Co-Evolution of Models and Feature Mapping in Software Product Lines

Christoph Seidl
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
christoph.seidl@tu-dresden.de

Florian Heidenreich
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
florian.heidenreich@tu-dresden.de

Uwe Aßmann
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
uwe.assmann@tu-dresden.de

ABSTRACT

Software Product Lines (SPLs) are a successful approach to software reuse in the large. Even though tools exist to create SPLs, their evolution is widely unexplored. Evolving an SPL manually is tedious and error-prone as it is hard to avoid unintended side-effects that may harm the consistency of the SPL. The main contribution of this paper is the conceptual basis of a system for the evolution of model-based SPLs, which maintains consistency of models and feature mapping. As further contribution, a novel classification is introduced that distinguishes evolutions by their potential to harm the mapping of an SPL. In addition, multiple remapping operators are presented that can remedy the negative side-effects of evolutions in order to co-evolve the feature mapping. Finally, an implementation of the evolution system in the SPL tool FeatureMapper is provided to demonstrate the capabilities of the presented approach when co-evolving models and feature mapping of an SPL.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*

General Terms

Design, Algorithms

Keywords

Software Product Lines, Evolution, Co-Evolution, Model Transformation, Feature Modeling, Feature Mapping

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPLC '12 September 02 - 07 2012, Salvador, Brazil
Copyright 2012 ACM 978-1-4503-1094-9/12/09 ...$15.00.

1. INTRODUCTION

In feature-oriented software product lines (SPL), functionality of a set of interrelated applications is described in terms of core functionality and multiple individual features [14], which can be combined to build one particular program. The conceptual notion of features is located in the problem space [7]. Individual products of the SPL are created by combining the core functionality with a subset of the available features. To aid this process, all legit combinations of features are described in a variability model. The work of this paper employs feature models for this purpose but the described concepts can be adapted to other approaches such as decision modeling [17] or orthogonal variability modeling (OVM) [14] as well. Finally, implementation assets, such as design models or source code, are located in the solution space [7]. In this paper, problem and solution space artifacts are assumed to be models in the sense of EMOF[1], e.g, implemented in EMF[2] Ecore. All artifacts can be perceived as models provided that a suitable metamodel exists. This is the case for graphical notations such as UML as well as for the Java programming language (using JaMoPP [6]) or textual notations in general, such as the over 100 languages of the EMFText Syntax Zoo[3].

In order to assemble a concrete product from a selection of features, a mapping from the elements in the problem space to parts of the solution space is required. This feature mapping [3] relates logical expressions of features in the feature model to individual parts of solution space artifacts. The feature mapping may be implicit, e.g., through annotation of realization artifacts or conditional compilation, or explicit in a separate artifact containing the mapping information. For the approach in this paper, an explicit model-based feature mapping is assumed. Besides mapping features to entire solution space assets (e.g., classes), the mapping model may also be used to target mere fractions of these assets (e.g., methods). Creating and maintaining the feature mapping is both tedious and error-prone so that tool support is a great aid in the development of an SPL. One tool in this domain is FeatureMapper[4] [7], which operates on arbitrary EMF Ecore-based models.

Similar to other software systems, SPLs have to change

[1] `http://omg.org/mof`
[2] `http://eclipse.org/emf`
[3] `http://emftext.org/zoo`
[4] `http://featuremapper.org`

over time in order to meet new requirements or to stay satisfactory, which is referred to as evolution [11]. However, the evolution of SPLs poses significant challenges as it is easy to accidentally harm the feature mapping, which might render the feature model and the realization artifacts inconsistent and, thus, prevent the creation of certain products. For example, if an implementation asset is deleted but a mapping to the now missing element remains in the system, products that include the referencing combination of features will be invalid. Furthermore, modifying solution space assets may harm the mapping as well, e.g., when parts of a Java method are extracted but the original mapping is not extended to include the new method. Due to these reasons, evolutions can not be applied to elements of an SPL without jeopardizing the feature mapping in the general case.

In this paper, three contributions are presented to address this problem. First, a novel classification for evolutions in SPLs is introduced that captures the effects of model modifications on the feature mapping. Second, the conceptual basis for the co-evolution of models and feature mapping is presented and, third, an implementation that can be used in practical scenarios is provided. Throughout this paper, we assume that an SPL is in a consistent state before evolutions are applied and that its features are properly modularized in the solution space.

The remainder of the paper is structured as follows. In Section 2, various evolutions for the problem and solution space are introduced. In Section 3, the novel classification for evolutions in SPLs is presented and the introduced evolutions are classified. In Section 4, remapping operators are introduced, which are employed to modify existing mappings in order to remedy the negative side-effects of evolutions in SPLs. In Section 5, the presented evolutions are complemented with adequate remapping operators where necessary. Furthermore, the implementation of the presented concepts within FeatureMapper is discussed in Section 6 and an evaluation of the approach is presented in Section 7. Finally, work related to the area of SPL evolution is examined in Section 8 before the paper is concluded in Section 9.

2. EVOLUTIONS IN PRODUCT LINES

A number of evolutions for the problem as well as the solution space is introduced in the following, which is later used to demonstrate the presented classification for evolutions in SPLs and the introduced remapping operators. In the problem space, the only type of model that is regarded for evolution within this approach is the feature model. However, in the solution space, a multitude of different models has to be targeted because of the large number of different types of implementation artifacts. Due to this elementary difference, evolutions are provided for either space separately depending on where the changes originate. In the course of this paper, it is assumed that changes are performed on the SPL itself but not on individual products. Furthermore, only changes to models but not to their respective metamodels are considered.

The presented evolutions are exemplified by realistic application scenarios taken from the evaluation of the described evolution system published in [19]. In the evaluation, an SPL for the software part of a multimedia system in a car is subject to various modifications. Focussing on the parts relevant to demonstrating the presented evolutions, the SPL consists of various options for an audio player (radio, CD player, cassette player), which each use the speaker system of the car as well as a display built into its middle console. Besides modifications on these features and their realization, the SPL is extended by a personal navigation device.

A variety of evolutions for different models is presented so that the wide applicability of the co-evolution approach can be demonstrated later on. In the following, these evolutions are explained and illustrated by example.

2.1 Evolutions from the Problem Space

Within the presented approach, the feature model is the only target of evolutions originating in the problem space. Other assets located in the problem space (such as use cases, functional requirements etc.) are not regarded. In total, five different evolutions are presented for the problem space.

Duplicate Feature copies the selected feature (*OriginalFeature*) and adds the clone (*CopiedFeature*) as sibling of the selected feature to the feature model. This evolution is used to create a new feature that is largely similar to an already existing feature in terms of the solution space elements it references. For instance, in the aforementioned automotive multimedia SPL, a basic version of a feature for a personal navigation device may be added as clone of the audio player. In this case, the similarity of the features is exploited as both employ the speaker system and the display of the car.

Insert Feature creates an entirely new feature (*NewFeature*) in the feature model as child element of the currently selected feature. The evolution is intended to be employed when creating a new feature that has very little to no similarity to already existing features. For example, the newly created personal navigation device depends on digital geographical maps in order to calculate routes. A mandatory feature representing these maps may be added as child of the personal navigation feature using *Insert Feature*.

Split Feature is used to model a feature in a more fine-grained way than was originally intended by distributing its functionality to an arbitrary number of constituents. For this purpose, a specified number of child features (*SplitFeatures*) is added to the affected feature (*OriginalFeature*). Semantically, the sum of all child features comprises the same functionality as the original feature. An example of *Split Feature* can be found in Figure 1, where the geographical mapping material provided along with the personal navigation device is split into smaller units representing different continents.

Figure 1: Example of *Split Feature* to divide feature `Maps` into parts for different continents.

Remove Feature can be applied to delete a selected feature (*OriginalFeature*) from the feature model.

Remove Feature and Owned Assets is a more complex evolution, which deletes a selected feature (*OriginalFeature*) from the problem space and removes all solution space assets that were used exclusively by this feature but no other feature (*OwnedAssets*). Due to this characteristic, the evolution is used to completely delete a feature and its implementation from an SPL. The effects of *Remove Feature and Owned Assets* are presented in Figure 2, where the obsolete feature `Cassette Player` is removed from the automotive multimedia SPL. Along with the feature of the feature model,

the exclusively used implementation assets in the form of a UML class and aggregation are deleted as well.

Figure 2: Example of *Remove Feature and Owned Assets* to delete the feature CassettePlayer and its implementation.

2.2 Evolutions from the Solution Space

For the solution space, evolutions for multiple types of models have to be provided as principally any kind of model-based artifact may be used as target of a feature mapping. In the course of this paper, two different types of models serve as representatives of the software development process. The Unified Modeling Language (UML) is employed as example of design models via Eclipse MDT[5] and Java source code is converted to a model representation by use of JaMoPP[6] [6] to demonstrate textual formats.

Replace Method with Method Object for UML is used when the functionality described by a method (*Method*) has to be relocated to a separate class. During this process, a new class is created (*NewClass*) and a reference (*Reference*) to it from the original class (*OriginalClass*) is added. Furthermore, the selected method is moved from the original class to the new class. Within the automotive multimedia SPL, the class representing the CD player logic is capable of decoding audio CD text information. In the evaluation scenario, the CD player is aided with additional capabilities to decode the MP3 audio format, for which a new child class is created. Due to this reason, the method to decode audio CD text is no longer needed in the base class for the CD player and can, thus, be moved to a separate class.

Extract Method for Java is employed to restructure the contents of a method (*OriginalMethod*) when it becomes convoluted due to too many lines of code. For the evolution to be applicable, multiple lines of code of a method have to be selected (*Statements*), which are then moved to a newly created method (*NewMethod*). At the site of the original code, a call to the new method is added (*MethodCall*). Thus, the semantics of the code fragment are maintained while improving its structure.

Rename Element, as further evolution for Java, allows to assign new names to entities such as classes, methods or attributes (*Element*). The evolution automatically updates all relevant occurrences of the name (*References*), such as method calls or references to attributes, in order to maintain the validity of the code fragment. For example, the class `CDPlayer` of the automotive multimedia SPL may be renamed to `MultiFormatCDPlayer` in order to signal its increased capability to decode MP3 files.

3. CLASSIFICATION OF EVOLUTIONS

In order to adapt an SPL to changed requirements, a variety of different model evolutions might have to be employed. Thus, evolutions may add new model elements as well as modifiy or delete existing ones. Changes may appear in models of the problem space, the solution space or both of them simultanously. According to these criteria, the effects of an evolution on the SPL in its entirety differ. In order to precisely address individual types of model evolutions by their effect on an SPL, a classification is required that captures the effects of an evolution on all relevant models and the feature mapping.

3.1 Classifications in the Literature

Unfortunately, classification systems in the literature are unsuitable for this purpose. For example, a classification by whether an evolution preserves semantics is defined for parts of an SPL but not for its entirety covering the semantics of feature model, feature mapping and solution space assets. The most comprehensive classification of this type was defined by Borba et al. [4] along with a formal theory of product line refinement. They classify evolutions by their effect on the semantics of an SPL by considering feature model, configuration knowledge and asset mapping, where the latter two in combination are equivalent to what is referred to as feature mapping in this paper. However, they provide only a brief example of a modification to a realization asset, which modifies the artifact's semantics (i.e., renaming a menu option for the user interface). Apart from that, the semantics of realization assets in the solution space is not regarded. However, the realization assets ultimately define the behavior of products derived from the SPL. Thus, when attempting to classify evolutions by whether they preserve semantics of an SPL, the effects on semantics of solution space assets has to be captured as well. Unfortunately, this seems practically impossible due to the sheer number of different solution space artifacts and their varying definitions of semantics. For instance, source code defines semantics as program behavior whereas the semantics of a documentation format are its contents as they are perceived by the reader. Due to these reasons, it is not feasible to apply a uniform classification by semantic preservation for evolutions encompassing an entire SPL.

Further classifications in the literature group model evolutions by various other characteristics such as the relation of goal and target model. In [15], mapping and update transformations are distinguished. A mapping transformation relates the elements of a source model to the elements of a different target model, whereas an update transformation modifies a given model in place. A very similar classification is presented in [12] where endogenous and exogenous transformations are differentiated. Endogenous transformations rephrase elements of one language to different elements of the very same language. In contrast, exogenous transformations are a translation of elements from a source language to elements of a target language. Furthermore, a distinction of horizontal and vertical evolutions is presented in [12] to capture a change in abstraction level. Horizontal evolutions maintain the current level of abstraction whereas vertical evolutions make the transformed artifacts either more concrete or more abstract.

In addition, the reversibility of evolutions is used for classification in [1] with the groups for unidirectional and bidirectional modifications. Unidirectional evolutions extend the configuration options of the feature model in the problem space of an SPL and, thus, can not be reversed without disallowing certain products. In contrast, bidirectional evolutions

[5] `http://eclipse.org/uml2`
[6] `http://jamopp.org`

maintain configurability so that they can be reversed. As the semantics of the problem space are equivalent to the configuration options of the feature model, this classification is closely related to a classification by semantics preservation and, thus, suffers from the same problems as described above.

As all evolutions presented in this paper work directly on the provided models, each evolution uniformely had to be considered a horizontal, endogenous update transformation when the classifications above were employed.Furthermore, the general problem of the presented classifications is that they consider only one model at a time but not the entirety of all models encompassed by an SPL. It would be problematic to classify changes that, e.g., refine one model and make another one more abstract within a single evolution step. Thus, above classifications are not feasible to capture the effects of evolutions on the feature mapping and models of an SPL.

3.2 Classification by Semantical Extent of Model Changes

For an adequate classification of evolutions in the context of an SPL, modifications of problem and solution space models as well as effects on the feature mapping have to be considered. Due to this reason, a novel classification is introduced that uses the semantical extent of model changes as criterion. For this purpose, evolutions are categorized into two groups: *intraspatial* and *interspatial* evolutions. Intraspatial evolutions merely affect the space they originate from (i.e., either problem or solution space) but not the feature mapping. The semantical extent of changes in interspatial evolutions reaches beyond the boundaries of the originating space. This group of evolutions is further divided into interspatial evolutions of the first and second degree. *Interspatial evolutions of the first degree* affect their originating space as well as the feature mapping and *interspatial evolutions of the second degree* additionally modify elements in the opposite space. In Figure 3, a schematic overview of these groups of evolutions is presented.

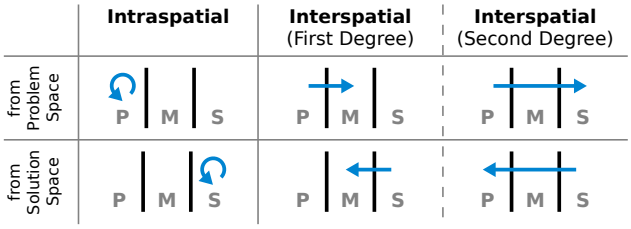

Figure 3: Schematic overview of the classification by semantical extent of model changes.

An example for an intraspatial evolution in the problem space could be the change of an optional to a mandatory feature, which affects configuration options but not the mapping to solution space elements. Likewise, restructuring internal details of a mapped element (e.g., statements in a mapped method) qualifies as intraspatial evolution in the solution space. An example of interspatial evolutions in either space is the deletion of elements, such as with *Remove Feature*. Further examples for members of each category can be found in Section 5.

In order to attribute one particular evolution to its appropriate group in the classification, structural changes in the model as well as the intent of the evolution have to be considered. For an inspection of structural changes, the operations an evolution performs are separated into the three abstract groups of adding, modifying (e.g., setting attribute values) and removing elements. If an evolution exclusively consists of operations that merely modify existing elements, the evolution is categorized as intraspatial as the mapping is not affected. As the approach works on models, it is possbile for artifacts to hold references to elements of other models, which are not affected by name changes so that a modification of e.g., a feature name does not harm the feature mapping. When elements are removed from a model, the evolution is necessarily interspatial as potential mappings of the removed element have to be deleted as well.

However, when adding new elements to a model, the mere inspection of structural changes does not suffice for an adequate classification but the intent of the evolution has to be considered instead. When adding elements to a model, it has to be determined whether the newly created element has a logical relation to previously existing ones (e.g., having largely similar functionality). If that is the case, the evolution is interspatial as it crosses the boundary to the mapping. Otherwise, the evolution is intraspatial (see comparison of *Insert Feature* and *Duplicate Feature* in Section 3.3).

Once it was established that an evolution is interspatial in nature, it remains to be determined whether it is of the first or the second degree. For this purpose, the space opposite to the originating space of the evolution has to be examined for model changes. If at least one model was modified in the opposite space as well, the evolution is of the second degree, otherwise it is of the first degree. The steps for classifying evolutions are summarized in a decision diagram in Figure 4.

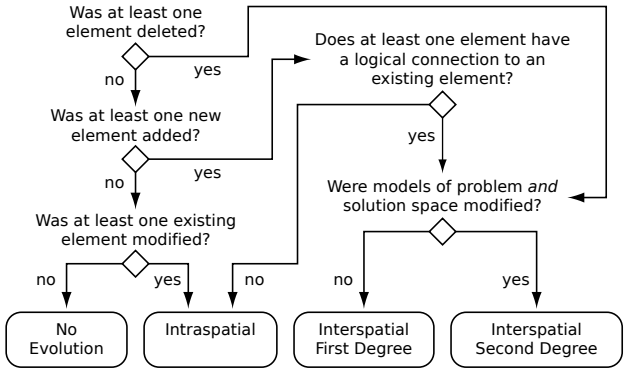

Figure 4: Decision diagram for the classification by semantical extent of model changes.

3.3 Classification of Presented Evolutions

According to the presented classification, the evolutions of Section 2 can be attributed to their respective classes.

In the problem space, *Duplicate Feature* clones a selected feature by copying it. Due to the semantical connection of the original and cloned element, the evolution is interspatial of the first degree. *Insert Feature* merely adds a new feature to the feature model that has no connection to previously existing ones so that the evolution is considered intraspatial and, thus, does not require remapping. Even though *Duplicate Feature* and *Insert Feature* perform similar structural changes, the evolutions have a different intent so that they are classified as interspatial and intraspatial respectively. *Split Feature* distributes functionality of a selected feature to

various newly created child features. Thus, a logical connection between original and new elements can be established and the evolution is considered interspatial. *Remove Feature* deletes elements in the problem space so that it is interspatial by definition. *Remove Feature and Owned Assets* potentially deletes elements in the problem as well as in the solution space. Thus, both spaces are affected by *Remove Feature and Owned Assets*, which makes it an interspatial evolution of the second degree.

In the solution space, *Replace Method with Method Object* for UML creates a new class and a reference to it before the selected method is move to the new class. In order to classify the evolution, its two constituent operations have to be examined. The move operation merely modifies a previously existing element. Futhermore, the newly created class has a clear semantical link to the already existing class as it assumes part of the original functionality, which is signaled by the newly added reference between the original and new class. Thus, the evolution is interspatial of the first degree. *Extract Method* for Java adds a new element and a method call at the site of the original code, which establishes a semantical link between the new method and the previously existing one. Hence, the evolution is interspatial of the first degree. *Rename Element* for Java merely modifies existing elements by changing their name so that the evolution is considered intraspatial and does not require remapping.

4. REMAPPING

As intraspatial evolutions do not affect the feature mapping, they are unproblematic for consistency of the feature mapping. However, for interspatial evolutions, the feature mapping has to be modified in order to ensure consistency as these evolutions affect the feature mapping as a side-effect. For this purpose, various remapping operators are provided for the problem as well as the solution space, which can be used in conjunction with interspatial evolutions to co-evolve the feature mapping. The intent of these remapping operators is to modify existing feature mappings in accordance with the performed model evolutions but not to introduce entirely new mappings. In order to disambiguate remapping operators for the problem and solution space, they are referred to as "feature remapping operators" and "object remapping operators", respectively.

4.1 Feature Remapping Operators

The problem space end of a mapping is the feature expression, which is a logical term referencing various features. Thus, remapping operators in the problem space modify this term in order to alter the mapping in accordance with the performed evolution. Some remapping operators allow to specify multiple targets for a single operation. In these cases, users may choose a subset of all targets to use as effective targets for each affected mapping provided that the designers of an evolution granted this freedom of choice as they assessed that it is reasonable in this case. In sum, five different feature remapping operators are provided for the problem space. Each operator is specified by a number of steps that have to be carried out in order to perform the remapping. Graphical examples of all feature remapping operators can be found in Figure 5.

Move Feature Mapping is used to relocate the mapping of one feature to another feature by performing the following steps:

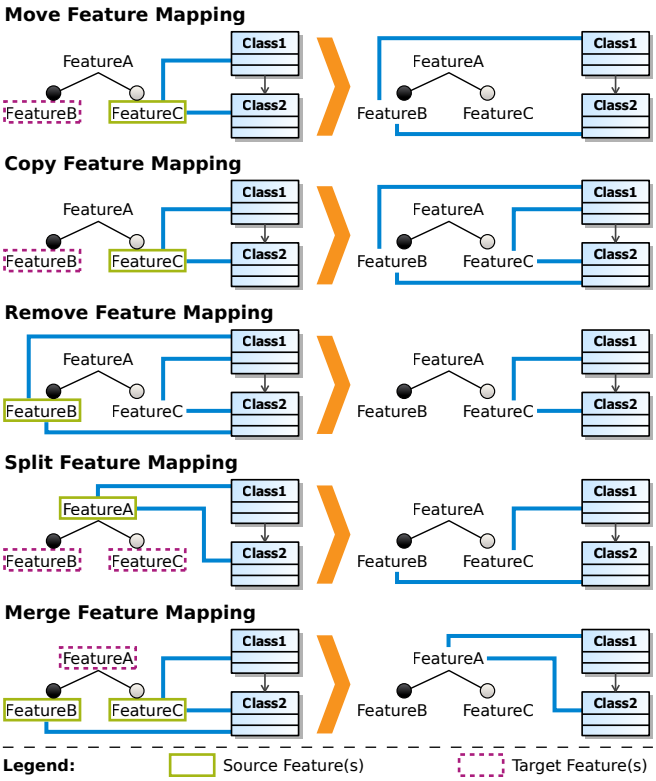

Figure 5: Graphical examples of all feature remapping operators using UML in the solution space.

1. Parameters: f_s the source feature, F_t the set of potential target features
2. Let M be the set of all mappings containing a reference to feature f_s.
3. For each mapping $m \in M$:
   (a) If permissible, have users specify the set $F_{et} \subseteq F_t$ of effective target features, otherwise $F_{et} := F_t$.
   (b) Let t be the current term of mapping m.
   (c) For each feature $f \in F_{et}$:
      i. Create a copy of term t as term t_i.
      ii. Replace all occurences of feature f_s in term t_i with feature f.
   (d) Concatenate all terms t_i using the OR operator and set the result as term for mapping m.

Copy Feature Mapping is employed to logically duplicate a mapping of one feature for another one. The steps to carry out this procedure are identical to those for *Move Feature Mapping* except for step 3(d) where the concatenation of all terms t_i has to be concatenated with the original term of mapping m using the OR operator before setting the result as term for mapping m.

Remove Feature Mapping deletes a particular feature from a mapping by rephrasing the feature expression carrying out these steps:

1. Parameters: F_s the set of source features
2. For each feature $f \in F_s$:
   (a) Let M be the set of all mappings containing a reference to feature f.
   (b) For each mapping $m \in M$:

i. Let t be the current term of mapping m.
ii. Remove all occurrences of f and all superfluous operators from term t.

(c) If term $t = \epsilon$ (the empty term), delete mapping m from the system, otherwise set the term of mapping m to term t.

When removing the references to feature f in step 2(b)ii, some operators may temporarily be rendered invalid (e.g., the AND/OR operators with one or less remaining operands or the NOT operator with no operand). In these cases, the affected operator is removed and the remaining operands (if any) are relocated to the respective superordinate operator. This procedure might reduce the entire term to ϵ (the empty term). In this case, the mapping is deleted in step 2(c).

Split Feature Mapping is used to distribute the mapping from one feature to multiple others. In detail, the following steps have to be carried out for this remapping:

1. Parameters: f_s the source feature, F_t the set of potential target features
2. Let M be the set of all mappings containing a reference to feature f_s.
3. For each mapping $m \in M$:
   (a) Have users select an operator $o \in \{AND, OR\}$.
   (b) If permissible, have users specify the set $F_{et} \subseteq F_t$ of effective target features, otherwise $F_{et} := F_t$.
   (c) Create a term t as concatenation of all features $f \in F_{et}$ using operator o.
   (d) Set the term of mapping m to term t.

Merge Feature Mapping combines mappings from different source features on one target feature. Through this operation, the changes created by *Split Feature Mapping* can be nullified. The steps for the remapping procedure are as follows:

1. Parameters: F_s the set of source features, f_t the target feature
2. Let M be the set of all mappings containing a reference to at least one feature $f \in F_s$.
3. For each mapping $m \in M$:
   (a) Let t be the current term of mapping m.
   (b) Replace all occurrences of feature $f \in F_s$ in term t with feature f_t.
   (c) Set the term of mapping m to term t.

Whether applying a feature remapping operator alters semantics of a feature expression depends on the concrete situation. For example, a feature in a feature expression may be replaced by an OR expression of other features as result of *Move Feature Mapping* as well as *Split Feature Mapping*. In the former case, semantics would be altered whereas, in the latter case, semantics would be maintained if the various features are the split parts of *Split Feature*.

4.2 Object Remapping Operators

In contrast to the problem space, remapping in the solution space is performed mostly by altering the reference to a solution space element in a mapping. In sum, there are three object remapping operators in the solution space. Graphical examples of these operators are presented in Figure 6.

Move Object Mapping relocates the mapping of a particular solution space element to one or multiple other elements with the following steps:

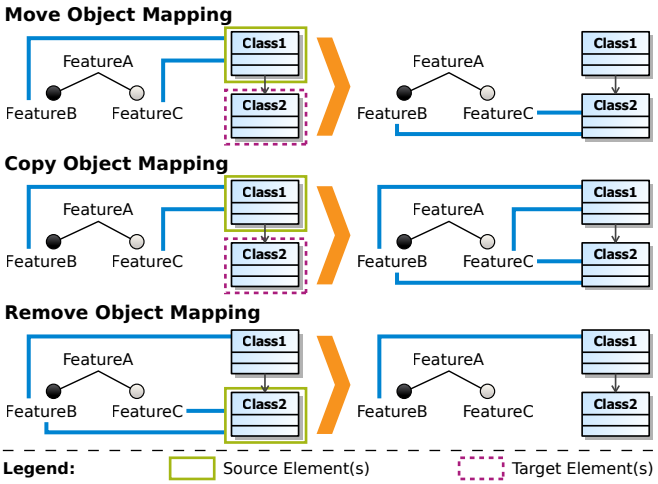

Figure 6: Graphical examples of all object remapping operators using UML in the solution space.

1. Parameters: e_s the source element, E_t the set of target elements
2. Let $m1$ be the mapping referencing element e_s.
3. For each element $e_t \in E_t$:
   (a) Let $m2$ be the mapping referencing e_t.
   (b) If $m2 = \epsilon$ (the empty mapping):
      i. Create a copy of mapping $m1$ as mapping $m3$.
      ii. Set the target element of mapping $m3$ to element e_t.
   (c) Else
      i. Concatenate the terms of mapping $m1$ and mapping $m2$ using the OR operator as term t.
      ii. Set the term of mapping $m2$ to term t.
4. Delete mapping $m1$ from the system.

Copy Object Mapping faces similar issues when duplicating the mapping of one solution space element to another. However, unlike with *Move Object Mapping*, the original mapping is not deleted in order to perform the copy operation. Thus, the steps to perform are equivalent to those for *Move Object Mapping* except for step 4, which is not performed.

Remove Object Mapping deletes the mapping of one or more solution space elements by deleting the respective mappings from the system provided that they exist. This procedure is performed by the following steps:

1. Parameters: E_s the set of source elements
2. For each element $e_s \in E_S$:
   (a) Let m be the mapping referencing e_s.
   (b) Delete mapping m from the system.

For all feature and object remapping operators affecting more than one mapping, it might be possible to exclude individual mappings from remapping (e.g., copy only a subset of all affected mappings). However, this degree of freedom may jeopardize SPL consistency in some cases. For example, not moving all mappings away from an element deleted during an evolution would render the SPL inconsistent. Due to this reason, designers of an evolution have to explicitly allow exclusion of individual mappings when they assessed that it is reasonable in this case.

5. CO-EVOLUTIONS IN PRODUCT LINES

With the presented remapping operators, it is possible to aid the evolutions of Section 2 with adequate remapping support where necessary. *Insert Feature Mapping* in the problem space and *Rename Element* for Java in the solution space were identified as intrapstial evolutions and, thus, do not require remapping. The remaining evolutions are interspatial and have to be extended by remapping operations in order to be applicable in SPLs without endangering the feature mapping. For the adequate remapping operations, the names of participating roles in an evolution from Section 2 are used.

With *Duplicate Feature*, the original feature should be cloned. Thus, it is appropriate to also duplicate the respective mappings. For this purpose, the evolution is complemented by the *Copy Feature Mapping* operator, which is parameterized with $f_s = \mathit{OriginalFeature}$ and $F_t = \{\mathit{CopiedFeature}\}$. In the case of duplicating the feature for the audio player as basis for a personal navigation device in the automotive multimedia SPL, only those mappings might be copied to the new feature that concern the speaker system and the display as these parts are used by the audio player as well as the personal navigation device.

For the *Split Feature* evolution, the analog *Split Feature Mapping* operator is utilized with parameters $f_s = \mathit{OriginalFeature}$ and $F_t = \mathit{SplitFeatures}$. In the example of splitting a feature for geographical maps to features for specific continents, users might choose to redirect the original mapping to geographical mapping material for countries such as Germany or Poland to the new feature `Europe`, China and Japan to `Asia` as well as USA and Canada to `America` by selecting only those targets for the respective mappings.

Remove Feature is accompanied by *Remove Feature Mapping*, which is employed to delete the mapping of all no longer existing features, with the parameter $F_s = \{\mathit{OriginalFeature}\}$.

Remove Feature and Owned Assets deletes elements from the problem and solution space. Due to this reason, the evolution requires remapping in both spaces. In the problem space, *Remove Feature Mapping* is applied to delete the mappings of the removed feature with $F_s = \{\mathit{OriginalFeature}\}$. In the solution space, *Remove Object Mapping* is employed to delete the mapping of all assets that were removed from realization artifacts so that $E_s = \mathit{OwnedAssets}$. In the example of deleting the cassette player and its assets from the SPL, the respective mappings are deleted along with the feature for the cassette player and its representation in the UML design model.

Replace Method with Method Object for UML is complemented by *Copy Object Mapping* to duplicate the mapping of the original class to the newly created method object as well as the association to it with parameters $f_s = \mathit{OriginalClass}$ and $F_t = \{\mathit{NewClass}, \mathit{Reference}\}$. In addition, the mapping of the relocated method is extended to the new containing class and its incoming association by use of *Copy Object Mapping* as well so that they are part of all SPL variants where the method is required ($f_s = \mathit{Method}$ and $F_t = \{\mathit{NewClass}, \mathit{Reference}\}$).

Extract Method for Java is accompanied by *Copy Object Mapping* to clone the mapping of the method originally containing the extracted lines of code to the newly created class and the call to it ($f_s = \mathit{OriginalMethod}$, $F_t = \{\mathit{NewMethod}, \mathit{MethodCall}\}$). Furthermore, the new method and the call to it should be part of all variants that originally required at least one of the extracted statements. Thus, the mapping from the statements is extended using the *Copy Object Mapping* operator with $f_s = \mathit{Statements}$ and $F_t = \{\mathit{NewMethod}, \mathit{MethodCall}\}$.

6. IMPLEMENTATION

The concepts described in the paper were implemented as part of the Eclipse-based FeatureMapper[7]. The tool allows to create and maintain model-based SPLs and is able to assemble assets representing selected features for an individual product. In order to realize the evolution system for SPLs, the tool Refactory [16] by Reimann et al. was employed for the modification of models. Despite its name, Refactory is not limited to semantic preserving changes but can be used for the more general task of model evolution as well. In order to perform remapping, the post processor mechanism of Refactory was used, which enables execution of additional operations (i.e., remapping plans) immediately after an evolution. An extension point of Eclipse is used to register a remapping plan with a particular evolution. Immediately after the evolution was executed, all remapping operators contained in the remapping plan are executed sequentially. All eight described remapping operators can be employed in order to maintain consistency of the mapping. Where applicable, the remapping plan is accompanied by a user interface for its contained remapping operators that allows customization of the remapping process. However, designers of an evolution have to explicitly enable this option when they assessed that it is reasonable in order to prevent users from creating illegal remappings (e.g., not all mappings of duplicated elements may have to be copied to maintain consistency).

Within the evolution system, a declarative textual format is used to specify remapping plans, which was designed to integrate with Refactory's role-based [20] specification of evolutions. The so called object remapping specification (ORSpec) of the evolution system uses the roles specified in an evolution to express the operands of particular remapping operations within a remapping plan, e.g. in Listing 1. The format is used to specify a sequence of remapping operators that should be executed as remapping plan for a particular evolution. Besides a textual description of the remapping operation for users, it is possible to specify two types of options: "OPTIONAL" states that the remapping may be skipped at users' request and "SELECTABLE_TARGETS" permits users to create a subset $F_{et} \subset F_t$ of effective targets for the remapping operation according to Section 4. Alternative to using an ORSpec, remapping plans may be created in source code by calling the remapping operators on elements of the problem or solution space programmatically. The respective Java classes are registered via extension point for a particular evolution in a similar manner as with the ORSpec.

Using above techniques, 37 different evolutions were realized within the evolution system for SPLs including the evolutions presented in Section 2 (a full list can be found in Chapter 3.1 of [19]). In sum, 16 of the implemented evolutions are intraspatial. The remaining 21 interspatial evolutions are complemented by adequate remapping plans. In the problem space, further evolutions were realized ranging from basic evolutions to change the variation type of features to complex ones, such as *Merge Features*, which melds multiple features

[7] `http://featuremapper.org`

```
OBJECT REMAPPING FOR <ExtractMethod>

STEPS {
  COPY MAPPING: OriginalMethod => NewMethod, MethodCall;
  COPY MAPPING: Statements => NewMethod, MethodCall {
    description = "From Statements (optional)";
    options = OPTIONAL, SELECTABLE_TARGETS;
  };
}
```

Listing 1: Object remapping specification for *Extract Method* for Java.

into one in order to create a more coarse-grain abstraction. In the solution space, various adaptations of refactorings described by Fowler in [5] were implemented for UML models as well as Java source code. Additionally, evolutions to add, remove and restructure contents of a textual documentation format called DocBooklet, which is a derivative of DocBook[8], are provided. Furthermore, the system can be extended with further evolutions and remapping operators if necessary by use of Eclipse extension points.

7. EVALUATION

The intent of the evaluation of the evolution system was to assess the practical applicability and usefulness of the remapping operators in conjunction with concrete evolutions in an SPL setting. For this purpose, the implemented interspatial evolutions were aided with adequate remapping plans and implemented within the tool FeatureMapper. A fictitious SPL for the multimedia system in a car was devised and the goals of two major revisions of the SPL were defined. The authors modified the SPL to meet these goals using the implemented evolutions or manually where necessary. Besides the feature model in the problem space, the SPL consists of three different types of solution space models: UML, an automatically created model representation of Java source code and the textual DocBooklet format for documentation. The extent and number of presented models were deliberately held small to make it easier to grasp the effects of an evolution. Nevertheless, essential artifacts of the software development process are demonstrated by using these three types of solution space models. The SPL was evolved in the course of two iterations, which are explained in the following. An overview of the changes performed on the feature model in the evolution iterations is presented in Figure 7 and the full report on the evaluation can be found in Chapter 5 of [19].

In the initial situation of the evolution process, the feature model of the SPL consisted of merely two top level features for the on board computer and an audio player. The audio player was further divided into child features for a radio, a cassette player and an audio CD player. The solution space was comprised of a UML design model, various Java classes and a user manual in the DocBooklet format.

During the first iteration of evolution, three major changes were performed. First, the cassette player and its implementation were removed using *Remove Feature and Owned Assets* and its subsequent remapping procedure.

In the second step, a feature for an MP3 CD player as alternative to the audio CD player was created using *Split Feature*. Its subsequent remapping distributed the existing

[8] `http://docbook.org`

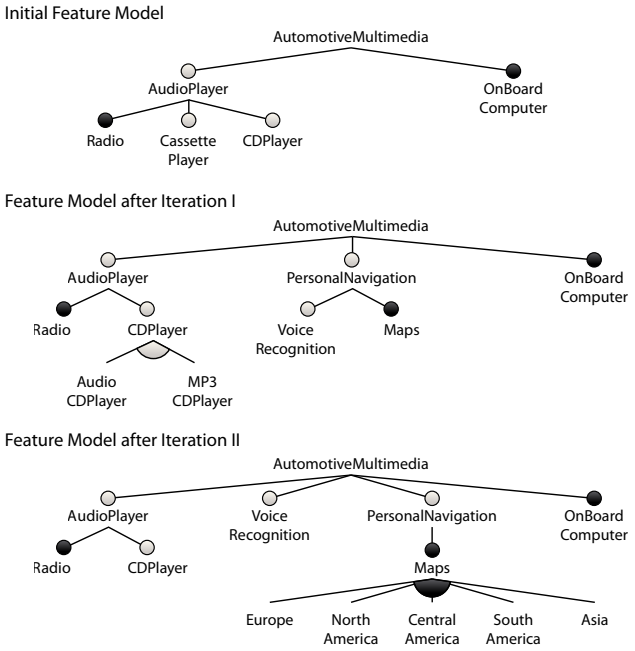

Figure 7: Feature model of the evaluation scenario over the course of the evolution iterations.

mapping to solution space assets so that some parts were referenced exclusive by either one of the CD players whereas others were shared. The solution space was further modified by *Replace Method with Method Object* for UML and *Duplicate Section* for DocBooklet. The content of the new section in the user manual was altered and additional elements in the design model and the respective mappings to them were added manually.

In the third major modification during the first iteration, a personal navigation device was added as configuration option for the SPL. The *Duplicate Feature* evolution was employed to create the new feature by copying the feature for the audio player and the majority of its mappings as both the audio player and the personal navigation device utilize the car's speaker system and its display. Furthermore, child features for the voice recognition to control the navigation device and the geographical mapping material required for planning routes were added using *Insert Feature*. As part of this change, a new solution space model was added that represents the geographical maps and a mapping from the respective feature was added manually.

To conclude the first evolution iteration, the implementation of the Java class responsible for constructing the graphical user interface to control the multimedia system was restructured using *Extract Method* for Java and its subsequent remapping operation.

In the second interation, another three major changes were performed. First, the previous distinction of audio and MP3 CD player was revoked by merging both features and their respective mappings by employing *Merge Features*.

As second modification, the capabilities of the voice recognition feature were increased to not only control the personal navigation device but also the audio player. For this purpose, the feature model was modified by the intraspatial evolution *Pull Up Feature* and the UML design model was adapted using the aforementioned *Extract Super Class* evolution. To

complete this evolution step, three references in the UML model had to be added manually and a mapping needed to be altered. This was due to the reason, that the creation of the voice recognition feature required creating entirely new model elements that had to be mapped immediately and no evolution existed to handle this case.

During the third modification, the feature for the mapping material of the personal navigation device was refined to represent different continents. The mapping of the original feature was distributed to the newly created child features.

In addition to the three major changes on the SPL in the second iteration, a minor modification of the CD player's internal structure was performed using *Inline Method Object* for UML, which relocates a method from one class to another, and its subsequent move of the feature mapping.

During the evaluation, 20 occasions arose for the use of evolutions on the feature model in the problem space as well as the UML, Java and DocBooklet models in the solution space. Within the evolution process, 16 different evolutions out of the 37 implemented in the evolution system were applied including intraspatial as well as interspatial evolutions.

For each interspatial evolution, an adequate remapping procedure was employed so that no existing mapping needed to be altered as result of employing an evolution. However, as the creation of entirely new mappings is outside the scope of the remapping operators, it had to be performed manually in some cases. Furthermore, the second modification in the second evolution iteration required manual modification of an existing mapping as completely new elements were added to the solution space that had to be included in a complex feature expression. As this case could principally be handled with the evolution system as well, further evolutions should be implemented to create directly mapped solution space elements in the future. Apart from these exceptions, all cases could be dealt with employing an evolution of the evolution system and its respective remapping operations to co-evolve models and feature mapping in the SPL.

However, the performed evaluation used only few models of a relatively small size and was carried out by the authors themself. To address these threats to validity, it is planned to carry out a more extensive evaluation in the future. Furthermore, a real-world practical example may contain challenges that the presented approach currently can not cope with. For example, changing a mandatory feature to optional may be problematic for SPL consistency if the feature resides on the uppermost level and not all feature expressions describing valid products contained a reference to the mandatory feature. Identifying these issues in a more extensive evaluation will help to improve the presented approach.

8. RELATED WORK

A survey of related work suggested that a system for performing the evolution of SPLs with capabilities similar to the presented one has not been conceptualized or implemented before. However, several authors have treated various different areas of SPL evolution.

Schulze et al. [18] extend the notion of refactoring to SPLs in order to perform variant-preserving modifications on the code basis of an SPL in the solution space within the context of feature-oriented programming (FOP). Even though they have a similar goal of creating an operative approach to SPL modification, the challanges they address are somewhat different from those in this paper through their focus on FOP. As all (partial) assets relevant to a feature are grouped in a feature module with equal name as the feature in FOP (which is an implicit form of feature mapping), changes in the solution space affecting merely a single feature can be performed without risk. However, with an explicit feature mapping model, changes, such as extracting a method within the same class, require the mapping to be adapted to the newly created method. For modifications targeting more than one feature, it is merely stated that references have to be adapted but not how this procedure is performed programmatically. Furthermore, no modifications for the feature model in the problem space are presented.

Borba et al. [4] present a theory for product line refinement describing when evolutions alter the semantics of an SPL. However, they focus on feature model and mapping and do not consider the semantics of individual solution space assets. Furthermore, they do not provide concrete operations for evolution. However, their work serves as basis for that of Neves et al. who present a number of templates for safe product line evolution in [13]. Seemingly, the steps described in the templates were not automated and have to be carried out manually by users. Furthermore, adding entirely new functionality to existing products of an SPL is out of the scope of their work, which distinguishes it from the approach presented in this paper.

Additionally, Heider [8] outlines an iterative process for reactive SPL evolution. He proposes the development of a tool-supported method for iterative, reactive, model-based evolution of SPLs. However, the focus is on analyzing application requirements from various customer-specific product extensions to find similar requirements as basis for new features and not on performing the modifications of an evolution. Furthermore, Heider and Rabiser [9] present tool-support for a rapid feedback loop where custom requirements in application engineering of a particular SPL product are communicated to SPL engineers immediately in order to decide on their relevance for the SPL in its entirety before custom assets for individual products are implemented—performing evolutions is not considered on an operational level.

Moreover, Vierhauser et al. [21] describe their experience with a tool for flexible and scalable consistency checking on variability models in SPLs. As part of their work, they describe various different categories of inconsistencies in SPLs within problem and solution space as well as in between spaces that may result from modifications to the SPL. The authors motivate the need to detect these inconsistencies but are not concerned with fixing them. Unfortunately, technical details on the approach are omitted.

In [10], Jirapanthong and Zisman present an approach to automatically identify semantical traceability links (e.g., refines, implements) between SPL artifacts. However, their approach is currently limited to a predefined set of artifacts, which conflicts with the goal of generic solution space assets in our approach. Anquetil et al. [2] present another approach to trace links between artifacts in an SPL using four dimensions. The *time* dimension describes how an artifact changes through evolution so that it could be used to revert the changes made by an evolution. The *variability* dimension relates problem to solution space artifacts and vice versa. Thus, it is conceptually equivalent to the feature mapping used in this paper. Finally, the *refinement* and *similarity* dimensions capture relations between artifacts of different or the same abstraction levels, respectively. The trace links of

both these approaches could be used to automatically propagate changes on one solution space element to other related elements. For example, modifying the name of a method in a UML class might cause the equivalent change in a Java class automatically. Currently, the procedure for change propagation has to be encoded into an evolution manually. Thus, the work by Jirapanthong and Zisman as well as that by Anquetil et al. might serve as basis for future work.

9. CONCLUSION

The work presented in this paper provides the conceptual foundation for an evolution system for SPLs. As basis, a classification by semantical extent of model changes was presented, which groups evolutions by their potential to jeopardize the feature mapping of an SPL. The category of intraspatial evolutions does not affect the mapping whereas interspatial evolutions have the potential to harm the mapping and, thus, require remapping in order to ensure consistency of the feature mapping. Therefore, eight remapping operators for problem as well as solution space were presented, which are able to remedy the negative side-effects of interspatial evolutions on the feature mapping.

The presented evolutions for various model types were attributed to their appropriate groups in the classification. Furthermore, for each interspatial evolution, the respective adequate remapping steps were explained in order to allow co-evolution of models and feature mapping. The conceptual aspects described in the paper were implemented as part of FeatureMapper and put to test in an evaluation scenario.

10. ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their valuable comments and suggestions. This work has been partially supported by the European Social Fund and the Federal State of Saxony within project VICCI #100098171.

11. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, 2006.

[2] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa. A Model-driven Traceability Framework for Software Product Lines. *Software and Systems Modeling*, 2010.

[3] P. Borba. An Introduction to Software Product Line Refactoring. In J. a. Fernandes, R. Lämmel, J. Visser, and J. a. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*. Springer Berlin/Heidelberg, 2011.

[4] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. In A. Cavalcanti, D. Deharbe, M.-C. Gaudel, and J. Woodcock, editors, *Theoretical Aspects of Computing (ICTAC 2010)*. Springer Berlin/Heidelberg, 2010.

[5] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Longman, 1999.

[6] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the Gap between Modelling and Java. In M. van den Brand and J. Gray, editors, *Proceedings of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers*, 2010.

[7] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, 2008.

[8] W. Heider. Reactive and Iterative Evolution of Model-based Product Lines. In *Proceedings of the Doctoral Symposium at the 18th IEEE International Requirements Engineering Conference (RE'10)*, 2010.

[9] W. Heider and R. Rabiser. Tool Support for Evolution of Product Lines through Rapid Feedback from Application Engineering. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '11, 2010.

[10] W. Jirapanthong and A. Zisman. XTraQue: Traceability for Product Line Systems. *Software and Systems Modeling*, 2009.

[11] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 1980.

[12] T. Mens, K. Czarnecki, and P. van Gorp. A Taxonomy of Model Transformations. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, 2005.

[13] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering*, GPCE '11, 2011.

[14] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.

[15] I. Porres. Model Refactorings as Rule-based Update Transformations. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Springer Berlin/Heidelberg, 2003.

[16] J. Reimann, M. Seifert, and U. Aßmann. Role-Based Generic Model Refactoring. In D. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*. Springer Berlin/Heidelberg, 2010.

[17] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '11, 2011.

[18] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proceedings of the 6th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, 2012.

[19] C. Seidl. *Evolution in Feature-Oriented Model-Based Software Product Line Engineering*. Diploma Thesis, Technische Universität Dresden, 2011.

[20] F. Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data and Knowledge Engineering*, 2000.

[21] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and Scalable Consistency Checking on Product Line Variability Models. In *Proceedings of the International Conference on Automated Software Engineering*, 2010.