On the Refinement of Model-to-Text Transformations

Salvador Trujillo, Ander Zubizarreta

IKERLAN Research Centre, Spain
{strujillo, ander.zubizarreta}@ikerlan.es

Abstract. Model Driven Development (MDD) is a paradigm to automate the generation of code. A key artifact in this paradigm is a model transformation which defines the mappings from a model to another model or even a code artifact. Although MDD was initially aimed at the generation of an individual program, shortly after appeared the need for families of programs. Hence, the combination of Model Driven and Software Product Lines (SPL) appears as a promising paradigm. Most of the previous work was focused on the necessity to support variability on models, but little work has been done so far on supporting the variability of remaining MDD artifacts such as model transformations or metamodels. This work first motivates the need for variability of model transformations. We address then the application of step-wise refinement to model-to-text transformations expressed in MOFScript. We illustrate this with a case study.

Introduction

Modeling is essential to cope with the increasing complexity of current software systems. Models assist developers during the entire development life cycle to precisely capture and represent relevant aspects of a system from a given perspective and at an appropriate level of abstraction.

MDD is a paradigm to automate the generation of boiler-plate code. Raising the abstraction level enables to focus on the domain details and separate the implementation details. This brings a number of specific benefits such as productivity, reduced cost, portability, drops in time-to-market, and improved quality. Overall, the main economic driver is the productivity gain achieved, which is reported by some studies [14,19].

A key artifact in MDD is a model transformation that defines the mappings between a model and another model or between a model and a code artifact. Although MDD was initially aimed at the generation of an individual program, shortly after appeared the need for families of programs.

Researchers and practitioners have realized the necessity for modeling variability of software systems, where software product line engineering poses major challenges [25]. A software product line is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [5,21].
For example, in industrial software systems the presence of different types of subsystems (e.g., exclusive subsystems from different providers) implies that each is controlled in a similar though different way. This is typically achieved by defining two features that are not necessarily present in all possible systems. A feature is an end-user visible behavior of a software system, and features are used to distinguish different software systems or variants of a software product line [16].

This impacts not only on the implementation, but on the modeling level. The modeling used in software product lines can be twofold. First, there are approaches for describing the variability of a software product line, e.g., there are feature models that specify which feature combinations produce valid variants [16]. Second, all variants in the product line may have models that describe their structure, behavior, etc.

However, when dealing with variability in an MDD scenario, there are further artifacts apart from models that may need to cope with such variability. Model transformations are a case in point. In some scenarios, they may have too to cope with the variability imposed by the software product line. Hence, this paper takes a step back to study such impact into a broader perspective by analyzing the scenarios for Model Driven Product Lines.

We shift our attention from the variability of models to a more general situation where the variability may embrace models, metamodels and model transformations. Hence, a realization of one feature may consist of variations of such artifacts. This work specifically analyzes the variability of model transformations.

The contribution of this paper is to apply step-wise refinement to model transformations. MOFScript-based model-to-text transformations are refined by using XAK [1]. We illustrate our ideas with a simplistic case study, which is inspired on the industrial cases we work with.

We begin by reviewing the background.

Background

*Model Driven Development* is a paradigm where models are used to develop software. This process is driven by model specifications and by transformations among models or models and code. It is the ability to transform among different model representations that differentiates the use of models for sketching out a design from a more extensive model-driven software engineering process where models yield implementation artifacts. This paradigm eases cumbersome and repetitive tasks, and achieves productivity gains.

The main artifacts in MDD are models, metamodels and model transformations.

**Models.** Model is a term widely used in several fields with slightly different meanings. A model represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system [17]. Typical
Figure 1 shows a simple statechart model showing the behavior of a simple temperature sensor. The statechart has states $s_{\text{Closed}}$ and $s_{\text{Opened}}$ in addition to initial and final states. Transitions between states and the conditions for these transitions are described in the model. The actions to be executed should be defined too, but are omitted in the example.

Metamodels. A model is frequently considered an instance conforming to a metamodel. A metamodel is a model of a modeling language where the language is specified [17]. In other words, the metamodel describes the elements of the domain and their relationships.

A metamodel for a statechart might define its elements such as states, transitions, their relationships, etc. Note that the model in Figure 1 might conform to such metamodel. The mappings between metamodels are typically defined by model transformations.

Model Transformations. Model transformations play a pivotal role in MDD because they turn the use of models for sketching into a more extensive model-driven usage where implementations can be directly obtained [23]. A model transformation is the process of converting one model to another model of the same system [20]. In general, a model transformation is the process of automatic generation from a source model to a target model, according to a transformation definition, which is expressed in a model transformation language [17].

Depending on the target, model transformations can be model-to-model or model-to-text transformations. The former takes as input one or more models conforming to given metamodels and produces one or more models conforming to the same or another metamodels. The latter produces text as its output, that can be implementation code, documentation, or any other textual form.

Model-to-model transformations usually make use of rules that are defined as mappings between input and output metamodels. Model-to-text transformations combine rules with text templates that define the form of the output text.
Fig. 2. Example of a model-to-text transformation using MOFScript

There is a variety of open-source and commercial tools and languages for model transformations\(^1\).

Our case study focuses on model-to-text transformations defined using MOFScript transformation language. MOFScript language is a metamodel-independent language that allows to use any kind of metamodel and its instances for text generation. MOFScript tool is based on EMF and it is available as an Eclipse plugin.

The definition of a MOFScript transformation consists of rules. Figure 2 shows a base transformation definition for transforming a statechart model into textual code. Implementation-wise, a transformation module called `baseTransf` is defined. Such module is composed by several rules. They can have a context type scoping to which metamodel elements can be applied such rules\(^2\).

**On the Need for Variability**

A Model Driven Development scenario typically involves models, metamodels and model transformations. Next, we analyze the need for variability of model transformations.

\(^1\) ATL [4], RubyTL [7], ATC [24] and other tools are for model transformations. Though these tools may be also used (and so we do) for model-to-text transformations, we focus on MOFScript since it is specialized on model-to-text.

\(^2\) The execution of the transformation in Figure 2a starts with the `sc.ScxmlType::main()` rule. This rule is applied to the context type scoped by the type `sc.ScxmlType` where `sc` is the input model. That means that it applies to every `ScxmlType` in `sc`. The output text file is defined with `file()`.
Scenario. There are different scenarios when combining MDD and SPL. Consider the differences on the modeling language used: it is not the same to use UML or a Domain Specific Language (DSL).

There are scenarios where model variability may be enough and the variability of model transformations may not be needed. This may happen in situations where the used metamodel is standard and so it is not subject to variability. For instance, when using a UML class diagram, it seems unlikely to make its metamodel variable since it is somehow standardized. This may apply generally to the metamodels within the UML. A similar situation occurs when the model transformations come from a common library of model transformations that are shared. For instance, consider the dozens of model transformations expressed in ATL that are available online\(^3\).

Therefore, in scenarios with standard metamodels and/or shared transformations, the use of model variability seems enough and thus variability of model transformations may not be needed. However, there exist other scenarios where the variability of model transformations may be needed.

Consider the case in which different models and model transformations need to be customized for different targets. Model transformations share a significant common part while differing in some variable parts. For instance, different implementation code can be generated from the same source model. The target code is expected to be executed in different platforms with different programming languages. This situation could be well handled by defining features of the software product line. There is a large proportion of shared code and some particularities are bound to each programming language. In such situation, the application of variability to model transformations may enable to handle those differences in a unique model transformation. Next, we illustrate this with our case study.

Case Study. Although this motivating scenario is realistically more likely to occur within a larger system, we illustrate our ideas with a family of control software systems developed following MDD. Particularly, we focus on the control of a simple temperature sensor that opens/closes some device (e.g. water pump). Our motivating scenario demands to cope with the variability of models, metamodels and model transformations. The files used in this example are available online\(^4\).

Consider a model of a simple state machine and a model-to-text transformation definition to get implementation code from it. The aim is to generate a switch statement implementing the statechart model. There are however different target platforms for the generated code. Such different programming languages are Ada and Java. Switch statements differ from Ada to Java. Actually, Ada does not provide a proper switch statement, but a case statement with equivalent structure and behavior to Java’s switch.

It would be possible to define a separated and independent model transformation definition for each target language. In that way, however, it would be

\(^{3}\) http://www.eclipse.org/m2m/atl/atlTransformations/

\(^{4}\) http://www.ikerlan.es/softwareproductline/jisb2009examples.zip
necessary to define a new transformation definition for each new target language
we would like to generate code for. Doing so, no reuse will be achieved.

The structure of the switch statement is similar in most of the programming
languages. Its difference mostly lies in the syntax. Hence, it would be possible to
define a base transformation common for all target platforms and to refine it with
partial or refined transformations incorporating target specific elements. This is
what we called in this work as the refinement of model-to-text transformations.

Refinement-based Variability

The refinement of model-to-text transformations is introduced in order to enable
their customization for different variability needs. In our example, there is a
statechart as a source model, Ada and Java as target programming languages
and a model-to-text transformation with variability incorporated to connect the
source with different targets.

As a case in point, we considered the generation of switch statements that
implement the code of a statechart model. Although switch statements are com-
monly found in most of current programming languages, they are not exactly
defined in the same way. In our example, we define a base transformation with
all the common elements of the switch, that will be refined for each output
language. There will be a transformation refinement associated to each target
language, each adding its specific elements.

Base of a Transformation

```xml
<MOFScriptModel:MOFScriptSpecification ... xak:artifact="base.m2t.model.
mofscript" xak:feature="baseTrans">
  <transformation line="6" name="baseTransf" xak:module="transModule">
    <variables line="7" column="2" name="outputFile" type="String">
      <xak:module="oFileModule"/>
    </variables>
    <parameters line="6" column="39" name="sc" type="http://www.w3.org
      /2005/07/scxml">
      </parameters>
    <transformationrules line="64" name="writeSwitchStart" xak:module="
      switchStartModule">
      </transformationrules>
    </context line="64" name="self" type="module"/>
  </transformationrules>
</transformation>
</MOFScriptModel:MOFScriptSpecification>
```

Fig. 3. Base Transformation (XMI representation)

A base transformation definition has been defined with MOFScript (see Fig-
ure 2). This definition takes a statechart model as input, and has the rules to
generate a switch statement as output. Since the switch statement differs among
different languages, this base transformation only has the common elements.
In our example, the base transformation defines some common rules to generate a switch statement based on the statechart model. Some rules for writing the output are defined (e.g., writeSwitchStart) but they are empty due to the fact that the output text varies depending on the syntax of the output programming language. In the base transformation a variable named outputFile is defined too, to specify the name of the output file, but this variable is not yet given a particular value. The assignment will be done with a refinement, which will assign a name related to its output language later on.

MOFScript allows to represent a transformation definition as a model, conforming to the MOFScript metamodel. Figure 3 shows the model representation of the transformation using XML Metadata Interchange (XMI). (Note that this figure is equivalent to the textual representation of the transformation in Figure 2.) The transformation module is defined with the <transformation> element. Nested to this element, variables, parameters and rules are defined using the <variables>, <parameters> and <transformationrules> elements, respectively.

Actually, the base transformation can not be executed standalone since it is not complete, so it may not generate the expected implementation. The base transformation needs to be refined for each output language.

**Refinement of a Transformation**

```xml
<axk:refines axk:artifact="base.m2l.model.moofscript" axk:feature="
  adaDelta" ...>
  <axk:extends axk:module="oFileModule"/>
  <value xsi:type="MOFScriptModel:Literal" value="example.adb"/>
  <axk:extends axk:module="switchStartModule"/>
  <value xsi:type="MOFScriptModel:Literal" value="case_state_is
  &x9;&x9;"/>
  <axk:extends>
  <value xsi:type="MOFScriptModel:PrintStatement">
    <printBody xsi:type="MOFScriptModel:Literal" value="false"/>
  </value>
  </axk:extends>
</axk:refines>
```

**Fig. 4. XAK-based Refinement of a Transformation**

A refinement can be deemed of as a function that takes an artifact as an input, and returns another similar artifact which has been leveraged to support a given feature [3]. XAK is a language for defining refinements in XML documents and provides a composer tool for them [1]. We used it for representing and composing our MOFScript-based transformations. MOFScript offers the option to represent
a transformation definition as a model. Such model can be represented using XML. Therefore, XAK can be used for refining such transformations.

Any traditional XML document can be a base document, but some additional annotations are needed (see Figure 3). The attributes @xak:artifact and @xak:feature are added to the document root element. The first one specifies the name of the document that is being incrementally defined, while the second one specifies the name of the feature being supported ("base" for base documents). To specify which elements are modularizable, @xak:module is used. That is, it indicates those elements that play the role of modules, and henceforth can be refined. In general, a XAK module has a unique name.

Refinement documents refine base documents and they have <xak:refines> element as root (see Figure 4). Its content describes a set of module refinements (i.e. elements annotated with the xak:module attribute) extending a given base document (i.e. the xak:artifact attribute). The xak:super node is a marker that indicates the place where the original module body is to be substituted. In general, a XAK refinement document can contain any number of xak:module refinements.

Those elements of the transformation that can be refined must be defined before refining such transformation. In our example, the refinement units are the transformation definition itself, variables and rules. That is, we can add new rules or variables to the transformation definition, we can add or set values to variables already defined, or we can extend existing rules with new statements. We add xak:module attribute to <transformation>, <transformationrules> and <variables> elements to specify that they are modularizable (see Figure 3). Note that not all elements are modularizable, but only those we designate beforehand by xak:module.

Figure 4 shows a refinement for the base transformation. This refinement refines the base transformation definition to generate the code in Ada programming language. Some of the rules in the base transformation are refined adding Ada specific elements. The variable outputFile is given a value. The rules that are refined are those which are used to produce specific code to the output file5. Since the syntax differs between languages, those rules must be refined for each language.

**Composition of Base and Refinements**

XAK composer tool is used to compose the refinement to the base transformation, executing the following command:

```
> xak -xak2 -c baseTrans.mofscript adaDelta.xak -o composedAdaTrans.mofscript
```

---

5 Due to the lack of space, in the figure only the refinement of the rule writeSwitchStart is shown. The complete example is available for download at http://www.ikerlan.es/softwareproductline/jisbd2009examples.zip
Fig. 5. Composed Transformation: (a) XMI representation; (b) MOFScript representation

Fig. 6. Generated code: (a) Ada; (b) Java
The result of the composition is a composed transformation model (Figure 5a), which can be executed directly giving as input the statechart model, or can be converted to a MOFScript transformation definition file (Figure 5b) using that option in MOFScript tool and then executed. Note that some rules that were empty in the base transformation definition, have now some body specifying how the output has to be in Ada programming language. So, the customization of model transformations is achieved. The output of the transformation is the example.adb file which has the switch statement according to our model, written in Ada syntax (Figure 6a).

Another possible refinement may be to refine the base transformation definition to generate Java code. In this case, the rules should be refined to generate Java code. Although the structure of the generated code is almost the same, the syntax differs from Ada to Java. Additionally, break; sentences are necessary in Java, so a new rule needs to be added to the base transformation to write such sentences. The new rule will be called from generateSwitchCases rule, that need to be refined to add the new statement. Figure 6b shows the code obtained when running the transformation refined for Java.

The use of refinements in the context of model-to-text transformations enables the reuse and customization that Software Product Lines promote.

**Related Work**

Merging MDD and product lines is not new, we know of examples that explicitly use features in MDD [9,8,10,11,12,22]. One is BoldStroke: a product-line for supporting a family of mission computing avionics for military aircraft [12]. Czarnecki introduces super-imposed variants and model templates to map features to models [8]. Weber et al. introduce the Variation Point Model that models variation points at the design level [26].

There is a line of work on feature-based composition of models. Feature-oriented model-driven development is an approach that ties feature composition to model driven development [25]. Recent work by Apel describes superimposition as a model composition technique to support variability of product lines [2]. FeatureMapper is a tool that supports mapping features from feature models to solution artifacts [13]. These works do not yet consider the composition of model transformations or metamodels.

Sanchez-Cuadrado presents an approach for the reuse of model transformations in RubyTL by using an idea reminiscent of libraries in programming [6]. This first step towards model transformation reuse does not yet incorporate the notion of product family. The superimposed modules of ATL language can be composed into different transformation definitions. This is not related to features, neither to the notion of composition demanded in a product family scenario [15]. Oldevik proposes an aspect-based extension of the MOFScript model-to-text transformation language, which is called a Higher Order Transformations (HOT) [18].
Conclusions

This paper presented an approach for the variability of MOFScript-based model-to-text transformations in a software product line scenario. The main contribution is the application of step-wise-refinement in the context of a model transformation. Doing so, the application of variability shifts from scenarios focused only on model variability towards broader scenarios handling the variability of MDD-related artifacts.

We claim the need to shift our research attention from the variability of models to a broader perspective embracing the variability of models, metamodels and model transformations.

This is indeed the direction of our current research efforts where we are addressing the variability of models, metamodels, and model transformations and their relationships concomitantly, since we believe they are often closely inter-related.

Acknowledgments. This work was co-supported by the Spanish Ministry of Science & Innovation under contract TIN2008-06307-C02-02. We thank to Maider Azanza for her comments on earlier drafts.

References


