Automated Feature Model-based Generation of Refinement Transformations

Lucia Kapova  
Institute for Program Structures and Data Organization  
Universität Karlsruhe(TH)  
Karlsruhe, Germany  
kapova@ipd.uka.de

Thomas Goldschmidt  
FZI Research Center for Information Technology  
Karlsruhe, Germany  
goldschmidt@fzi.de

Abstract

Model-driven application engineering builds on the concept of model transformations. To weave additional refinement parts into an application model, so-called refinement transformations are used. In many cases these refinement parts are highly variable and configurable. Such a configuration could depend on application specific features. Today, application developers need to define refinement transformations manually, including all possible configuration combinations. Due to the high number of possible initial requirements such a development method is costly and means significant effort. Therefore configurable refinements should be embedded in an overall model-driven application development process. Currently there is a lack of automated support for integrating these configuration decisions into the development process of refinement transformations. In this paper, we introduce a novel approach for automated feature model-based generation of refinement transformations. To express application features, we consider configurations specified by extended feature diagrams. In addition, we provide a running example giving insight into the application of the presented approach.

1. Introduction

In many domains, requirements regarding the final software product are constantly evolving. Development decisions that are based on these requirements are a foundation for the creation of software application variations. Creation of application variations introduces demands for highly efficient and low-complexity reconfiguration methods. In most of the cases software applications include variable and configurable parts, called refinements. Such application refinements should be embedded in an overall model-driven application development process to decrease costs and development effort. Model transformations are a central concept of model-driven engineering. Applying the refinements on an abstract level using model transformations to create refined application models on a lower-level of abstraction is an adequate instrument.

Nowadays, the most frequent way to configure model transformations is by means of external annotations to a source model (also called mark model [1][2]). Mark models are used to provide configuration details that are specific to the source model. This configuration model is considered as an additional input for the transformation on the model level. However, this way of transformation configuration is not always preferable. There are cases where this configuration mostly depends on externally defined properties and is not specific to special model elements. In this case the configuration happens on the higher level of abstraction. Therefore, it is much more appropriate to decouple the configuration and the actual model. Additionally, there is a need to define the configuration as reusable construct. For example, the configuration could be read by other development tools that build on that application model. Therefore, the configuration should be specified on a more generic metamodel-independent level.

We use feature models to express application configurations. The refinement transformation can then be based on the very same feature model to apply the appropriate changes in the application model according to currently selected features. Such a configuration is reusable, focused and better readable. Writing a transformation that considers all possible combinations of selected features is very tedious and error-prone. In such transformations certain fragments of the transformation are activated depending on the activation of a certain feature in the configuration. To relieve from the substantial development effort that arises from these facts, we map refinement transformation fragments on features within the configuration model. Based on the selected combination of features, a refinement transformation is automatically generated. This way we influence directly the transformation itself, allowing the association between configuration and the transformation to be more visible.

In the proposed approach we lift the configuration model to a higher abstraction level. Therefore, the transformation fragments do not get polluted with code that is only responsible for checking the actual feature configuration. Furthermore, as the binding of fragments and features is more explicit this alleviates the complexity of transformation evolution.
The Object Management Group (OMG) introduced the specification of Query/View/Transformation (QVT)[3] as a standard for model-to-model transformations. One part of this specification is the QVT Relations. As this language is declarative, transformations written in this language can be modularized very easily. Therefore, we use this language to specify the refinement fragments.

Although proposed approach has a wide range of application domains, we further investigated opportunities in the component-based applications domain. In this domain we analysed different application scenarios. We have identified following reconfigurable constructs, for example, Remote Procedure Call Connector, Message Oriented Middleware, ThreadPool Usage, Component or Resource Replication and Asynchronous Communication refinements. Depending on the purpose of the component-based application model (for example, for further analysis or performance predictions) such detail as configuration of infrastructure services (for example, used middleware or threadpool) could influence the expressiveness of the model. To provide realistic analysis or predictions of the application behaviour we need to raise these low-level details to the higher-abstraction level of the modelled application.

To illustrate the application of the presented approach we used a running example based on the configuration of threadpool middleware implementation properties. The used scenario consists of a simple distributed architecture where a service provider is deployed on an application server. The application server provides infrastructure services to deployed components. One of these services is the coordination of parallel access to resources based on a threadpool concept. In the modelled application deployed on a such an application server this information is not included but needed for accurate further analysis [4].

The contributions of this paper are (I) a feature-model based configuration process that allows easy variation of application configurations, (II) guidelines for specifying transformation fragments based on the feature model and (III) higher-order transformations for generation of refinement transformations. The main advantage is provided by performing the model transformation configuration automatically based on features instead of models. This separation of concerns can achieve high variability and flexibility in the development of software applications.

The paper is structured as follows. An overview on the process for defining and integrating the feature model based configuration is presented in section 2. The following 3 sections describe details of this process: Section 3 explains how the creation of a feature model is done. Guidelines on how to specify the feature based transformation fragments are given in section 4. The configuration and refinement process is described in section 5. Section 6 performs a discussion of the advantages and drawbacks of our approach. Related work is discussed in section 7 and the paper is finally concluded in section 8.

2. Feature Model-based Configuration Process

Generally, the presented software development process is very similar to those with the common goal of reusability and customizability. Our process is focused on reuse of process artifacts, especially feature models and their configurations. The goal of the process is to automatically generate a refinement transformation based on these configurations. The overview of this process is illustrated in Figure 1 with our contribution pointed up by star symbol. The stripes illustrate automated task execution.

In the presented development process (based on [2]) we distinguish two phases, the first is domain engineering and the second is application engineering. In the following we describe the tasks included in these phases.

In the domain engineering phases the reusable and reconfigurable refinement constructs are specified. The most important task is domain analysis consisting of the extraction and analysis of possible features and configurations of the refinement. Typically at the beginning of development there is only an abstract idea about these requirements. Towards later development phases, these incomplete, variable and contradictory requirements could change. The domain analysis task has the main goal to recognize and analyse possible requirements on the product and define allowed combinations of them. This analysis defines the first step attempting to design a new reconfigurable construct that could be used in application design. This helps to reduce the risk of a complete redesign of application models based on major requirements changes.

Once the possible requirements are determined they should be analysed and clearly stated. For this purpose, the feature model is used where the requirements are specified

![Figure 1. Model Refinement Process](image-url)
as configurations of properties belonging to a defined reconfigurable construct.

The next step, the refinement design defines how features and their combinations affect the application model. Here it is necessary to determine the dependency between different configuration properties, the model structure and the elements’ attribute values. The result of the refinement design step is an extension of the pre-defined feature model by dependencies and documentation how the features map to the application model changes, called feature effects.

Step Transformation Fragments Development represents the activity of developing actual transformation fragments that encode the feature effects to the application model. The result of this activity is a feature model that is extended by refinement annotations in the form of model-to-model transformation fragments (see section 4 for a detailed explanation of this step). Additionally, the line Reuse Boundary illustrates the end of reusable construct creation process and separates the process of reusable construct usage and reconfiguration.

The phases of the application engineering include actual application model development and requirements analysis. The task configuration benefits of reusable constructs defined by domain engineering and attaches the actual configuration to them. The main goal of this step is making sure the software application will meet the requirements defined for the product, as well as ensuring that future requirements can be addressed.

The most important step that is included in the development process is the automated transformation generation. The generated transformation is then applied (transformation execution) to the input application model resulting in the refined application model.

3. Domain Engineering

In their book on Generative Programming Czarnecki and Eisenecker [2], used so called feature models to capture variabilities of applications. Feature models define all valid combinations of application property values, or features. One feature defines a certain option in the considered domain. Actual combinations of features are called configurations (feature configuration). Feature models are hierarchical decomposition of features including information if a feature is mandatory, alternative or optional.

The metamodel of the used feature diagrams is illustrated in Figure 2. As the metamodel of feature-models is pretty straightforward we build on [2]. To be able to use feature models to configure refinement transformations, we already intended to extend it so, we developed it having an easy extension mechanism in mind.

The extensions to the feature model metamodel make it possible to add transformation rules as annotations to the features. These extensions to the used metamodel are depicted in figure 2. The most important, even quite unintrusive extension was the addition of a reference from the feature to the Relation class the QVT Relations metamodel. This allows to annotate transformation fragments to features. As we also want to allow the specification of variable values through feature configurations we additionally added a reference to the OperationCallExp from the OCL metamodel. This allows features to refer to the “=”-operation from the OCL Standard Library and thus assigning values to variables that are e.g., present in parent features. The third extension was the addition of so called “DisambiguationRules” which is explained in section 4.

Furthermore, the feature model could include feature composition constraints, that indicate which feature combinations are valid and which are not. These constraints can either be hard (depends or excludes constraints) or weak (default values or allowed override). We will refer to these constraints further in section 4.

Using feature models as mark models brings the advantage of having a focused and less-complex configuration method understandable by all of the roles in development process. Such a reconfiguration method can be mapped to transformation parameters and allows generation of refinement transformations. The concept of automated refinement transformation generation is discussed in section 5. In the following section we will illustrate the step of feature model specification on the running example.

Hence, the different feature configurations of the thread-pool implementation should be described as a separate feature model. The current feature model is illustrated in figure 3. The actual feature configuration is illustrated by
check(selected feature) and cross(eliminated feature) -marks.

The ThreadPool Feature Model illustrates different configuration options for a ThreadPool implementation. Each ThreadPool configuration has to include the mandatory feature Optimization Properties. This feature may define either a static or a dynamic threadpool implementation. The exclusive selection is indicated by the excludes constraints between both features. Each of these features have to have a number of threads specified. This is either a static size or, for the dynamic feature, an initial and a maximum number. Additionally, we can specify a ThreadPoolPolicy as PriorityLanes, meaning that a threadpool has a number of thread groups for tasks with different priorities, or Purity, meaning that all tasks have the same priority. When PriorityLanes is selected we have to specify, how the task priorities are defined. They could be either natively or externally specified.

An optional feature of the priority lanes implementation is Thread Borrowing, which defines that groups with higher priority can take a thread from a lower priority group in a case when its threads are all already working and there are idle threads in a lower priority group. Only in this case the Dispatcher could be selected. Dispatcher defines the logic of thread borrowing based on a Semaphore or Condition. Each of the introduced features could have additional information attached as, for example, fragments of code.

Figure 3 additionally depicts one possible configuration of a threadpool on which the next explanation depends. This feature configuration defines a simple static implementation of ThreadPool with the size of 32 threads treating all incoming tasks with the same priority.

4. Transformation Fragments Specification

As presented in section 3, the nodes of the feature model are annotated with transformation fragments. Since it has to be possible to compose those fragments together to a single transformation, there are several constraints on the way the annotated transformation rules are specified. Those constraints result mainly from the structure of the feature model and the patterns which can occur in such a model.

According to the different kinds of relations that can occur between features in a feature model there apply different constraints for the transformation fragments that are annotated to the features. Constraints \( C_1 \) (\( C_1 \) to \( C_5 \)) describe the rules that have to be obeyed when annotating transformation fragments to a specific feature. Furthermore, these constraints serve as basis for the generation of the resulting refinement transformation, which will be explained in section 5. We use the running example from section 3 to explain the different constraints.

\( C_1 \): Relation access for child features: The basic shape of a feature model is that of a tree. Features can have subfeatures forming a parent-child relationship. A child-feature can only be activated if its parent feature is activated. For the scope of the transformation fragments that are attached to the child node this means that the child’s rules may reference those of the parent within its when- and where-clauses.

In the running example this pattern is depicted in figure 4 this pattern occurs between the ThreadPool and the Static feature. The transformation fragment of the Static feature TP_Static has a when-dependency to the transformation fragment TP of its transitive parent ThreadPool.

```
1 — Resulting composed transformation
2 top relation TP_Static {
3  varSize : Integer;
4  checkonly domain in p : Component {};
5  enforce domain out s : TP {
6      size = varSize;
7     when (TP(p, s)) {
8     where { s = 32; }
9    } }
```

Listing 1. Example transformation fragments (\( C_1, C_2 \))

\( C_2 \): Variable assignment for child features: Additionally to the access to when- and where-clauses it is possible for transformation fragments of child rules to control the assignment of free variables of their parents.
See figure 4 and listing 1 for an application of P1 and P2. Feature TP Size can be used to statically configure the size of the threadpool. Hence, the transformation fragment refers to the free variable declared in the TP_Static fragment of feature Static (for the sake of simplicity a simple path notation with the fragment’s name as prefix is used to denote the referred relation). This way the value specified in the feature configuration ($size = 32$) ends up in the assignment within the where-clause of the resulting generated transformation.

$C_3$: Inheritance of mandatory features: Feature models distinguish between mandatory and optional features. As mandatory features are always activated it is possible to reference rules of mandatory features of (transitive) parents within child rules. This means that even siblings can use each other’s rules within their when- and where-clauses if both of them are mandatory within their parent feature.

For example, the fragments of the Dispatcher feature presented in figure 3 can reference fragments of the Optimization Properties feature.

$C_4$: Referencing through constraints: In addition to the parent-child relationship a feature can depend on other features within the feature tree. Such dependencies are modelled as depends-relationships. For the scope of the transformation rules of the dependent feature this results in an import of the rules of the required feature and its scope (that is computed using $C_1$ to $C_3$). All imported rules may then again be used in when- and where-clauses of the current transformation rules.

In the thread pool example (figure 3) this pattern would apply for fragments of the Dispatcher feature referencing relations from the Thread Borrowing feature.

As counterpart to depends, excludes inhibits a concurrent activation of two features. As both features can then never be activated at the same time an interference of their transformation fragments is also impossible.

$C_5$: Disambiguation of inclusive-or: An exclusive-or between subfeatures poses no problem, as they may never occur at the same time and thus their transformation rules can never interfere which each other. A more challenging construct is the inclusive-or relationship. Features connected within such a relationship may occur in an arbitrary combination.

In the thread pool example such different combination possibilities could occur with the Optimization Properties feature: Either ThreadPool Policy, Static or Dynamic, a combination of them (excluding Static or Dynamic selected at the same time, due to the excludes relationship between them) or none of them could be selected. Each possibility results in a different transformation rule in the generated refinement transformation.

$C_{|3|}$: Inheritance of mandatory features: Feature models distinguish between mandatory and optional features. As mandatory features are always activated it is possible to reference rules of mandatory features of (transitive) parents within child rules. This means that even siblings can use each other’s rules within their when- and where-clauses if both of them are mandatory within their parent feature.

For example, the fragments of the Dispatcher feature presented in figure 3 can reference fragments of the Optimization Properties feature.

$C_{|4|}$: Referencing through constraints: In addition to the parent-child relationship a feature can depend on other features within the feature tree. Such dependencies are modelled as depends-relationships. For the scope of the transformation rules of the dependent feature this results in an import of the rules of the required feature and its scope (that is computed using $C_1$ to $C_3$). All imported rules may then again be used in when- and where-clauses of the current transformation rules.

In the thread pool example (figure 3) this pattern would apply for fragments of the Dispatcher feature referencing relations from the Thread Borrowing feature.

As counterpart to depends, excludes inhibits a concurrent activation of two features. As both features can then never be activated at the same time an interference of their transformation fragments is also impossible.

$C_{|5|}$: Disambiguation of inclusive-or: An exclusive-or between subfeatures poses no problem, as they may never occur at the same time and thus their transformation rules can never interfere which each other. A more challenging construct is the inclusive-or relationship. Features connected within such a relationship may occur in an arbitrary combination.

In the thread pool example such different combination possibilities could occur with the Optimization Properties feature: Either ThreadPool Policy, Static or Dynamic, a combination of them (excluding Static or Dynamic selected at the same time, due to the excludes relationship between them) or none of them could be selected. Each possibility results in a different transformation rule in the generated refinement transformation.

Figure 5. Transformation Process

To be able to specify this disambiguation special disambiguation rules were introduced into the feature metamodel (cf. figure 2). The disambiguation is configured by defining one DisambiguationRule for each combination of features that should be treated exceptionally. Within the DisambiguationRule the combination is specified by assigning the features for which the rule applies to the selectedFeatures reference. In theory it would be possible to make a transitive selection of inclusive-or-ed children. However, in the current version of the approach this is not supported. Therefore, a constraint (see listing 2) applies to the selection of features, restricting the possible selection to direct children of the current feature.

Listing 2. Constraint on DisambiguationRule

5. Application Engineering

The model refinement process illustrated on figure 5 depends on a specification of inputs for Higher-Order Transformations.

Higher-Order Transformations are transformations that themselves operate on transformations and are used to generate or transform transformation specifications. Providing higher-order transformations (HOTs) for QVT-Relational can elegantly be done based on their abstract syntax. QVT-Relations with its when- and where-dependency network can by understood quite well when it comes to transformations that create the abstract syntax model. In this case, relations are used to generate or transform the model of other relations. Depending on the QVT engine it is possible to directly execute this transformation model. Alternatively
the model could be pretty printed first in its concrete textual syntax and then loaded again by the transformation engine.

The lack of support for model-driven refinement transformation development methods is a motivation for previous work in [5]. This work proposed general approach for HOTs to generate default copy relations (copy transformation) in QVT Relations [3] for a given metamodel. This way the development of refinement transformations focuses only on the development of refinement relations which modify the source model and there is no need to deal with relations which copy large parts of a model. The most difficult part of this approach are challenges related to support of automatic refinement relation integration and configuration, which was not the focus of the previous work. The contribution of this paper goes beyond the mere action of creating a copy transformation by offering transformation configuration based on a feature model. The basis for a refinement transformation is a generated copy transformation. Parts that are refined by the configured transformation will then replace the standard copy rules for the corresponding metamodel element.

The composition process that follows here is realized using a HOTs, illustrated on a figure 5. It merges the transformation fragments (refinement relations) that are annotated to the feature model nodes together creating the final Refinement Transformation. The first input for this HOT is a Feature Model with mapped Transformation Fragments, these fragments are used by a Higher-Order Transformation for the actual refinement transformation generation. The second input is the actual Feature Configuration, which defines which features are selected and which values of attributes are set. In contrast to an in-place transformation a refinement transformation may also be specified to create a new model where the refinements are applied. For this case the refinement transformation extends a copy transformation that is responsible for creating an exact copy of its input. The Higher-Order Transformation includes the Transformation Fragments into a frame of the generated Copy Transformation. As we rely on QVT Relations for the implementation of our transformations which does not provide native support for copy transformations we use the higher-order transformation based approach from [5] to automatically create this copy transformation from the metamodel of the application model.

For composition of refinement transformation is responsible another Refinement Higher-Order Transformation. This HOT merges holding previously introduced constraints in section sec:fragments Transformation Fragments to map certain requirements of application. The result of this chain of higher-order transformations is Refinement Transformation that transforms Refined Application Model.

```java
6  top relation SelectedFeatureRelation2Relation {
7    n : String;
8  }
9  checkonly domain feat selectedFeature : featureconfig::ConfigNode {
10    configState = featureconfig::ConfigState::SELECTED
11    origin = originFeature : featuremodel::Feature
12    name = n
13    variableAssignments = assignment : OperationCallExp {
14      parentRelation = parentRel : featuremodel::ChildRelation {
15        parent = parentFeature : featuremodel::Feature
16      }
17    }
18  }
19  when { MarkTransformation(transfo); }
20  where { SelectedFeatureRelation(originFeature, targetRel); }
21  CopyRelation(featureRel, targetRel); }
22 }
23 }
24
25  top relation SelectedFeatureVariableAssignment2VariableAssignment {
26    n : String;
27  }
28  checkonly domain feat selectedFeature : featureconfig::ConfigNode {
29    configState = featureconfig::ConfigState::SELECTED
30    origin = originFeature : featuremodel::Feature
31    name = n
32    variableAssignments = assignment : OperationCallExp {
33      parentRelation = parentRel : featuremodel::ChildRelation {
34        parent = parentFeature : featuremodel::Feature
35      }
36    }
37  }
38  when { MarkTransformation(transfo); }
39  where { CopyAssignment(assignment, copiedAssignment); }
40 }
```

**Listing 3. HOT for Fragments Composition.**

For composition of transformation fragments that follow constraints $C_1$ and $C_2$ the HOT that weaves the transformation fragments of the selected features into the final transformations is shown in listing 3. The transformation is based on a generated copy transformation for QVT Relational itself. The copy rules (such as, CopyAssignment or CopyRelation) are used to copy the rules that are specified by the transformation fragments on the selected features. Relation SelectedFeatureRelation2Relation is responsible for matching features that are optional from the feature model and copying the annotated transformation relations to the final transformation. A corresponding relation MandatoryFeatureRelation2Relation is provided to match all mandatory features which do not need to be selected explicitly. Similar HOTs are provided for the weaving process of constraints $C_3$ to $C_5$.

6. Discussion

6.1. Complexity Comparison

To give an indicator of the transformation generation complexity decreased by our approach, we provide an experiment based on comparison between generated and manually written transformation implementation. This illustrates that in transformations is a lot of infrastructure code included. This code could be generated and its implementation could be avoided. Additionally, generated transformations are more structured and therefore better understandable.
The main advantage of our approach is that developers can focus on impact of one selected feature on model at time and develop relations for this feature only, they are not concerned with all the feature combinations and their dependencies. By manual development the developer has to consider all the possible configuration combinations and check the state of features (selected or eliminated) by accessing additional model (feature model) from developed relation of refinement transformation. Even later in development the dependencies (where- and when- clause) between the relations need to be resolved manually. These dependencies are solved in our approach by the transformation generation based on defined constraints. Table 1 gives numbers of generated lines of transformation code in comparison to lines of manually written transformation code. The transformation frame consists of a generated copy transformation, which is used by both manual and automatic way of fragments integration. As shown in this comparison, generated refinement parts of transformation consist of 7 relations in 3 fragments, these fragments could be reused in a case of another feature combination. In a case of manual implementation without reusable fragments, we have to implement a new transformation for each feature combination.

<table>
<thead>
<tr>
<th>Complexity of the model</th>
</tr>
</thead>
<tbody>
<tr>
<td>290</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>1008</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated transformation frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>430</td>
</tr>
<tr>
<td>8550</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated configuration-dependent transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>195</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manually written transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>290</td>
</tr>
</tbody>
</table>

Table 1. Comparison of generation versus manual development for one feature combination

6.2. Discussion of Assumptions and Limitations

Despite the advantages in simplifying the configuration of transformations with our feature model based approach there are also some drawbacks that need to be discussed. One problem arises when the feature configuration is changed and the target model needs to be updated according to the newly woven transformation. The transformation traces that were stored during the last transformation execution will potentially become invalid as the structure of the transformation may have changed significantly. Incremental updates (which are mostly based on the transformation’s trace links) are then impossible. However, this problem only occurs if the transformation engine uses typed traces that are specific to the transformation that created them. Generic trace links pose no problem to the approach.

Another drawback of our approach is the debuggability of the transformation. The debugger of the transformation engine will execute and observe only the generated and woven transformation. Hence, a transformation developer will need to understand the generated transformation in order to be able to debug it. A specialized debugger would be needed if debugging should be possible on the configuration level.

7. Related work

In the domain of model transformation languages model-driven transformation composition and generation, as well as mapping of features and requirements for software applications to transformations, are relatively new. Czarnecki and Antkiewicz in [6] propose a template-based approach for mapping feature models to concise representations of model variabilities. Allowed configuration combinations are dependent on the existence of suitable model templates, that are bases for model instantiation. Our approach is not template-based, where templates and feature models build additional input for transformation. We rather lift the configuration up to the transformation creation itself. This way the complexity of handling all possible feature combinations within the transformations is decreased and is made explicit through the structure of feature model. Other feature oriented programming approaches like AHEAD [7] allow only sequences of independent refinement transformations and do not consider feature dependencies. In the area of product lines works published in [8][9][10] propose a mapping between features and model structure elements. These works proposed support for automated derivation of product line members based on a feature-driven development method. The expressiveness of these methods depends on composability of the mapped structural model fragments.

Dealing with composition of transformations we are heading towards complex problems, that are in the focus of many currently running research initiatives. One of them is [11] which proposes a superimposition composition technique for ATL and QVT Relations. Other works [12] and [13] investigate possibilities of composing complex transformations from atomic transformation definitions. Our approach is different to these composition methods because it is based on a predefined structure (i.e. the feature model) that guides the transformation developer. Furthermore, our focus is on configuration related transformation generation and not generic composition techniques. Therefore, many problems that arise when trying to compose arbitrary atomic transformation parts are avoided.

An automated framework DUALLY [14] aims to answer the issues concerning tools and languages interoperability. This approach introduces the concept of transformation
generation with the purpose of translating model specifications from one language to another. The transformation generation is based on a mapping of elements between these languages. In current state it is not able to generate refinement transformations. Lately, Herrmannsdoerfer et al. introduced a language for Coupled Evolution of Metamodels and Models COPE [15]. COPE is proposed as a solution to the problems that arise due to metamodel evolution and the resulting necessary model migration to a corresponding newer version of metamodel. This approach is based on a reusable migration transaction library that is used for migration transformation creation. Although, it provides advanced means for reuse of migration transactions and migration transformations, it is not suitable to express the variabilities of applications families based on metamodel-independent feature configurations.

8. Conclusion and future work

In this paper, we described how model refinement transformation can be configured and automatically generated in the model-driven software development process. The participating roles in this process have defined responsibilities and development artifacts. The configuration process is based on explicitly defined feature models that also bear the necessary transformation fragments to give their contribution to the refined application model.

Additionally, we have identified patterns that occur in a feature model based configuration and that are important for the realisation of refinement transformation generators. This generation is based on the specification of higher-order transformations.

The presented work is a part of continuous research on the automatic transformation composition and generation. An experimental evaluation and efficiency study is planned for the future.

Future work will deal with the composability of feature models and its effects on the specification and composition of the transformation fragments. Furthermore, extended properties of feature models defined by weak constraints, such as recommendations for feature configurations will be investigated. Additionally, we will investigate if it is feasible to soften the constraint on the disambiguation rules to allow for disambiguation of transient permutations of inclusive-or features. Another issue that should be investigated are dependencies between transformation fragments when feature model dynamically changes.

References
