Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines

Alexandre Bragança[1] and Ricardo J. Machado[2]
[1] *Dep. Eng. Informática, ISEP, IPP, Porto, Portugal,*
alex@dei.isep.ipp.pt
[2] *Dep. Sistemas de Informação, Universidade do Minho, Guimarães, Portugal,*
rmac@dsi.uminho.pt

Abstract

Features have been widely used by the product line community to model variability. They represent the common and variable characteristics of the members of a product line. They are very well suited for the configuration of product line members. Outside the product line community, use cases are also widely used to model the functionality of systems at a similar level of abstraction but from a user perspective. Significant work has been done by several authors regarding the possible relationship between these two perspectives of a system. Nonetheless, this has been done in an informal way. In this paper we explore the relationships between these two perspectives and describe a possible approach to automate the transformation from UML use case to feature models.

1. Introduction

Features have been widely used by the product line community to model variability. They represent the common and variable characteristics of the members of a product line. They are very well suited for the configuration of product line members. Outside the product line community, use cases are also widely used to model the functionality of systems at a similar level of abstraction but from a user perspective. In some approaches, use case diagrams are also used to model variability. The similarities between these two concepts have also been discussed by the product line community. However, to our knowledge, a formal approach that relates use cases and features in a way that supports automation is still an unexplored topic.

In this paper, we present an approach to formalize the mappings between use cases and features. We do so in the context of a model-driven approach and present a possible implementation roadmap based on open source modeling and transformation tools. The activity diagram presented in Figure 1 depicts our approach.

The remainder of this paper is structured as follows. In Section 2, we briefly discuss feature diagrams and present the feature metamodel. In Section 3, we discuss UML use cases and present our proposed extensions to the use case metamodel. Section 4 is dedicated to present the mappings between use case and feature model concepts. In Section 5, we present a possible implementation for the approach. Section 6 is dedicated to the discussion of related work and to concluding remarks.

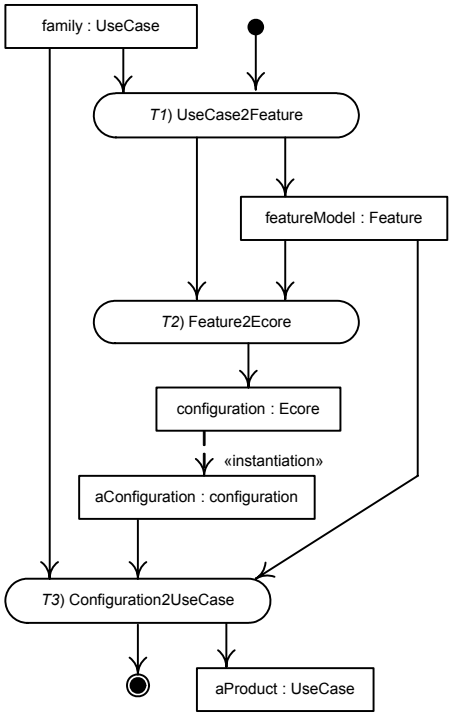

Fig. 1. Process for obtaining a product use case model from a family use case model.

0-7695-2888-0/07 $25.00 © 2007 IEEE
DOI 10.1109/SPLINE.2007.17

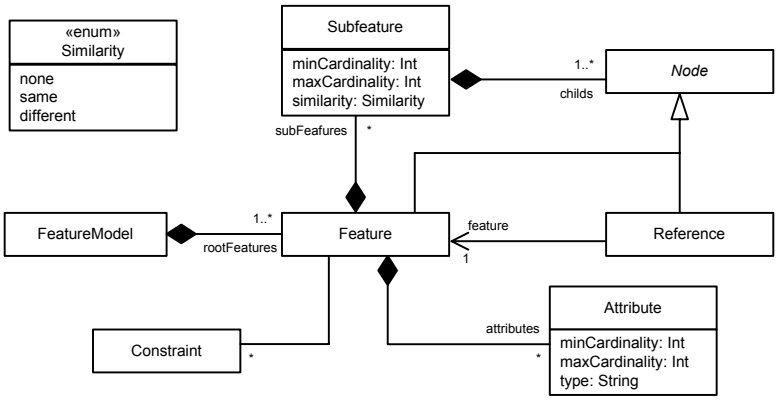

Fig. 2. Feature metamodel.

2. Feature Models

Feature modeling is widely used to model commonality and variability in software systems, particularly software product lines. Although this is true, implementations vary significantly and there is no common globally accepted metamodel for features. In this section we present our approach to feature modeling that is partially compliant on [1] and [2].

Figure 2 presents our metamodel for feature diagrams. We adopt the notion that a feature represents a characteristic or property of a system that is relevant to some user or stakeholder. From this perspective, a feature diagram represents the properties or characteristics that a system may have. Since features can be further characterized by subfeatures, a feature diagram is usually represented as a tree of features with adornments that visually represent relationships between the features. The feature at the root of the tree is called *root feature* and it is usually a conceptual feature that represents the whole system. Because features represent characteristics that may (or not) be present in a system, feature diagrams are well suited to represent common (for features that are always present) and variable (for features that may not be present) characteristics of a system. The process of removing the variability out of the feature diagram (by selecting –or not- optional features) results in the configuration of a system (*feature configuration*).

Basically, a feature is said to be *mandatory* if it is included in all configurations. A feature that may not be present in all configurations is called *optional*. *Alternative* features are features that form a group from which they are selected according to some rule (usually the rule states that only one feature of the group can be selected).

Our metamodel for feature diagrams supports all the presented concepts. We use *Subfeature* to represent containment relationships between features. A mandatory feature is a child (*Subfeature*) of some other feature for which the *minCardinality* and *maxCardinality* are 1. An optional feature is a child of some other feature for which the *minCardinality* is 0 and the *maxCardinality* is 1. A feature group can be modeled by a *SubFeature* with the alternative features as *childs* and the specific cardinality of the group stated by *minCardinality* and *maxCardinality*. The *similarity* enumeration is used to state if the selected alternatives must by of the same kind[1]. Since a feature can only be contained be another feature, we use the concept of *Reference* to enable a feature to be referred by several *Subfeature* relations.

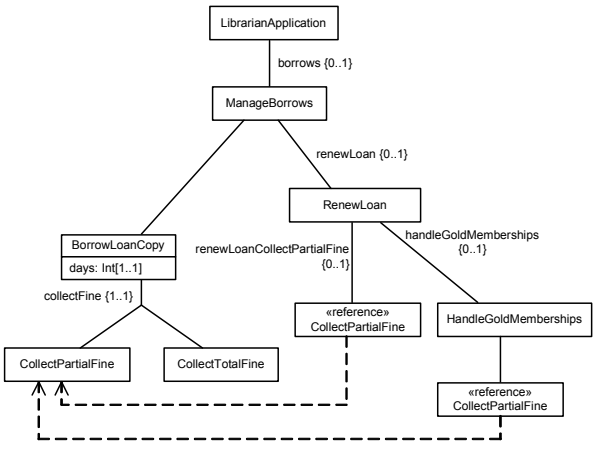

Fig. 3. Excerpt of a Library feature model.

[1] The *similarity* concept was proposed by [1].

Figure 3 presents an excerpt of a feature model for a family of Library applications. *Features* are represented within rectangles, in a way that is similar to UML classifiers [3]. *Subfeatures* are represented as links between two or more *Features*. These links can be adorned with the values for the attributes of *Subfeatures*. For instance, *collectFine* is a Subfeature that relates the *parent* Feature *BorrowLoanCopy* with the *childs CollectPartialFine* and *CollectTotalFine*. In this case, *minCardinality* and *maxCardinality* are both 1, which means that when configuring a member of the family we must select only one of the *Subfeatures* of *BorrowLoanCopy*. Figure 3 also presents examples of the use of *References*. For instance, a *Reference* is used to state that the feature *RenewLoan* can also have *CollectPartialFine* as a *Subfeature*. *Attributes* can be used to further characterize features. In this example, the feature *BorrowLoanCopy* can be further characterized by the attribute *days* that represent the maximum number of days that a library member can borrow a book.

Finally, it is possible to model dependencies between features using *constraints*. A constraint language, such as OCL, can be used to express these dependencies [4]. For instance, the following is an example of an OCL constraint that disables the subfeature *renewLoanCollectPartialFine* if the feature *CollectTotalFine* is selected:

```
context ManageBorrows inv:
self.borrowLoanCopy.collectFine
  ->oclIsTypeOf(CollectTotalFine) implies
  self.renewLoan.renewLoanCollectPartialFine
    ->isEmpty();
```

A feature model, such as the one presented in Figure 3, represents all the possible features for applications of a family of applications. We *configure* a specific application of the family, by removing all the variability from the feature model. If we follow the analogy that a feature is similar to a classifier, then a *configuration* is achieved by a *valid* instantiation of the features (classifiers). We will elaborate such approach in section 4, when we describe a possible implementation roadmap.

In the next section, we present and discuss the use case metamodel and in section 4 we discuss how use cases can be used as a source for feature modeling.

3. Use Cases

Use cases have been widely adopted since its introduction [5]. They have become an integral part of the UML standard modeling language. Use cases are used essentially for functional requirements modeling, as a source for the initial design of a system and for documentation. However, with the recent model-driven approaches, such as MDA [6], and the appearance of supporting tools, using computational independent models – such as use cases – as first class development artifacts can become a reality. However, to achieve this goal with use cases, it is necessary to remove all ambiguities existent in the UML use case metamodel, specially for modeling variability [7, 8]. In this section we present our approach to achieve that goal. Figure 4 presents an excerpt of the UML 2.0 metamodel that is related to use cases. Our main extensions to the original metamodel are depicted in gray. Next, we explain these extensions.

According to the UML 2.0 specification, a use case is the "specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system". As such, these set of actions represent behavior of a system. As it is possible to observe from Figure 4, a *UseCase* has one *mainBehavior* and can have several *alternativeBehavior*'s. The UML 2.0 specification does not state how the behavior of use cases should be specified but, since our approach needs a formal specification, we will use *activities*. So, each behavior of a use case is specified by an *Activity*.

Use cases can have relationships between them. Basically, a use case can *include* (or be included by) other use cases and can *extend* (or be extended by) other use cases.

The *include* relationship acts like a procedure call, i.e., at some specific point of a use case the behavior of another use case is executed. We have introduced the *InclusionPoint* element that represents the point in the use case where the inclusion occurs (see Figure 4). The *location* attribute is a reference to a node of an activity that models one of the behaviors of the use case.

On the other end, the *extend* relationship acts like a deviation of the normal flow of a use case. This deviation is usually conditional, so the base behavior is unaware of the extension. We formalize the UML original notion of extension fragment (*ExtensionFragment* element) and add the notion of rejoin (*Rejoin* element). As such, if the extend condition is true, the use case behavior is extended at one or more extension points, by the corresponding fragments (which are alternative behaviors of the extending use case). The attribute *location* of the *ExtensionPoint* element is a reference to a node of an activity that models one of the behaviors of the extended use case.

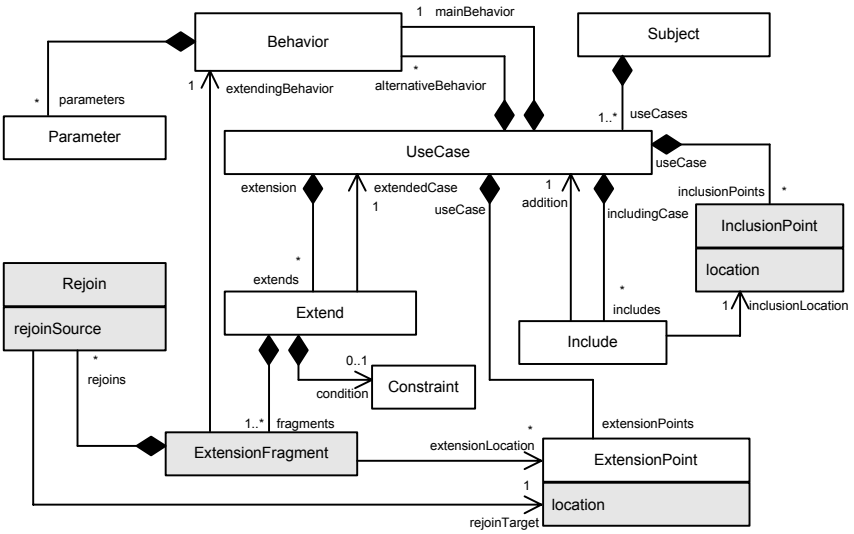

Fig. 4. Excerpt of Use Case metamodel.

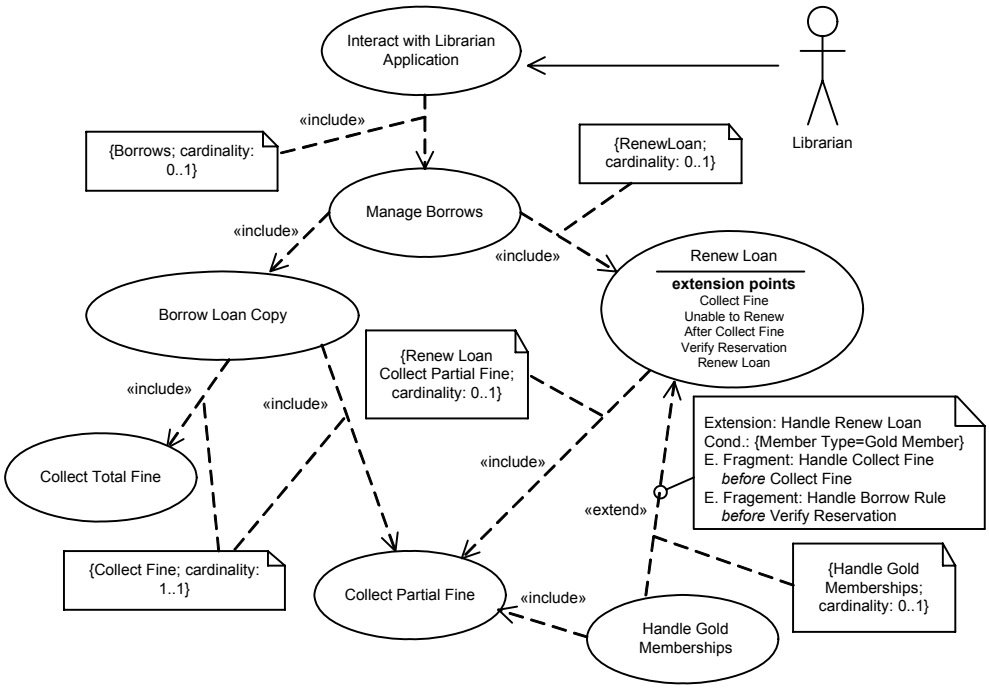

Fig. 5. Example of a use case diagram for a Library product line.

The *rejoinSource* of the *Rejoin* element is a reference to a node of an activity that models the alternative behavior of the fragment of the extending use case. A more deep discussion of the extend relationship can be found in [9].

With these extensions to the original UML use case metamodel we remove the existing limitations that restricted its application into model-driven approaches. Of course, further specifications can be added, notably constraints to validate the model, such as the ones presented next:

```
context Include
inv: self.inclusionLocation.useCase =
       self.includingCase;
inv: self.addition <> self.includingCase;

context Extend
inv: self.extension <> self.extendedCase;
inv: self.fragments->forAll( f |
       f.extensionLocation.useCase =
       self.extendedCase );
inv: self.fragments->forAll( f |
       f.rejoins->forAll( r |
       r.rejoinTarget.useCase =
       self.extendedCase ) );
```

In this section we have presented an approach to remove ambiguities from the UML use case metamodel. It is now possible to analyze the semantic and syntactic relations between use cases and features. In the next section we present our view on this topic.

4. Relating Use Cases and Features

The issue of relating use cases and features is not new. Notably, there is the much referenced work of Griss, Favaro and d'Alessandro [10]. In their work they propose an approach by which functional features are extracted from the domain use case model. They also propose that the structure of the feature model can be created according to the structure of the use case model (by using the «include» and «extend» relationships). As the authors suggest, further types of features can be discovered and added along the development process, such as features resulting from architectural or design modeling tasks. More recent works in this field are also aligned with this approach [7, 11, 12]. We also follow this approach, since feature modeling requires an extensive knowledge of the domain, which is only possible after the effective modeling of such a domain. This is true, particularly for the functional features of the domain. So, the initial feature model is build from the domain use case model. In the remainder of this section we will discuss this mapping based on the use case model example of Figure 5. Figure 5 already contains visual annotations that are used to model variability. For the moment we will disregard these annotations. As the figure shows, a Library system has functionality that regards to the Librarian. The Librarian can use it to manage borrows. He can borrow loan copies to library users and also renew loans. Such borrows can be subject to fines if they surpass a certain duration. In the case the user is a *gold member* of the library, special treatment applies.

Use Cases

According to our approach, each use case is mapped to a feature. Top use cases become root features (see feature metamodel in Figure 2). The complete structure of the feature model can only be created by examining the relations between use cases. As such, we cannot say a use case is mandatory or optional without a context. This context results from the relationships the use case has with other use cases. For instance, if the functionality of a use case is always *referenced* by other use cases, then we can say that such a use case is mandatory. This is the case for the top level use case *Interact with Librarian Application*.

Next, we examine each of the use case types of relationships and elaborate on how they can be used to model variability and how they can be mapped to the feature model.

Include Relationship

In the UML standard documentation there is nothing to support that the include relationship can be used for modeling variability. The documentation states that "The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case". However, in the context of a variability intensive system, like a product line, it is common to have alternative or optional includes [11]. In the example of Figure 5, we have two alternative includes from the use case *Borrow Loan Copy*: one includes the use case *Collect Total Fine* and the other includes the use case *Collect Partial Fine*. In this case, if the two included use cases match the requirements for the inclusion point, no harm is done, since one of them will supply the expected behavior to the including use case.

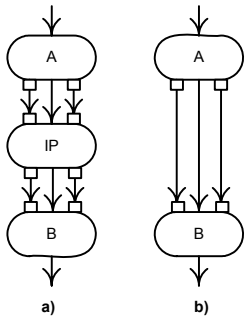

Fig. 6. Removing a node from an activity diagram.

In the case we need to model only one include as optional, some extra care is needed. If such include does not take place, the result is the suppression of the inclusion point on the including use case. Regarding the behavior of the use case, this results in the removal of the activity node corresponding to the inclusion point. If the node had only one incoming and one outgoing control flow, we simply connect the outgoing

flow of the previous node with the incoming flow of the next node. We must also make sure that the output object nodes of the previous node are connect to the input object nodes of the next node and that they are compatible. This situation is presented in Figure 6, where an inclusion point node (a) is removed (b). If such requirements are meet, then it is also safe to have optional includes. More complex scenarios may also be possible but require human intervention or more contextual knowledge from the modeling tool.

Since there is no previous notation to model variability in use cases, we introduce the concept of *variability annotation*. In Figure 5, they are represented as notes linked to the *include* and *extend* relationships. They represent variability points with a name, a minimum and a maximum cardinality and the respective options. For instance, the variability annotation *Collect Fine*, has a cardinality `1..1` that says that one and only one of the options must be selected. The two options are the includes that related the use case *Borrow Loan Copy* to the included use cases *Collect Total Fine* and *Collect Partial Fine*. Since it is the modeler of the use case domain model that is editing these use cases and relationships, he/she is also capable of making these annotations.

With this variability information annotated in the use case model it becomes possible to map the use case relationships to the feature model. In the case of the *include* relationship, each include annotation is mapped into a *Subfeature*. The including use case is mapped to the parent feature of the *Subfeature* and the included use cases are mapped to the *childs* of the *Subfeature* (see Figure 2). Since a use case can be referenced by more than one *include*/*extend* it can also become a *child* in several *subfeatures*. Because a feature definition exists only once, a use case is mapped only once to a *Feature* and the subsequent references are mapped to a *Reference* in the feature model.

Extend Relationship

Contrasting with the include relationship, the extend relationship is used to model variability. As we can observe from Figure 4, an *extend* has an associated condition. If this condition evaluates to true, the use case is extended by the extension fragment's behaviors. On the other end, if the condition is not true, no extension is performed, and the behavior of the base use case remains unchanged and unaware of the extending use case.

In the example of Figure 5, the use case *Renew Loan* can be extended by the use case *Handle Gold Memberships*. As the extend note states, the extension only takes place if the member that is renewing the loan is a gold member. As the example shows, these conditions typically relate to alternative or extending behavior at an application level, not at a product line level. As such, and also not to alter the semantics of the extend relationship, we also use variability annotations to mark the extend relationship. In Figure 5, the annotation *Handle Gold Memberships* states that the corresponding extend relationship is optional. The possibility of also annotating the extend relationships with variability annotations permits, for instance, the modeling of groups of alternative extends.

5. Implementation Roadmap

In this section we present a possible implementation roadmap to the approach described in the paper. For that we use Eclipse Modeling Framework (EMF) version 2.2.0 [13] and SmartQVT version 0.1.3 [14]. The EMF provides a modeling and code generation framework for Eclipse applications based on *Ecore* models. These Ecore models support Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [15]. We note that the code presented is in compliance with such versions and may not be valid in other versions of the tools. For the validation of the Ecore models a possible approach is to use an implementation similar to the one described in [16].

The process of mapping use cases to features is the one presented in Figure 1. The main goal of the process is to obtain a use case model for a specific application of a domain based on a feature configuration model. For that, we use the approach to map use cases to features as discussed in the previous sections. Basically, it consists of three transformations: transform a family use case model into a feature model (*T1*); transform a feature model into a configuration metamodel (Ecore model) (*T2*); and finally, transform a configuration model and a family use case model into an application use case model (*T3*).

T1: Family Use Case Model to Feature Model

The family use case metamodel is similar to the one presented in Figure 4 with the addition of two new elements used to annotate variability: *ExtendVariability* and *IncludeVariability* (see Figure 7). These enable the annotation of variability into *extend* and *include* relationships, as described in the previous section. Figure 5 presents an example of these annotations in a family use case model. The resulting feature model must be in conformance with the feature metamodel presented in Figure 2. Figure 8 presents an extract of the QVT operational transformation that map a use case family model into a feature model.

Basically, the program starts by mapping each use case to a feature (line 6). Features resulting from use cases have the name of the corresponding use case. The program then maps each *include* and *extend*, that are not referenced by variability annotations, into *Subfeatures* (lines 7 and 8). For that, it verifies if the *Feature* that maps to the included use case is already member of a *Subfeature* (lines 34 and 35). If so, it uses a *Reference* to reference that *Feature*. If not, the *Feature* becomes a direct child of the *Subfeature*. Obviously, these *Subfeatures* are mandatory (line 37).

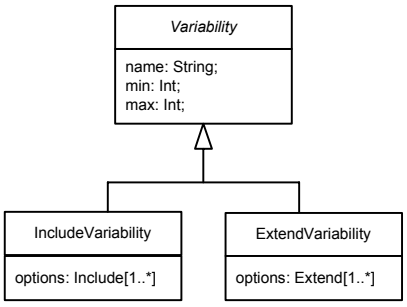

Fig. 7. Variability *annotations* for use case models.

After mapping non-annotated *include* and *extend* relationships, the transformation program maps *IncludeVariability* and *ExtendVariability* elements to *Subfeatures* (lines 9 and 10). The transformation involves mapping each of the *options* of the *IncludeVariability* (or the *ExtendVariability*) into a *Subfeature* child (lines 55 to 61). These *options* will become *Subfeature* child's of type *Reference* or *Feature*, according to the previously described logic. The information regarding cardinality of the variability option groups is mapped directly to the cardinality of the respective *Subfeature* (line 64 and 65).
With this transformation program, if we take as input the family use case model presented in Figure 5, we obtain a feature model similar to the one presented in Figure 3.

T2: Feature Model to Configuration Metamodel
With the previous transformation we obtain a feature model that is in conformance with the feature metamodel of Figure 2. What we would like to do now is to build configuration feature models that are in conformance with the feature model that resulted from the previous transformation, i.e., the feature model should become the metamodel for the configuration models. To achieve this, we use an approach in which a model is *promoted* to a metamodel. In this case, the feature model that resulted from the previous transformation is transformed into an Ecore metamodel. If we map *Features* to *Classes*, then *Subfeatures* become naturally associations between *Classes*. With this approach, feature configurations are simply instances of the corresponding *Classes*. This is similar to the approach proposed in [1]. Although it is a recent discussion topic, at least among practitioners, the generic process of promoting models to metamodels is out of scope of this paper [17]. Next, we briefly describe the transformation between feature models and Ecore feature configuration metamodels. An extract of the QVT operational transformation is presented in Figure 9. In the transformation, each *Subfeature* is mapped to an abstract *EClass* with the same name (line 15). Later, this abstract *EClass* will become the *eSuperType* of the types that will map to the *childs* of the *Subfeature* (lines 50 to 53). After transforming all *Subfeatures* to abstract *EClasses*, the program maps each *rootFeature* to a non-abstract *EClass* (line 16) and to an *EReference* from the *EClass* resulting from the *FeatureModel* element to each of the new *EClasses* (line 17). This will map all top level *Features*. Next, all not yet mapped *Features* are also mapped to non-abstract *EClasses* (lines 19 and 20). In a second pass, all *Features* are again processed (line 22). This time, for each *Feature*, its *Subfeatures* are mapped to *EReferences*. The *eType* for each of these *EReferences* is the abstract *EClass* that resulted from the initial transformation of the *SubFeatures* (line 47 and 48). This abstract *EClass* becomes the *eSuperType* of the *EClasses* that were mapped from the *childs* (*References* or *Features*) of the *Subfeature* (lines 50 to 53). With this transformation we obtain an Ecore metamodel that is equivalent to the feature model. This transformation regards the activitity *T2* depicted in Figure 1. We can now use the EMF generation capabilities to generate an editor from which we can create feature configurations that are in conformance with the metamodel.

T3: Family to Application Use Case Model
The last step in the transformation process involves the generation of application use case models from feature configurations (activity *T3* in Figure 1). This requires a transformation that must have at least the family use case model and the feature configuration model as input and a product (or application) use case model as output. Basically, the transformation involves including in the output model only the use cases, includes and extends relationships that are referenced by the feature configuration model. This transformation can be similar to the other two described in the paper.

```
transformation Usecases2Features(in ucModel:USECASE, out fModel:FEATURE);

main() {
  ucModel.objects()[Subject]->map subject_to_feature_model();

  ucModel.objects()[UseCase]->map usecase to feature();
  ucModel.objects()[Include]->map include to subfeature();
  ucModel.objects()[Extend]->map extend_to_subfeature();
  ucModel.objects()[IncludeVariability]->map includeVariability_to_subfeature();
  ucModel.objects()[ExtendVariability]->map extendVariability_to_subfeature();
}

mapping Subject::subject_to_feature_model () : FeatureModel {
  rootFeature := ucModel.objects()[UseCase]->map usecase to root feature();
  name := self.name;
}

mapping UseCase::usecase_to_root_feature () ...

mapping UseCase::usecase_to_feature () : Feature ...

helper includeInVariability(i: Include) : Boolean {
  var x :=ucModel.objects()[IncludeVariability]->select(iv | iv.options->exists(i1|i1=i) );
  var y := x->first();
  return if y = null then false else true endif;
}

mapping Include::include_to_subfeature () : Subfeature
  when { not includeInVariability( self ); } {
  var f: Feature; var r: Reference;

  parent := self.includingCase.resolveone(Feature);
  f := self.addition.resolveone(Feature);
  r := if repeatedFeature(f) then object Reference{ name:=f.name; feature:=f; }
       else null endif;

  name := self.name;
  minCardinality:=1; maxCardinality:=1;
  childs := if repeatedFeature(f) then Sequence { r.asType(Node) }
            else Sequence { f.asType(Node) } endif;
}

helper repeatedFeature(f: Feature) : Boolean {
  var x := fModel.objects()[Subfeature]->select(sf | sf.childs->exists( f1| f1=f) );
  var y := x->first();
  return if y = null then false else true endif;
}

mapping IncludeVariability::includeVariability to subfeature () : Subfeature {
  var f: Feature; var r: Reference;

  parent := self.options->first().includingCase.resolveone(Feature);

  childs := Sequence { };
  self.options->forEach(i) {
    f := i.addition.resolveone(Feature);
    r := if repeatedFeature(f) then object Reference{ name:=f.name; feature:=f; }
         else null endif;
    childs += if repeatedFeature(f) then Sequence { r.asType(Node) }
              else Sequence { f.asType(Node) } endif;
  };

  name := self.name;
  minCardinality := self.min;
  maxCardinality := self.max;
}
```

Fig. 8. Extract of QVT Operational transformation from use case to feature model.

```
transformation Features2Ecore(in fModel:FEATURE, out eModel:Ecore);

main() {
  fModel.objects()[FeatureModel]->map feature_model_to_epackage();
}

mapping FeatureModel::feature model to epackage () : EPackage {
  var fm: EClass;
  name := self.name;

  -- first pass
  fm := self->map feature_model_to_eclass();
  eClassifiers := Sequence { fm };
  -- Each Subfeature becomes an abstract class
  eClassifiers += fModel.objects()[Subfeature]->map subfeature to eclass();
  eClassifiers += self.rootFeature[Feature]->map feature to eclass();
  fm.eStructuralFeatures := self.rootFeature[Feature]->map rootfeature_to_ereference(fm);
  -- Transform all other features...
  eClassifiers += fModel.objects()[Feature]->select(x | x.resolveone(EClass)=null)
                    ->map feature_to_eclass();
  -- second pass
  fModel.objects()[Feature]->map subfeatures();
}

mapping FeatureModel::feature_model_to_eclass () : EClass ...

mapping Feature::rootfeature_to_ereference ( fm: EClass) : EReference ...

mapping Feature::feature_to_eclass () : EClass ...

mapping inout Feature::subfeatures () {
  var c:EClass;
  c := self.resolveone(EClass);
  c.eStructuralFeatures := self.subFeatures[Subfeature]->map subfeature_to_ereference();
}

mapping Subfeature::subfeature_to_eclass () : EClass ...

mapping Subfeature::subfeature to ereference () : EReference {
  var c: EClass;

  name := self.name;
  containment := true;
  lowerBound := self.minCardinality;
  upperBound := self.maxCardinality;

  c := self.resolveone(EClass);
  eType := c;

  self.childs[Feature]->select(c|c.oclIsKindOf(Feature)).resolveone(EClass)
                          ->map childs_to_subtype(c);
  self.childs[Reference]->select(c|c.oclIsKindOf(Reference)).feature.resolveone(EClass)
                            ->map childs_to_subtype(c);
}

mapping inout EClass::childs_to_subtype (superType: EClass) ...
```

Fig. 9. Extract of QVT Operational transformation from feature to Ecore model.

A more generic transformation program could be required if we wanted our transformation to support changes in the family use case model (which can by very probable). Such changes result also in changes to the feature model that acts as metamodel for the feature configurations. To have only one *T3* transformation process regardless of number of configuration metamodels, the *T3* activity transformation could also have as input the feature model. Since this model is the source for obtaining the configuration metamodels, it could serve as guide to process the feature configuration models regardless of their metamodels (they would have to be processed as generic Ecore models), thus allowing a generic

transformation process that can be applied to all possible feature models.

6. Conclusions

In this paper, we have presented a model-driven approach to map use case to features. This approach is inspired by the original work of *Griss et al.* [10]. Gomaa also proposes a similar relationship between use cases and features [11]. We differ from their works because we base the variability annotations in the *extend* and *include* relationships and not in the use cases. We have explained in the paper why this is more appropriate to model variability.

Czarnecki *et al.* presented an approach to map features to design models [18]. Basically, in their approach, a template design model is annotated with presence conditions that are logic expressions based on features. Non-annotated elements have an implicit true presence condition. If these expressions evaluate to true, the design element is included in the result model. In their approach, design elements must be annotated after the feature model. In our approach, the use case variability annotations have a similar effect, but we differ, since the design elements included in a configuration will result from the ones that are necessary to realize the product use case model that results from the transformation process. Eriksson *et al.* also describe a model based approach that relates use cases and features [7]. However, their work is focused on used cases being described by scenarios and sequences of steps. As such, it does not explicitly deal with mappings at a UML use case diagram level (as we do) but at an inner use case level. They document the transformation process but do not precisely specify it, so it is difficult to tell if such approach is possible to implement with transformation languages such as QVT. Their approach to modeling use case variability has similarities with the one of Fantechi *et al.* [19]. Although we have not explore this topic in the paper, our approach to model variability within use cases is by doing so with the activities that model the use case's behavior. Similar variability annotations as the ones present for the *extend* and *include* relationships can be used to annotate activity model elements and a similar transformation approach can also be used to map such elements to feature model elements.

We have discussed and proposed mappings between use case and feature models in a formal way that supports its implementation. Regarding applicability of our approach, we have showed that an implementation is feasible even with a not yet mature QVT implementation. Clearly, transformations could be improved by a complete QVT implementation enabling, for instance, the use of the QVT relational language. As discussed, the proposed method seams to open a feasible approach to implement mixed use case and feature model-driven based product line engineering methods.

7. References

[1]T. Asikainen, T. Mannisto, and T. Soininen, "A Unified Conceptual Foundation for Feature Modeling," presented at SPLC2006, Baltimore, 2006.

[2]K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," presented at SPLC2004, Boston, 2004.

[3]OMG, "Unified Modeling Language Version 2.0: Superstructure (formal/05-07-04)," OMG, 2005, Available at http://www.omg.org.

[4]OMG, "Object Constraint Language Specification v2.0 Final Adopted Specification (formal/06-05-01)," OMG, 2006, Available at http://www.omg.org.

[5]I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[6]OMG, "Model-Driven Architecture Guide Version 1.0.1," OMG, 2003, Available at http://www.omg.org.

[7]M. Eriksson, J. Borstler, and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations," presented at SPLC2005, Rennes, France, 2005.

[8]T. v. d. Maßen and H. Lichter, "Modeling Variability by UML Use Case Diagrams," presented at REPL'02, Essen, Germany, 2002.

[9]A. Braganca and R. J. Machado, "Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification," presented at SPLC 2006, Baltimore, Maryland, 2006.

[10]M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," presented at Fifth International Conference on Software Reuse, Victoria, Canada, 1998.

[11]H. Gomaa, *Designing Software Product Lines with UML*: Addison Wesley, 2005.

[12]I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*: Addison Wesley, 2005.

[13]E. Foundation, "Eclipse Modeling Framework (EMF)," Eclipse Foundation, 2007, http://www.eclipse.org/modeling/emf/.

[14]F. T. R&D, "SmartQVT - Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language," 2007, Available at http://smartqvt.elibel.tm.fr/.

[15]OMG, "Meta Object Facility (MOF) 2.0 Core Specification (formal/06-01-01)," OMG, 2006, Available at http://www.omg.org.

[16]C. W. Damus, "Implementing Model Integrity in EMF with EMFT OCL," 2006, http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html.

[17]E. Merks and *Others*, "Making EMF models valid Ecore models for a two-level code generation," eclipse.tools.emf newsgroup thread, 2006, http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg20713.html.

[18]K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," presented at GPCE'05, Tallinn, Estonia, 2005.

[19]A. Fantechi, S. Gnesi, G. Lami, and E. Nesti, "A Methodology for the Derivation and Verification of Use Cases for Product Lines," presented at SPLC2004, Boston, 2004.