Mapping Features to Domain Models in Fujaba

Thomas Buchmann  
Angewandte Informatik 1  
Universität Bayreuth  
D-95440 Bayreuth  
thomas.buchmann@uni-bayreuth.de

Alexander Dotor  
Angewandte Informatik 1  
Universität Bayreuth  
D-95440 Bayreuth  
alexander.dotor@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 present a mapping of feature models and configurations to executable model elements which is one step towards an overall model driven process for product line engineering. We present a tool called MODPLFeaturePlugin, which allows the user to establish a mapping between a distinct feature model and the domain model of the software system realized in Fujaba and also to create code for specific product configurations.

Keywords
version control, product lines, model-driven development, configuration management, code generation, feature mapping

1. INTRODUCTION
The term model-driven development [13] 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 explicitly. In the context of software product lines [5], 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 [1], [14], [10]. 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 project called MOD2-SCM, we develop a model-driven product line for Software configuration management (SCM) systems [4]. 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. This project is an example for the application of the general model-driven development process of software product lines, which is supported by MODPLFeaturePlugin.

The domain model of our product line consists of both class diagrams and behavioral diagrams developed in Fujaba. In the following we present MODPLFeaturePlugin which enables the Fujaba user to map elements from feature models to fujaba domain models and to generate code for specific product configurations. The plugin consists of two main parts: (1) an extension of the Fujaba editor to provide capabilities to establish a mapping between feature diagram elements and domain model elements and (2) a preprocessor for the Fujaba codegenerator, which is able to generate code for specific product configurations.

Even though the MODPLFeaturePlugin was developed within the MOD2-SCM project, it is by no means specific to the SCM domain and can be applied to all Fujaba models being composed of class diagrams and story diagrams.

2. MAPPING FEATURES TO FUJABA MODEL ELEMENTS
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 be mapped automatically to the domain model, and code for the specific feature selection should be generated. In the context of product line engineering, feature models are widely used. The diagram in Figure 1 depicts our proposal of a model-driven development process of software product lines, which is supported by MODPLFeaturePlugin. Since there is no tool which supports the whole model-driven development process of software product lines, but several tools supporting single tasks in the whole development process, our work shows a way how to bridge the gap between feature modelling and domain modelling. The left part of Figure 2 shows the feature model of our model-driven product line of SCM systems. On the right hand side an excerpt of the domain model (particularly the package diagram and
The feature model shown on the left side in Figure 2 was created using the tool FeaturePlugin [1]. It offers the option to export feature models and respective configurations via XML. MODPLFeaturePlugin is able to import feature models and configurations exported from FeaturePlugin.

Our approach establishes a mapping between feature model elements and their respective realizations in the domain model through attaching annotations containing the feature’s unique identifier to the domain model elements. Fujaba’s metamodel offers basic support for annotations by providing the class UMLTag. Instances of UMLTag can be attached to any element derived from UMLIncrement (see Figure 3).

A model element can also be decorated with multiple tags. The syntax that is used to describe the tag is similar to Java annotations: Each tag starts with the symbol “@” followed by a string literal which specifies the name of the tag and key-value pairs surrounded by parentheses (e.g.

```java
@name(key=value)
```

Examples are shown in Figure 4, like `@feature(id="directedDeltas")`. The annotation of model elements with the respective feature identifier 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 multivariant architecture manageable. But it is up to the modeller’s discipline to use the feature annotations carefully.

In an extensive way of using feature annotations, the modeller may easily lose track and may face a degree of complexity which cannot be managed anymore. Therefore, sev-
eral aids for the developer have been implemented in MODPLFeaturePlugin.

- First of all, the creation of tags, which annotate model elements, has been made much easier by providing a class FeatureTag, which is derived from UMLTag. It has a fixed name feature and it only support one key-value pair: the unique identifier of the feature model element. MODPLFeaturePlugin also provides a special dialog for creating and editing such FeatureTags. The modeler can only choose between the identifiers of an already existing feature model. The advantage over manually specified feature identifiers is integrity of configurations based on existing feature models and the annotated model elements.

- To keep track which model elements are used to realize a certain features, MODPLFeaturePlugin highlights all FeatureTags that match a feature when clicking on that specific feature in the tree view (c.f. in Figure 4 the feature Forward Delta has been selected in the tree view). Selected features in the tree view are symbolized by FeatureTags with green background in the Fujaba editor.

- Additionally, the plugin also checks for missing features in the domain model (depicted by yellow triangles in the feature model with an expression mark in the feature tree and the text "[ unused feature "] and it provides several constraint checkers. E.g. highlighting domain elements with feature tags violating an exclusive-or constraint defined in the feature model. The elements displayed with a red background in Figure 4 show that kind of constraint violations.

Additionally, propagation rules for the tagged elements to their dependent elements can be applied, to give the modeler an overview which other model elements (attributes, roles, associations, subclasses etc.) are affected by adding a FeatureTag to a domain model element. The automatically added tags are highlighted with a blue background (not shown in Figure 4). These rules are also applied automatically during the code generation process, cf. section 3. Enumerating the propagation rules is not possible due to space restrictions. All rules and their application can be found in [2].

MODPLFeaturePlugin was primarily designed to work with the Fujaba/Eclipse-SwingUI branch, but it can also be used with the stand-alone version of Fujaba. Please note that Eclipse specific extensions, like the separate view showing the feature tree is not supported in the stand-alone version.

### 3. GENERATING CODE

After a domain model has been tagged with the feature annotations, code for a specific configuration can be generated. Therefore, a configuration created with FeaturePlugin is loaded by our plugin (also via XML import). A configuration is a distinct selection of features from the feature model which describes a concrete product. This configuration is used to control the preprocessor of the codegenerator.

![Figure 5: MODPLPreprocessorCodeWriter in the Codegen2 context.](image)

The preprocessor is a plugin to the Fujaba Codegen2 plugin [8]. Since Codegen2 provides a so called "chain-of-responsibility" to address codewriters for different model elements or purposes (see Figure 5), it was fairly easy to add a preprocessor to this chain. During the code generation process, these feature annotations are evaluated against a given system configuration which is specified in FeaturePlugin [1]. 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. For all model elements which contain tags representing features not part of the current configuration no code is generated. This process is very similar to the preprocessor step in compilers. As a consequence we face the same problems as compiler preprocessors: it is very easy to produce syntactically wrong code. Therefore we set up several consistency rules [2] that have to be met to ensure syntactical correctness of the resulting code.

### 4. RELATED WORK

Czarnecki et al. describe in their work about Mapping Features to Models [6] a way to establish a bidirectional mapping between feature models and Ecore elements, based on Ecore class diagrams (see [14]). It uses many of the same notations and display elements as a previous version named FeaturePlugin [1]. 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. Furthermore a 1:1 mapping of a feature model to a class model is not appropriate in many cases: E.g. cross-cutting feature such as "Synchronization" in our example shown in the feature model in Fig. 2, mapping features to model elements of different granularity such as classes or methods - not every feature may be realized by a class. But also...
UML class diagrams do not provide capabilities to model variability in terms of cardinality of association ends. The realization of a feature could require a to-one association, whereas the realization of another feature could require a to-many end of the same association. According to the project website, there is no further development of Ecore.fmp.

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) [11]. 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 [9]. The plugin aims at supporting the developer in the complex task of defining mappings between features / configurations and their realizations. FeatureMapper provides no support for consistent model annotations. In MODPLFeaturePlugin, dependencies between model elements are taken into account which are specific to the meta-models for class diagrams and story diagrams. FeatureMapper does not know these dependencies, since it is a generic tool working with any meta-model defined in Ecore. FeatureMapper provides the possibility to configure models and to use that configured models in the further development process. Our approach allows to either configure the code directly (by using the java code generation engine) or to obtain a configured Ecore model, by using the EMF Code Generator [7]. The EMF-Adapter for Fujaba [12] could be a promising approach to combine FeatureMapper and Fujaba, but it is still in a very early stage. In case of a working EMF-Adapter, it could be feasible to try using FeatureMapper on Fujaba models.

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. Usually they work on features provided on a file basis and only cover a small part of the product line process - variant management.

5. CONCLUSION

In this paper we presented MODPLFeaturePlugin, a plugin to Fujaba which aims at supporting a model-driven development process of software product lines in terms of realizing a mapping between elements from feature models and domain
model elements. It provides several aids to the modeler and it supports the generation of specific product configurations based upon feature selection. Currently work is addressed to add several other constraint checkers to the editor plugin and to integrate it with the package diagram editor presented in [3].

6. ACKNOWLEDGEMENTS
We would like to thank our supervisor Prof. Dr. Bernhard Westfechtel for the invaluable input during the development of this plugin and his feedback as a reader of this paper. Further thanks go to (in alphabetical order): Leif Geiger, Ruben Jubeh and Dietrich Travkin for their hints how to extend Fujaba and the codegeneration engine.

7. REFERENCES