var.	Arbitrary languages[e]	yes	yes
DSL approaches	Positive var.	Arbitrary	Yes	No
MODPLFeaturePlugin	Negative var.	UML + Fujaba	Yes	Yes

[a] Only for context-free errors
[b] Only errors which do not address the well-formedness of the configured domain model
[c] As long as a grammar for it is provided
[d] Context-free error correction is provided
[e] As long as a VML-language for it is provided

focus on UML class diagrams, sequence diagrams and state diagrams. In contrast to other aspect oriented modeling approaches, MATA does not make use of explicit join points. Rather, any model element can be a join point and composition is treated as a special case of model transformation. While we use graph transformations both externally for behavioral modeling and internally for repair actions, MATA uses graph transformations only internally for composing the resulting model based upon a specific feature configuration.

The graph transformation rules are specified using AGG [46], which also serves as transformation engine to execute the model compositions defined with MATA. MATA employs critical pair analysis to automatically detect structural inconsistencies between different aspect models. Initially, critical pair analysis was invented for term rewriting systems but it has been adapted to graph rules [19]. MATA uses critical pair analysis to detect overlaps (interactions) between aspects. Interactions can be classified into conflict and dependency.

In contrast to our approach or to PLUS [25], MATA separates kernel and variant features into different diagrams. In particular, the kernel is represented by a UML model, whereas the variants are stored in separate MATA models [35]. Critical pair analysis for inconsistency detection is required since each variant feature is modeled independently from other variant features. In case conflicts are detected, they are fed back to the user. In contrast to our approach, error correction is the user's task and can be performed by either modifying the ordering of the different models to ensure a consistent composition or by updating the feature dependency diagram to resolve inconsistencies.

7.2 FeatureHouse

FeatureHouse is a framework and tool chain to support language-independent and automated software composition [3]. The composition technique used by FeatureHouse is based on feature structure trees (FSTs). The approach presented in [3] and [2] is a general approach which can be applied to compose software using superimposition as composition technique and does not only address source code artifacts (which can eventually be written in different programming languages), but also non-code artifacts like models, documentation or even build files. Or in other words, it can be applied to any textual representation of a software artifact, as long as a corresponding grammar for the specific artifact is provided. FeatureHouse offers a framework and tool chain which can be extended by attribute grammars to automate the integration of additional languages.

FSTs are a general model of the structure of software artifacts. Any kind of artifact can be represented with a hierarchical structure using FSTs. In fact, an FST is a stripped-down abstract syntax tree (AST) since only information specifying the modular structure of an artifact is contained [3].

Superimposition is based on the FSTs and is realized by merging the nodes of different FSTs. Nodes are identified by their names, types and relative positions, respectively. The merging process starts from the root node and descends recursively. Depending on the artifact language and node type, different rules for composition are used to reflect the different content of the nodes which is not represented as a subtree but as plain text.

In [2], the approach has been applied to UML models as well. The authors used an attribute grammar for UML models serialized in an XMI file. In contrast to our approach, the tool does not operate on the AST of UML, but on the AST of the XMI file instead which is an important difference because the tool can only detect context-free errors in the AST of an XMI file. Furthermore, our approach contains automatic repair actions operating on the abstract syntax tree of UML to ensure not only the correctness of the context-free but also the context-sensitive syntax of the resulting configured UML model. Like our tool, FeatureHouse assumes that a valid feature configuration is used when the final product is created.

7.3 fmp2rsm

fmp2rsm[10] integrates FeaturePlugin with the IBM Rational Software Modeler. The integration assumes that cardinalities and attributes are not used, reducing configuration of the feature model and the domain model to a selection process. The approach is based upon negative variability. Presence conditions determine the visibilities of model elements [16]. Explicit presence conditions are assigned by the user, while implicit presence conditions are maintained by the tool.

Implicit presence conditions supplement the explicitly assigned conditions. A model element is visible only when both its explicit and its implicit condition hold. Implicit presence conditions are pre-defined and depend on the underlying metamodel. In [16], implicit presence conditions are given for class diagrams and activity diagrams. These conditions are based on the generic rule that a model element is visible only when its required context elements are visible, as well. Thus, the dependencies between model elements are considered when configuring the domain model.

This approach differs from our work in various ways. First, fmp2srm and MODPLFeaturePlugin deal with different kinds of behavioral models (activity diagrams and story diagrams, respectively). Second, MODPLFeaturePlugin may generate fully executable code, which is not possible with fmp2rsm. Third, we use propagation rules rather than implicit presence conditions to ensure the consistency of the target model. While implicit presence conditions are wired into the tool and are hidden at the user interface, MODPLFeaturePlugin allows to expose automatically assigned features at the user interface, as shown in Fig. 20. In this way, the user may recognize erroneous annotations more easily than by inspecting the result of the configuration process (configured model or configured source code).

In [18], a general approach is described which allows to verify the correctness of configured domain models with respect to well-formedness constraints. This approach, which was implemented in fmp2rsm, as well, may be applied to any model which has a MOF based metamodel. Constraints regarding the well-formedness of models conforming to a MOF based metamodel are defined in OCL. For each constraint, it is checked whether the constraint is satisfied for each configuration of the domain model which may be constructed for some given feature model. This check is performed by abstract interpretation with the help of a SAT solver for boolean expressions. When a constraint is violated, the tool reports an object and a sample feature configuration in which the constraint does not hold.

The proposed approach is very general, and the constraints to be checked may be taken directly from the definition of the metamodel. In contrast, in our approach we had to derive the OCL constraints from the metamodel and convert them into repair actions. However, in contrast to our work, the approach presented in [18] supports only error detection and no error correction.

7.4 Feature Mapper

The tool *FeatureMapper*[11] [29,31,33] was developed to bridge the gap between feature models and Ecore-based domain models for model-driven product lines based on negative variability. These models comprise Eclipse UML2[12]-based models, domain-specific languages created with EMF [45] as well as textual languages, which have been described using EMFText [30].

The mapping of features to domain models is stored in an Ecore-based mapping model. Mappings may be created manually by selecting features from the feature model and elements from the domain model, respectively. Furthermore, the mapping is performed automatically in a recording mode, in which each created model element is decorated with a preselected feature expression.

FeatureMapper supports different kinds of visualization mechanisms to provide the user with analysis capabilities of the created mappings [29]. FeatureMapper allows to assign different colors for different features. Furthermore, different filtering views are provided which highlight elements being visible in a specific variant.

A feature configuration is defined by deleting unselected features from the underlying feature model. A configured model is derived from the domain model by deleting all model elements which are not visible in the respective feature configuration.

In [28], Heidenreich discusses different possibilities for checking well-formedness of SPLs. Most of the constraints listed in the paper are implemented in FeatureMapper. However, FeatureMapper does not provide a mechanism to ensure the well-formedness of configured domain models as a result of the generality of the approach. In [28], the author discusses this issue and states that the consistency of configured domain models can be checked if a complete set of constraints for the respective domain meta-model would be provided. In contrast to FeatureMapper, MODPLFeaturePlugin is restricted to UML and offers sophisticated mechanisms for ensuring the consistency of configurations of domain models, most notably feature propagation to dependent model elements. However, this requires the definition of metamodel-specific propagation rules. FeatureMapper does not provide a mechanism which provides user support for this purpose. Thus, it is easily possible to create syntactically inconsistent models, e.g., for the model presented in Example 1.

10 http://gsd.uwaterloo.ca/fmp2rsm.

11 http://www.featuremapper.org.

12 http://www.eclipse.org/modeling/mdt/?project=uml2.

7.5 pure::variants

pure::variants[13] is a tool which is commercially available in different versions and is heavily used in industrial projects. All versions share the same feature metamodel. For defining constraints for features or feature expressions, pure::variants offers a proprietary Prolog based rule language called *pvProlog*. Product derivation is performed based on negative variability.

The *Professional & Enterprise* version supports software product line engineering for file based systems, primarily focusing on source code. Additionally, requirement documents or user documentation can be managed. Multi-variant files and directories are defined by decorating the contents of files and directories with feature expressions.

The *Enterprise Architect* version offers a connection to Sparx Enterprise Architect[14], a tool supporting the creation of UML and SysML models. To connect elements of the feature model with domain model elements, a new constraint in Enterprise Architect must be created. Using this new constraint, simple mappings between single features and model elements can be established. A domain model may be configured by selecting the desired features using pure::variants and executing a model transformation.

Primarily, pure::variants addresses software product line engineering for file-based systems. Model-driven software product line engineering is supported by versions providing an integration with specific modeling tools. In these versions, the consistency of mappings from feature models to domain models is addressed only to a limited extent. In particular, dependencies among elements of the domain model are not considered, and repair actions (for automatic feature propagation) are not supported. Source code generation (from class models) is provided in the Enterprise Architect version. In contrast to MODPLFeaturePlugin, direct code generation from the multi-variant domain model is not supported.

7.6 PLiBS

PLiBS [59] is an Eclipse-based tool, which is based on the approach presented by Ziadi and Jezequel [58]. It allows for modeling and deriving behavior aspects using UML2 sequence diagrams extended by variability mechanisms in software product lines.

Product derivation is based upon negative variability and is realized by a two-step process. In the first step, all elements which are not part of the current configuration are filtered from the sequence diagrams. The second step uses UML state machine synthesis from sequence diagrams [56].

In contrast to our approach, the PLiBS process does not use a dedicated feature model to capture variability and from which valid configurations are derived. Instead, different stereotypes and tagged values are used to model variability directly within the UML model. For each feature a dedicated stereotype is created. The corresponding UML profile that is used is described in [57]. As a consequence, the constraints specified in [57] are used to express dependencies between features rather than defining constraints for ensuring syntactical correctness of the resulting configured model.

The approach presented in [58] and [59] does not provide mechanisms for repair actions which ensure context-free or context-sensitive syntactical correctness of the configured model. Furthermore, it does not support PL constraints checking [59].

7.7 PLUS

Gomaa presents in [25] an approach called *PLUS* to design software product lines with UML. Gomaa defines a profile containing various stereotypes to support expressing variability directly in UML. This approach is based upon negative variability and is similar to the one suggested by Ziadi and Jezequel [58] in terms of mixing up feature modeling and domain modeling.

This approach has several drawbacks, e.g., abstract classes are always treated as variation points. In case a hierarchical composition of variation points is required, this may result in a combinatorial explosion of subclasses. Since there is no dedicated feature model, feature interaction is addressed employing *feature impact analysis*. Dependencies among features can be detected using *object interaction modeling*. Another drawback of this approach is the huge amount of different stereotypes which have to be applied to corresponding domain model elements.

Gomaa does not address automatic repair actions to resolve violations of the abstract syntax of UML resulting from filtering elements based upon a current feature configuration. Tool support for this approach is provided by *PLUSEE*—the Product Line UML Based Software Engineering Environment [26].

7.8 CIDE

In [38,39], Kästner et al. present CIDE - a tool which allows to map feature model elements to source code fragments. Any language is supported, as long as a corresponding grammar is provided. They present a way to ensure the context-free correctness of the configured source code which is produced with the help of a preprocessor. This is quite similar to the generation of configured source code presented in this paper since we also implemented a preprocessor for the Fujaba compiler. The authors achieve context-free correctness of

[13] http://www.pure-systems.com.

[14] http://www.sparxsystems.eu/.

the configured source code through propagating the user set feature annotations to source code elements in the abstract syntax tree (AST). When a specific product configuration is derived, the AST is transformed into a configured one. All nodes that contain feature annotations with unselected features are removed [39].

A reference implementation for Java realizes two simple rules which are sufficient to guarantee the context-free correctness of the transformed AST. First, the user only may annotate nodes which are declared optional in the Java syntax specification [27]. For example, the name of a class cannot be removed in contrast to attributes or methods. Second, all children of an AST node have to be removed if the node is removed. For example, if an operation is removed, its parameters and its body are removed automatically. In contrast to our approach, only context-free correctness is considered. Context-sensitive rules such as specified in Sect. 3.2 are not taken into account.

7.9 VML*

In [60], Zschaler et al. present VML*. VML* is a family of languages for variability management in software product lines. It addresses the ability to express explicitly the relationship between feature models and other artifacts of the product line.

The VML* family of languages consists of: (1) a common metamodel for VML* languages which includes also variation points which can be customized for describing specific VML* languages; (2) a DSL allowing to specify the choices for each variation point made by a specific language and (3) an infrastructure which is based on generators.

The approach is very generic, since it supports any language, as long as a corresponding VML language exists for it. On the other hand, since the VML languages have to be defined first, they can take into account custom semantics of the target modeling language [32]. In [60], two VML languages for UML models are presented: VML4Arch and VML4RE. While the first is used to relate feature models and UML 2.0 architectural models, the latter is used for relating feature models and use case and activity models.

VML* supports positive and negative variability as well as any combination thereof, since every action is a small transformation of the core model. This has also several drawbacks, as mixing negative and positive variability mappings in the same specification may cause problems during product derivation as the order in which model transformations are executed is important. VML* allows the modeler to specify the order in which transformations are executed. Since the approach is very generic, providing automatic repair actions for arbitrary languages is a challenging task. So far VML* provides no support for it.

VML* is a very general and powerful approach which is on the other hand hard to use for SPL developers. First, a VML language for the desired modeling language has to be created. No support for diagrams in visual languages is provided. Furthermore, the mappings and the combination of the different possibilities to express variability result in very complex operators the SPL developer has to deal with.

In its current state, VML* does not produce diagram files for visual modeling languages. While our approach supports the detection of broken feature mappings, VML* provides no dedicated support for model evolution.

7.10 DSL-based approaches

In [50] Völter and Groher present an approach to support software product line engineering based on domain-specific languages and model-driven development. They propose the usage of *staged SPLs* or in other words meta-product lines.

The problem space (which is equivalent to our multi-variant domain model) is formally described by defining a meta model containing entities of the domain. Then, a DSL and corresponding editors based on this meta model are created. In the solution domain (which comprises a component-based architecture), a combination of manually written code and models are used to represent the components. Model-to-model transformations are used to instantiate, wire and deploy those library components based upon the problem domain model [50].

The authors present a meta product line for home automation systems. They address vendors developing systems for building architects. Using the meta product line, systems for the architects can be built enabling them to build smart homes for home owners. Weaving the domain meta model and adapting the DSL editors is required to support a configurable meta model and a DSL. A changing DSL also requires the adaption of the transformations from the problem domain model to the solution domain models. Configurable transformations are supported by using aspect weaving on the transformation level.

The approach presented in [50] uses lots of different techniques to realize this goal: both positive and negative variability is used, resulting in construction DSLs and configuration DSLs which are used on different stages of the development process. Furthermore, library components and target code variability are used. Runtime variability and aspect oriented programming on code level are also employed. As a consequence the variability has to be bound on different levels as well: during model-to-model transformations (e.g., when the problem domain model is mapped on the solution domain model), in model-to-text transformations (when the aspect oriented is generated) and during compile time, when the aspect oriented source code is compiled into the final application.

Arboleda et al. [4] present a similar approach, which is also based on domain-specific metamodels and variability models. Variability can also be bound at several stages of the configuration process, including model-to-text transformations based on openArchitectureWare (oAW). In contrast to Völter and Groher, Arboleda et al. use decision models to capture relationships between groups of variants and model transformations.

The approaches described above differ from our approach presented here in various ways. The most significant difference is the usage of staged SPLs in [50]. Furthermore, both positive and negative variability approaches are used. Different DSLs are needed on different stages of the development process, while our approach uses UML or Fujaba only. In contrast to our approach, automatic repair actions by feature propagation is not supported.

8 Conclusion

In this paper, we presented an approach that integrates model-driven software engineering and software product line engineering. Our approach is based on negative variability, i.e. a multi-variant domain model is annotated with feature sets. Products may be derived by configuring the feature model and applying the configuration to the annotated domain model. As a result, all elements which are annotated with unselected features are filtered from the resulting configured domain model. The filtering can easily result in syntactically wrong target models. To avoid this, we presented various constraints that help to ensure the consistency of the resulting configured model. We also developed a tool (MODPLFeaturePlugin) incorporating the consistency constraints and applying them to Fujaba's executable domain models. Furthermore, we successfully applied the tool to a product line for software configuration management.

Based on the concepts presented in this paper, we have recently launched the development of a next generation environment for model-driven software product line engineering. In particular, we are addressing the following issues:

Other domain metamodels Tool support for consistent mapping of feature models to domain models is generalized to work with instances of arbitrary Ecore models. In particular, we are addressing various kinds of UML2 diagrams, based on the Ecore model for UML2. With respect to behavioral modeling, story diagrams—which are specific to Fujaba—are replaced with state diagrams, activity diagrams, or text written in Alf (Action Language for Foundational UML [41]).

Rules for repair actions For each metamodel, a specific set of mapping constraints and repair actions needs to be defined. We are developing a rule-based language which allows to define custom constraints and repair actions in a declarative way.

Feature expressions In the approach described in this paper, domain model elements are annotated with feature sets. This approach is generalized by allowing the modeler to specify boolean feature expressions.

Propagation strategies In the approach described in this paper, feature annotations are always propagated to dependent model elements. In the new approach we are currently working on, propagation in the opposite direction is also supported.

Multiple stages The approach presented in this paper does not support multiple configuration stages (e.g., as required here [34] for example) explicitly (it can be simulated however). In our current approach we plan to add support for a multi-stage process.

Case studies Furthermore, we are looking for other complex case studies to further evaluate our approach.

Resources The update site for a Fujaba distribution including our plug-ins can be found on http://btn1x4.inf.uni-bayreuth.de/modpl/update. A screencast demonstrating the use is located here: http://btn1x4.inf.uni-bayreuth.de/modpl/screencast/MODPL-Screencast.htm.

Acknowledgments The valuable comments of the anonymous reviewers are gratefully acknowledged.

References

1. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange (eclipse'04), pp. 67–72. ACM Press, New York, NY (2004)
2. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In: Paige, R.F. (ed.) Proceedings of the International Conference on Model Transformation (ICMT), vol. 5563 Lecture Notes in Computer Science, pp. 4–19. Springer, Berlin, July (2009)
3. Apel, S., Kästner C., Lengauer, C.: FeatureHouse: Language-independent, automated software composition. In: Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE), pp. 221–231. IEEE, May 2009
4. Arboleda, H., Casallas, R., Royer, J.-C.: Dealing with fine-grained configurations in model-driven SPLs. In: Proceedings of the 13th International Software Product Line Conference (SPLC), Software Engineering Institute, pp. 1–10. Pittsburgh, PA, USA (2009)
5. Buchmann, T., Dotor, A.: Constraints for a fine-grained mapping of feature models and executable domain models. In: Mezini, M., Beuche, D., Moreira, A. (eds.) 1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09), pp. 9–17. CTIT Workshop Proceedings, CTIT, Twente, The Netherlands, June 2009
6. Buchmann, T., Dotor, A.: Mapping features to domain models in fujaba. In: van Gorp, P. (ed.) Proceedings of the 7th International Fujaba Days, pp. 20–24. Eindhoven, The Netherlands, November 2009
7. Buchmann, T., Dotor, A.: Towards a model-driven product line for SCM systems. In: Proceedings of the 13th International Software

Product Line Conference (SPLC 2009), vol. 2, pp. 174–181. Software Engineering Institute, San Francisco, CA, USA, August 2009
8. Buchmann, T., Dotor, A., Klinke, M.: Supporting modeling in the large in fujaba. In: van Gorp, P. (ed.) Proceedings of the 7th International Fujaba Days, pp. 59–63. Eindhoven, The Netherlands, November 2009
9. Buchmann, T., Dotor, A., Westfechtel, B.: MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In: Deridder, D., Gray, J., Pierantonio, A., Schobbens, P.-Y. (eds.) 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM08), pp. 50–65. Toulouse, France (2008)
10. Buchmann, T., Dotor, A., Westfechtel, B.: Model-driven development of software configuration management systems–a case study in model-driven engineering. In: Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT 2009), vol. 1, pp. 309–316. INSTICC Press, Sofia, Bulgaria, July 2009
11. Buchmann, T., Dotor, A., Westfechtel, B.: MOD2-SCM: A model-driven product line for software configuration management systems. Information and Software Technology (2012) (http://dx.doi.org/10.1016/j.infsof.2012.07.010)
12. Buchmann, T., Dotor, A., Westfechtel, B.: Model-driven software engineering: Concepts and tools for modeling-in-the-large with package diagrams. Comput. Sci. Res. Dev. 1–21 (2012) (online first)
13. Buchmann, T., Westfechtel, B., Winetzhammer, S.: The added value of programmed graph transformations–A case study from software configuration management. In: Schürr, A., Varro, D., Varro, G. (eds.) Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011), Budapest, Hungary, 2012. Presented at AGTIVE 2011, currently under review for publication in the post-proceedings
14. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA (2001)
15. Collins-Sussman, B., Fitzpatrick, B.W., Michael Pilato, C.: Version Control with Subversion. O'Reilly, Sebastopol, CA (2004)
16. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M.R. (eds.) 4th International Conference on Generative Programming and Component Engineering (GPCE 2005), vol. 3676 of Lecture Notes in Computer Science, pp. 422–437. Tallin, Estonia, September 2005, Springer, Berlin
17. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Softw. Process. Improv. Pract. **10**(1), 7–29 (2005)
18. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) Proceedings of ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'06), pp. 211–220. ACM Press, Portland, OR, October 2006
19. de Micheaux, N.L., Rambaud, C.: Confluence for graph transformations. Theor. Comput. Sci. **154**(2), 329–348 (1996)
20. Dotor, A.: Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen. PhD thesis, University of Bayreuth, Bayreuth, Germany, p. 459 (2011)
21. Estublier, J., Casallas, R.: The Adele configuration manager. In: Tichy, W.F. (eds.): Configuration Management, vol. 2, pp. 99–134. Trends in Software. Wiley and Sons, Chichester, UK (1994)
22. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In: Engels, G., Rozenberg, G. (eds.) TAGT '98–6th International Workshop on Theory and Application of Graph Transformation, vol. 1764. Lecture Notes in Computer Science, pp. 296–309. Paderborn, Germany, November 1998, Springer, Berlin
23. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, Indianapolis, IN (2003)
24. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns–Elements of Reusable Object-Oriented Software. Addison-Wesley, Upper Saddle River, NJ (1994)
25. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Boston, MA (2004)
26. Gomaa, H., Shin, M.E.: Tool Support for Software Variability Management and Product Derivation in Software Product Lines. In: Nord, R.L. (ed.) Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC), pp. 73–84. Boston, August 2004
27. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification., 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
28. Heidenreich, F.: Towards systematic ensuring well-formedness of software product lines. In: Proceedings of the 1st Workshop on Feature-Oriented Software Development, pp. 69–74. ACM, Denver, CO., USA, October 2009
29. Heidenreich, F., Şavga, I., Wende, C.: On controlled visualisations in software product line engineering. In: Thiel, S., Pohl, K. (eds.) Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), pp. 335–341. Limerick, Ireland (2008)
30. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) Proceedings of the 5th European Conference on Model Driven Architecture–Foundations and Applications (ECMDA-FA 2009), Lecture Notes in Computer Science, vol. 5562, pp. 114–129. Twente, The Netherlands, 2009, Springer, Berlin
31. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), pp. 943–944. ACM Press, Leipzig, Germany, May 2008
32. Heidenreich, F., Sánchez, P., Santos, J.P., Zschaler, S., Alférez, M., Araújo, J., Fuentes, L., Kulesza, U., Moreira, A., Rashid, A.: Relating feature models to other models of a software product line–A comparative study of FeatureMapper and VML*. Trans. Aspect-Oriented Softw. Dev. **7**, 69–114 (2010)
33. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07), pp. 38–42. Salzburg, Austria, October 2007
34. Jarke, M., Klamma, R., Pohl, K., Sikora, E.: Requirements engineering in complex domains. In: Engels, G., Lewerentz, C., Schäfer, A., Schürr, W., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering–Essays Dedicated to Manfred Nagl on the Occasion of His 65th Birthday, vol. 5765, pp. 602–620. Lecture Notes in Computer Science, Springer, Berlin (2010)
35. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), vol. 4735 of Lecture Notes in Computer Science, pp. 151–165, Nashville, USA, 2007, Springer, Berlin
36. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.A., Spencer Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute (1990)

37. Kang, K.C., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: A feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. **5**, 143–168 (1998)
38. Kästner, C., Apel, S., Saake, G.: Virtuelle Trennung von Belangen (Präprozessor 2.0). In: Software Engineering 2010—Fachtagung des GI-Fachbereichs Softwaretechnik, number P-159 in Lecture Notes in Informatics, pp. 165–176. Paderborn, Germany, February 2010, Gesellschaft für Informatik (GI)
39. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D.S.: Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In: Oriol, M., Meyer, B. (eds.) Proceedings of the 47th International Conference: Objects, Components, Models and Patterns (TOOLS EUROPE 2009), vol. 33, Lecture Notes in Business Information Processing, pp. 175–194. Springer, Zurich, Switzerland (2009)
40. Mezini, M., Beuche, D., Moreira, A.: (eds.) 1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09), CTIT Workshop Proceedings. CTIT, Twente, The Netherlands, June 2009
41. OMG: Action Language for Foundational UML (Alf), Beta 1 Specification. OMG, Needham, MA, ptc/10-10-05 edition, October 2010
42. OMG: Object Constraint Language, Version 2.2. OMG, Needham, MA, formal/2010-02-02 edition, February 2010
43. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3. OMG, Needham, MA, formal/2010-05-05 edition, May 2010
44. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations Principles and Techniques. Springer, Berlin (2005)
45. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Boston, MA (2009)
46. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance, vol. 3062, pp. 446–453. Springer, Berlin (2004)
47. The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. In: Giese, H., Zündorf, A. (eds.) Proceedings of the 3rd International Fujaba Days, pp. 1–13, September 2005
48. Tichy, W.F.: RCS–A system for version control. Softw. Pract. Exp. **15**(7), 637–654 (1985)
49. Vesperman, J.: Essential CVS. O'Reilly, Sebastopol, CA (2006)
50. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: Proceedings of the 11th International Conference on Software Product Lines (SPLC), pp. 233–242. IEEE Computer Society, Kyoto, Japan, September 2007
51. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology Engineering Management. Wiley and Sons, Chichester, UK (2006)
52. Weiss, D.M., Lai, C.T.R.: Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Boston, MA (1999)
53. Westfechtel, B., Conradi, R.: Multi-variant modeling–Concepts, issues and challenges. In: Mezini, M., Beuche, D., Moreira, A.: (eds.) 1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09), CTIT Workshop Proceedings, pp. 57–67. CTIT, Twente, The Netherlands, June 2009
54. White, B.A.: Software Configuration Management Strategies and Rational ClearCase. Object Technology Series. Addison-Wesley, Reading, MA (2003)
55. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Arajo, J.: MATA: A unified approach for composing UML aspect models based on graph transformation. In: Katz, S., Ossher, H., France, R., Jzquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI, vol. 5560, Lecture Notes in Computer Science, pp. 191–237. Springer, Berlin (2009)
56. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Revisiting statechart synthesis with an algebraic approach. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), pp. 242–251. IEEE Computer Society, Edinburgh, UK, May 2004
57. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Towards a UML profile for software product lines. In: van der Linden, F. (ed.) Software Product-Family Engineering, Lecture Notes in Computer Science, vol. 3014, pp. 129–139. Springer, Berlin (2004)
58. Ziadi, T., Jézéquel, J.-M.: Software product line engineering with the UML: Deriving products. In: Käköla, T., Duenas, C. (eds.) Software Product Lines, pp. 557–588. Springer, Berlin (2006)
59. Ziadi, T., Jézéquel, J.-M.: PLiBS: An Eclipse-based tool for software product line behavior engineering. In: Proceedings of the 3rd Workshop on Managing Variability for Software Product Lines, (SPLC, : Software Engineering Institute. Kyoto, Japan (2007)
60. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: VML*—A family of languages for variability management in software product lines. In: van den M., Gaevic, B.D., Gray, J. (eds.) Software Language Engineering, vol. 5969 of Lecture Notes in Computer Science, pp. 82–102. Springer, Berlin (2010)
61. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn (2001) (habilitation thesis)

Author Biographies

Thomas Buchmann received his diploma degree in mathematics in 2002 from the University of Bayreuth. For the following three years he was employed as manager of the software engineering department at a medium-sized company. Since 2005 he works as a research assistant at the software engineering chair. In 2010 he received his doctoral degree from the University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management and software architecture.

Bernhard Westfechtel received his diploma degree from University of Erlangen-Nuremberg in 1983 and his doctoral as well as his habilitation degree (all in computer science) from RWTH Aachen University in 1991 and 1999, respectively. Since 2004, he has been a full professor of computer science (in software engineering) at University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management, software process modeling, software architecture, and reengineering.