Constraints for a fine-grained mapping of feature models and executable domain models

Thomas Buchmann and Alexander Dotor

Angewandte Informatik 1, Universität Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

Abstract. In the past, several approaches have been made to combine feature models and domain models on the level of class diagrams. But the model-driven development approach also covers models that describe the behavior of a software system. In this paper we will describe a mapping of feature configurations to executable model elements which is one step towards an overall model driven process for product line engineering. We will specify consistency constraints that have to be met to ensure model correctness, and we will discuss the problems that arise during the final model-to-code transformation.

1 Introduction

The term model-driven development [1] of software systems describes the creation of systems by specifying models instead of writing code. Usually these models are created in CASE tools which provide class diagrams to model the static structure of a software system. These kind of diagrams lack the ability to model variability. In the context of software product lines [2], feature models are used to model variability in a family of software systems. Recently some approaches have been made to combine feature models and domain models created with CASE tools [3], [4], [5]. But model-driven development is more than just creating models that describe the static structure of a system - the behavior has to be described as well. In our current work, we develop a model-driven product line for *Software configuration management* (SCM) systems. The benefits of a model driven approach are (1) making the underlying models explicit, rather than having them implicitly defined in the program code, (2) providing reusable modules which can be combined in a flexible way through defining orthogonal components which are loosely coupled and (3) support rapid construction of new systems by providing a product line. The domain model of our product line consists of both class diagrams and behavioral diagrams. In the following we will discuss our approach to map features on elements of both structural and executable behavioral models which is one step towards a well-defined model-driven development process for software product lines.

2 Background

In our work we try to bridge the gap between features in a variability model and model elements of a system family. In a model-driven process a system configuration should

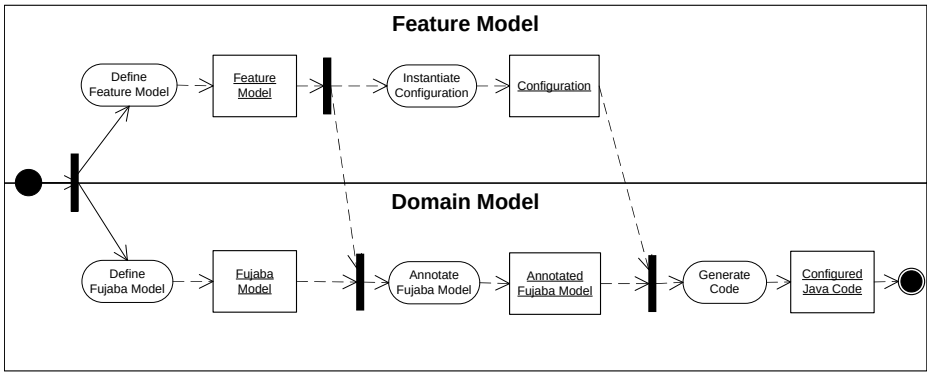

Fig. 1. The MDD process which combines Feature models and domain models using annotations

be mapped automatically to the domain model, and code for the specific feature selection should be generated. We use the CASE tool Fujaba [6] to create the executable domain model for the modular SCM system mainly because of its support for executable models. The added value compared to other CASE tools is provided by *story diagrams* (see p. 6) and their compilation into executable code. In contrast, code generation from class diagrams is supported by a large number of other CASE tools as well, like e.g., in the Eclipse Modeling Framework [7]. Due to space restrictions, we can not provide detailed information about the domain model itself. Information how the domain model has been designed to support orthogonal combination of features and how the behavioral model may be defined in a graphical notation at a level of abstraction above the (generated) program code, can be found in [8], [9] and [10]. Feature modeling was applied to define common and discriminating features of SCM systems. In the context of product line engineering, feature models are widely used. So far, we have defined no constraints on the combination of features (apart from those constraints which are expressed directly in the feature diagram itself). An essential goal of our project is building a product line for SCM systems where features may be combined as freely as possible. However, the major effort has to be invested into the design of a system which actually supports the features defined in the feature model. Defining the feature model itself is fairly easy. In our approach the domain model was annotated manually with features, to establish a mapping between elements of the domain model and the feature model respectively. These annotations can be performed on any level of granularity. On a coarse-grained level, units such as packages, classes and associations are decorated with features, whereas on a more fine-grained level, attributes, methods and even story patterns etc. can be decorated. The coarse-grained approach keeps the multi-variant architecture manageable. But it is up to the modeler's discipline to use the feature annotations carefully. In an extensive way of using feature annotations, the modeler may easily lose track and may face a degree of complexity which cannot be managed anymore. During the code generation process, these feature annotations are evaluated against a given system configuration which is specified in FeaturePlugin [3] and only code for selected elements is generated (see Fig. 1).

3 Technical Problem

To generate code for a system configuration, a mapping has to be established between features of the feature model and the model elements of the domain model. We chose **tagged values containing the name of the features** to realize this mapping. If a feature is selected in a configuration the model elements with its name have to be part of the configured domain model. Each model element can be tagged by multiple features, in which case they are evaluated analogous to a logical *and* (i.e., all features have to be selected). This means also, that untagged model elements are always part of the configured model. The mechanism to tag the model elements varies slightly between different metamodels. In UML, stereotypes with feature names can be used ([11], pp. 651), while e.g., Ecore provides annotations ([7], pp. 119).

Our ultimate goal is to generate executable code from the configured model. Therefore, the transformation of the domain model into a configured domain model can be viewed as analogy to a preprocessing step of a compiler – a **code generation preprocessor**. As a consequence we have to deal with the same problems as compiler preprocessors: it is easy to produce syntactically and semantically wrong code.

Take the following example: Figure 2 shows a class diagram whose class `B` has been tagged by a feature named *sampleFeature*. As long as the sampleFeature is part of the configurations everything runs well, but as soon as it is omitted several problems occur (assuming only class B is omitted from the configured model):

1. Both generalizations have either no target or no source.
2. Both associations have only one member end.
3. Both class X and class Y have a Property of a non-existing type.
4. print_J_from_A in C fails during opaque expression analysis step (e.g., during compile time, see [11], pp. 101), as C is no A anymore, so j cannot be accessed.
5. print_I_from_B in Y fails also, as B is not associated with X anymore, so i cannot be accessed

This example shows that several syntactical constraints have been violated in the configured domain model, e.g., associations and generalization with only one end node. These problems are **violating UML metamodel constraints**. But another kind of problem can only be detected **during compile time**, e.g., the broken inheritance hierarchy.

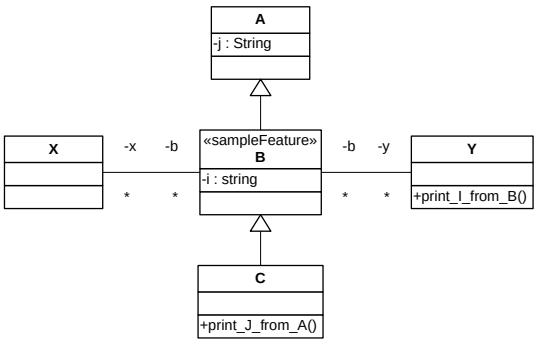

Fig. 2. Example of a tagged class diagram

This leads to the question, if there is a set of consistency rules to maintain syntactical correctness both when configuring a tagged domain model and when generating code for the configured domain model.

4 Solution

If the tagged domain model is syntactically correct all errors in a configured domain model result from the removal of tagged elements (this is quite similar to the deletion of model elements). An untagged element is present in every configured domain model. A tagged element is only present in the subset of configured domain models associated with the conjunction of the features it has been tagged with. So, if another element depends on a tagged element the following rule holds:

> **Rule 1:** If model element A depends on model element B, the set of configured domain models containing A must be a subset of the set of configured domain models containing B.

Or, thinking in terms of tags, the constraints must ensure that every tag on an element is present on its dependent elements, i.e., the tag set on element B must be a subset of element A[1].

4.1 Constraints for UML metamodel violations

This set of constraints is obtained by analyzing the UML Superstructure Specification [11]. All these rules works transitively, i.e., they have to be applied until no further tags are added.

> **Rule 1.1:** If an element is tagged all owned elements are tagged, too.

The basic `Element` of the UML Superstructure Specification introduces an aggregation between an `owner`-Element and its `owned elements` ([11], p. 25). By analyzing the inheritance hierarchy of the UML Superstructure we can define following dependencies (see Table 1).

> **Rule 1.2:** If the target of a directed relationship is tagged the relationship is tagged, too.

Each `DirectedRelationship` must have a source and a target ([11], p. 25). The source of a `DirectedRelationship` is always the owner, so rule 1 insists that a tagged source implies a tagged `DirectedRelationship`. This is also the case if the target is tagged which is demanded by this rule. The dependencies for the concrete elements are shown in Table 2.

> **Rule 1.3:** If the member end of an association is tagged the association is tagged, too.

[1] Please note that the direction of the subset relation has changed, because more tags mean less configured domain models.

Tagged type	**Elements to be tagged (type)**
`Association`	owned rules (`Constraint`), outgoing imports (`ElementImport` or `PackageImport`), generalizations to super-associations (`Generalization`), non-navigable or n-ary roles (`Property`)
`Class`	nested classes (`Class`), owned rules (`Constraint`), outgoing imports (`ElementImport` or `PackageImport`), generalizations to superclasses (`Generalization`), defined operations (`Operation`), defined attributes (`Property`)
`Constraint`	constraint definition (`ValueSpecification`)
`ElementImport`	*none*
`Generalization`	*none*
`Operation`	pre-/post and body conditions (`Constraint`), parameters (`Parameter`)
`Package`	contained associations (`Association`), contained classes (`Class`), owned rules (`Constraint`), outgoing imports (`ElementImport` or `PackageImport`), sub-packages (`Package`), outgoing package merges (`PackageMerge`)
`PackageMerge`	*none*
`PackageImport`	*none*
`Parameter`	default value (`ValueSpecification`)
`Property`	default value (`ValueSpecification`)
`ValueSpecifications`[2]	*none*

Table 1. Tag propagation Table for concrete UML class diagram elements (Rule 1.1)

Each `Association` requires at least two member ends. So, in case of tagged properties, the tags have to be propagated to the association that ends at this property.

Rule 1.4: If an association is tagged the member ends are tagged, too.

If an end of an `Association` is navigable it is owned by the appropriate `Class`. In this case a tagged association requires its member ends to be tagged.

Rule 1.5: If a type is tagged all typed elements of this type are tagged, too.

If a class is tagged which is a target of an uni-directional `Association` there is no link to the `Property` that holds a reference to the source class, because this direction is non-navigable (see [11], pp. 123). So, the `Property` of the source class that references the tagged class is only linked to the tagged class via the `type`-Association which is used in this rule.

4.2 Constraints for story diagram metamodel violations

UML provides several behavioral models but – except for state charts – there are no means to generate executable code. Instead, each Operation has a methodbody-string

[2] `ValueSpecification` is actually an abstract class that represents expressions consisting of various literals and so called *opaque expressions*. These expressions are strings associated with a language, e.g., java source code or OCL expressions that come with their own validator (i.e., java compiler or OCL checker). See [11], pp. 28 and pp. 101 for a complete definition.

Tagged type	Elements to be tagged (type)
`Association`	incoming element import (`ElementImport`), generalization from sub-association (`Generalization`)
`Class`	incoming element import (`ElementImport`), generalization from sub-class (`Generalization`)
`Package`	incoming imports (`ElementImport` or `PackageImport`), incoming package merges (`PackageMerge`)

Table 2. Tag propagation Table for concrete UML class diagram elements (Rule 1.2)

that is validated by a languages specific tool (e.g., a java compiler) [11] (pp. 101). The CASE tool *Fujaba* provides behavioral modeling through **story diagrams**, to specify the body of a method and generate executable code. Story diagrams are activity diagram with two kinds of nodes: Statement activities and story patterns. The first consists of a fragment of Java code, allowing for seamless integration of textual and graphical programming. The latter is a communication diagram composed of objects and links. Furthermore, objects may be decorated with method calls. Elements with dashed lines represent optional parts of story patterns. A crossed element means that the story pattern may be applied only when the respective element does not exist. In addition to method calls, a story pattern may describe structural changes: Objects and links to be created or deleted are decorated with the stereotype `<<create>>` (green color) or `<<destroy>>` (red color), respectively. Furthermore, `:=` and `==` denote attribute assignments and equality conditions, respectively.

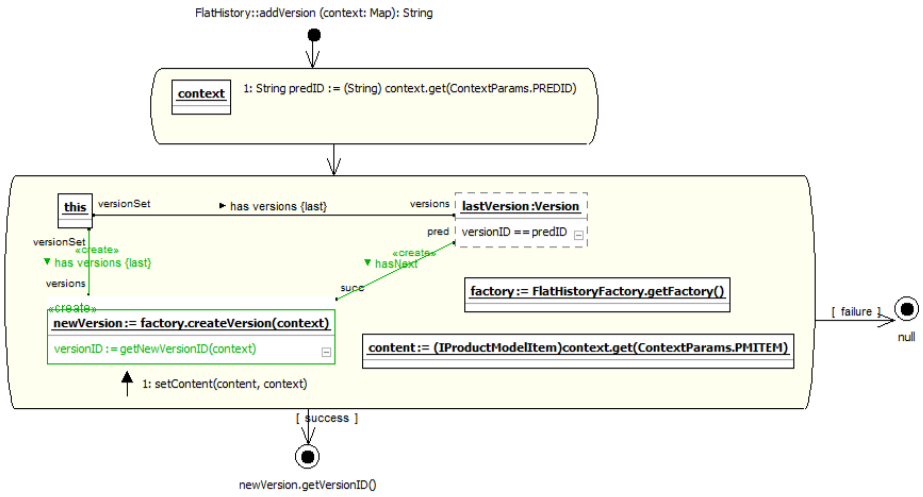

Fig. 3. Example of a Fujaba Story diagram

Figure 3 shows a story diagram that adds a new version to a single-dimensioned version history (i.e., a kind of linked list). The first activity is a collaboration call that retrieves the ID of the previous version (`predID`) out of the `context`-Parameter. The second is a story pattern. It creats a `newVersion` and adds it as last element to the `versionSet`. If a `lastVersion` exists already (note this object is optional) whose

`versionID` equals the `predID` it becomes the predecessor of the `newVersion`. Ultimately the `content` is added via method call to the `newVersion`. Depending on the outcome of this pattern (successful match or failure) the control flow continues to a stop node that either returns the new `versionID` or `null`. In Fujaba each metamodel element has a reference to its instances, e.g., a class to its objects, etc. The class *Version* has two instances in our example (Fig. 3): *lastVersion* and *newVersion*. Instead of passing the validation to a compiler it is now possible to think in terms of behavioral model elements and to define further rules to ensure their correctness.

Rule 1.6: If an UML element is tagged all its instances are tagged, too.

When an element of the class diagram is tagged all its instances in all story diagrams have to be tagged.

Rule 1.7: If a UML Operation is tagged its story diagram is tagged, too.

Each story diagram defines the behavior of a single operation. By tagging an operation its behavioral specification has to be tagged, too.

Rule 1.8: If an UML element is tagged all instances of owned elements of its superclasses are tagged in story patterns of its subclasses, too.

This solves the interrupted inheritance tree problem, by tagging all instances of elements that are not inherited anymore.

If we apply the above rules on the example given in Figure 2, we get the following result: Rule 1.1 implies that the properties *x* and *y* are tagged, as well as the generalization to class A. The generalization from class C to class B is tagged according to rule 1.2. Applying rule 1.3 results in tags on both associations. The opposite navigation ends (*b*) are tagged because of rule 1.4.

4.3 Open Problems and Limitations

The current version of the Fujaba story diagram metamodel does not link every story diagram element to its class diagram element, e.g., constraints, transition guards and collaboration statements. And even after a metamodel expansion, Fujaba still allows statement activities which contain arbitrary strings. A way to ensure syntactical correctness of the generated code is to avoid these elements altogether, or to generate and compile the configured model in the background. Another limitation is the inability to specify variants of model elements (both in UML and story diagrams) as both metamodels are not designed for this purpose, e.g., it is not possible to define cardinality variants for association ends or attribute assignment variants for objects.

5 Related Work

Czarnecki et al. describe in their work about Mapping Features to Models [12] a way to establish a bidirectional mapping between feature models and Ecore elements, based on

Ecore class diagrams (see [4]). It uses many of the same notations and display elements as a previous version named *FeaturePlugin* [3]. Unlike FeaturePlugin, which focuses strictly on feature modelling in an isolated context, Ecore.fmp aims to create Ecore compliant class diagrams out of existing feature models and vice-versa. In the current version of Ecore.fmp, the creation of a feature model from an existing Ecore model is not yet supported properly. The creation of Ecore model files out of feature models also still needs to be implemented. Since it is tightly coupled with Ecore, it does not support arbitrary EMF-models or even executable models.

In his work, Florian Heidenreich developed a set of Eclipse plugins that also allows the user to establish a mapping between features and feature realisations (i.e., model elements) [13]. The underlying model (feature realisation) can be defined in arbitrary Ecore-based languages. It provides four different kinds of views, that visualize the current feature selection in different ways [14]. The plugin aims at supporting the developer in the complex task of defining mappings between features / configurations and their realizations. However, support for executable models is very limited due to Ecore. E.g., statecharts, which do not have the same expressive power as Fujaba's story-diagrams, can be used to model the behaviour. Furthermore, both Ecore.fmp and Featuremapper model requirements and components on the same level of abstraction. Ecore.fmp also has no direct support for modeling in the large [15].

There are also some commercial tools, that support modeling a product line by specifying feature models, like pure system's *pure::variants*. These tools do not provide a model-driven process to develop a product line in a model-driven way. They only cover a small part of the product line process - variant management. The configuration of the final product is done during runtime, for example by specifying `defines` which are passed to a C/C++ preprocessor, or by selecting certain sourcecode files. *pure::variants* does not support the mapping of features to model elements.

6 Conclusion

In this paper we presented a novel way to combine feature modeling and model-driven software development, especially when creating executable models. The mapping between features and model elements was done using model annotations. A configuration of selected features is used as an input to a special preprocessor which is started before the actual code generation process. This preprocessor passes model elements with matching feature annotations to the code generator, and drops model elements that do not match. In that way it is possible to establish a mapping between feature model elements to elements in executable models on any level of granularity. The code that is delivered to the customer only contains the fragments which are neccessary for the desired configuration, in contrast to the process of runtime configuration and deploying the complete codebase [10]. We deduced several rules to ensure syntactical correctness of the annotated domain model. These rules have been implemented in a plugin which ensures consistency of the model when generating code. Current work is addressed to integrate the plugin into the Fujaba editor, to allow using our consistency checks during the edit process.

At the moment, our preprocessor only accepts configurations that have been created using the FeaturePlugin [3] plugin. We need to build different import modules to read configurations created with other feature modeling tools like FeatureMapper [5], Ecore.fmp [4] or commercial tools like pure::variants etc. We will also investigate to which level of granularity it is possible to annotate the executable model. It is possible to annotate the static structure at any level of granularity (Classes, methods or even attributes), but does the same also hold for story patterns? E.g., we will try to find out if it is possible to exclude single story patterns of a story diagram by annotations.

References

1. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
2. Clements, P., Northrop, L.: Software product lines: practices and patterns. Volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
3. Antkiewicz, M., Czarnecki, K.: Featureplugin: Feature modeling plug-in for eclipse. In: OOPSLA '04 Eclipse Technology eXchange (ETX) Workshop, Vancouver, British Columbia, Canada, ACM (Oct. 24-28 2004)
4. Stephan, M., Antkiewicz, M.: Ecore.fmp a tool for editing and instantiating class models as feature models. Technical report, University of Waterloo (2008)
5. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM (May 2008) 943–944
6. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn, Germany (2001)
7. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. 2 edn. The Eclipse Series. Addison-Wesley, Boston (2009)
8. Buchmann, T., Dotor, A., Westfechtel, B.: Triple graph grammars or triple graph transformation systems? a case study from software configuration management,1st international workshop on model co-evolution and consistency management, mccm 08, toulouse, france, september 30th, 2008. (2008)
9. Buchmann, T., Dotor, A., Westfechtel, B.: Mod2-scm: Experiences with co-evolving models when designing a modular scm system. In: 1st International Workshop Co-Evolution and Consistency Management (MCCM '08). (2008)
10. Buchmann, T., Dotor, A., Westfechtel, B.: Model-driven development of software configuration management systems - a case study in model-driven engineering. submitted for publication
11. OMG: OMG Unified Modeling Language (OMG UML), Superstructure. OMG. (November 2007) Version 2.1.2.
12. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE 05. (2005)
13. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07). (2007)
14. Heidenreich, F., Şavga, I., Wende, C.: On controlled visualisations in software product line engineering. In: Proceedings of the 2nd Int. Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008). (September 2008) To appear.
15. Buchmann, T., Dotor, A., Westfechtel, B.: Experiences with modeling in the large with fujaba. In Assmann, U., Johannes, J., Zündorf, A., eds.: Proceedings of the 6th International Fujaba Days, University of Dresden, University of Dresden (2008)