Using Software Product Lines to Manage Model Families in Model-Driven Engineering

Orlando Avila-García  
Open Canarias, S.L.  
Santa Cruz de Tenerife, Spain  
orlando@opencanarias.com

Antonio Estévez García  
Open Canarias, S.L.  
Santa Cruz de Tenerife, Spain  
aestevez@opencanarias.com

E. Víctor Sánchez Rebull  
Open Canarias, S.L.  
Santa Cruz de Tenerife, Spain  
vsanchez@opencanarias.com

ABSTRACT

The relationship between Software Product Lines (SPL) and Model-Driven Engineering (MDE) is not new in the literature. It mainly focuses on the use of Domain-Specific Languages to specify application families, rather than using the more classic feature models. However, more recent works propose another important synergy: the use of feature models to specify model families. In this paper we propose a Domain-Specific Transformation Language (DSTL) that helps in the creation of SPLs to manage model families. Moreover, we show the benefits coming from taking a new approach to DSTL development. In this approach, DSTL instances are not compiled into source code but transformed onto general-purpose transformation languages in order to be executed using already existent transformation engines.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software—Domain engineering, Reuse models; D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE); D.2.9 [Software Engineering]: Management—Software process models

Keywords

Model-Driven Engineering, Software Product Lines, Domain-Specific Languages, Model Transformations, SPEM

1. INTRODUCTION

Model-Driven Engineering (MDE) has emerged as the most promising paradigm in software engineering [24]. MDE advocates the use of models not just for documentation and communication purposes but as first class artifacts, to generate other work products during the development process, such as other models, source code, configuration files, etc. Modeling languages play a central role in MDE. They range from the more generic modeling languages like UML (Unified Modeling Language) [23] to the so-called Domain-Specific Languages (DSL) [6], that is, formal languages whose constructs represent concepts from a specific problem domain. Regardless of the life cycle stage where a modeling language is applied and its degree of specificity, we can benefit from the identification of commonalities and differences between its instances, that is, from the identification of model families.

In a Software Product Line, a family of software applications is developed from the same core assets in a prescriptive way [5]. Derivation of family members is carried out by configuring or assembling these core assets, following the selection, within a feature model [16, 9], of the features that identify the family member. Using this approach, time and costs of application development are reduced. In the same way that software product lines identify commonalities and variabilities within software application families, we might benefit from identifying model families that represent the most common modeling problems we come across during our developments. In this way, we do not have to start modeling from scratch every time we carry out a new development.

The relationship between Software Product Lines and MDE is not new in the literature. It mainly focuses on the possibility of specifying family members by using DSLs rather than the more classic feature models [14, 7]. However, more recent works propose another important synergy: the use of feature models to specify model families instead of application families. In [8] it is proposed that all variants within a model family may be identified by a feature model and represented in a superimposed form within a single model template. Model template objects are then annotated with presence conditions (Boolean expressions) of features from the feature model. Derivation of the desired family member is done by a specialization of the model template after resolving those expressions. A newer approach ([10]) proposes the use of OCL (Object Constraint Language) [25] constraints to specialize the template instead of using annotations1. This second approach provides more powerful specialization mechanisms as well as overcome the problem of having annotations scattered all over the model template. However, this approach makes use of OCL notation, which is a general-purpose constraint language.

In this paper we propose a domain-specific language to specify transformations (in this case specialization) of model templates based on feature models. We call our language Model Template Transformation Language (MTTL). This language will help a SPL engineer in specifying rules to specialize a model template. Nevertheless, this is not just an-

1Please refer to [8] and [10] to see an overview of related approaches that do not make use of feature models.
other domain-specific language, for which we will have to create a compiler or virtual machine in order to be executed. We propose to transform MTTL models onto ATC (Atomic Transformation Code) [11] models; ATC is a generic model transformation language for which there already exists an implementation and execution (transformation) engine.

First, Section 2 describes the development and execution of a product line of a model family. Section 3 explains then the rationale behind the use of domain-specific transformation languages and Section 4 presents our Model Template Transformation Language (MTTL). Section 5 shows an example and Section 6 draws some conclusions.

2. MODEL PRODUCT LINE

A Software Product Line (SPL) approach to software development implies two related processes [5]: SPL development, to create the SPL architecture, and the application development, to execute the SPL to produce a concrete product.

![Figure 1: SPEM model that describes the product development process for a model family (see text).](image1)

Our goal is to create a SPL to derive a family of models that conform to a particular DSL. Like in [8, 10], the model family is described by a feature model and a model template. Figure 1 shows a SPEM 2.0 (Software Process Engineering Metamodel) [21] model depicting the execution process of such a SPL. The product engineer must choose (Choose Features task), from a feature model, those features that identify the concrete model he wants. The result is a feature configuration, that feeds an automatic specialization of a model template. This transformation is specified as an ATC (Atomic Transformation Code) [11] model in order to be executed in the ATC execution engine.

Figure 1 shows that the product development process has three input (depicted as <<in>>) artifacts: the feature model, the model template and the ATC transformation model. These artifacts are work products of the SPL development process (Figure 2), and are necessary to execute the SPL. It might not be trivial for a SPL engineer to describe the template specialization in a generic transformation language like ATC. It is within this process where our Model Template Transformation Language (MTTL) plays its role. We simplify his job by providing him with this DSTL to describe those transformations, with specific constructs for his problem domain.

3. DOMAIN SPECIFIC TRANSFORMATION LANGUAGES

QVT (Query/Views/Transformations) [20] is an OMG (Object Management Group) standard model transformation language that is expected to be used across companies and projects. It is also a generic language that allows specifying many variants of model transformations. Although the benefits of being generic and standard are clear, it is also clear that leveraging the level of abstraction with domain-specific transformation languages (DSTL) might bring us the same benefits as when using any other DSL.

Some DSTL approaches propose languages that are compiled into source code in order to be executed. For example, [12, 13] propose a language to specify aspect weaving. Likewise, [22] proposes an imperative language to specify transformations from BPEL (Business Process Execution Language) [1] to UML [23] models. In both cases DSTL instances are compiled into source code to be executed.

Two main problems arise with this approach. Firstly, they lack the benefits of using a standard transformation language like QVT. Secondly, we must create new tooling and facilities (e.g. editors and compilers) for each DSTL. Although there exists active work on framework engineering to help in the creation of such facilities (e.g. [22]), in our opinion, a more MDE approach might tackle those problems. This new approach would see the use of DSTL not as stand-alone transformation languages but as languages that transform onto a general-purpose transformation language in order to be executed. For example, [2] proposes a Weaving Metamodel (WMM) to specify model composition. Its instances are transformed onto ATL (ATLAS Transformation Language) [15] to be executed in its related execution engine. This approach benefits from (1) keeping DSTL as a sort of platform-independent model (PIM) [18], (2) reusing an already existing model transformation engine, and (3) avoiding creating a new compiler for every DSTL.
Figure 3: CBFM metamodel

Figure 4: MTTL metamodel

Figure 5: ATC model generated from a MTTL model that contains a single Select operation.
4. MODEL TEMPLATE TRANSFORMATION LANGUAGE

This section presents Model Template Transformation Language (MTTL), a DSTL that helps a SPL engineer in specifying transformations of model templates from feature models. MTTL models are transformed onto ATC (Atomic Transformation Language) [11] models, a generic transformation language with an already existent execution engine.

We have used Eclipse 3.2 [4] and its Eclipse Modeling Framework 2.2 (EMF) [3] metamodeling platform. Therefore, all DSL metamodels used throughout this work have been modeled in Ecore, the Essential MOF (Meta Object Facility) [19] implementation of EMF. We have also used EMF to generate a tree editor for every DSL.

The domain of our DSTL is the transformation or specialization of model templates following a set of selected features (feature configuration) from a feature model (see Figure 1). The model template contains all model variants of the model family in a superimposed form [8, 10]. We have restricted the problem domain to those specializations done just by creating and deleting relations between objects. Nevertheless, the deletion of a containment relation will also mean the deletion of the target object from the template.

In our application domain, feature models will be instances of the Cardinality-Based Feature Model (CBFM) [9] metamodel (see Figure 3). The feature configuration model is a specialization of the feature model [9], which represents the model family of the product line, that is, all possible model variants. MTTL assumes that the metamodel of the configuration model is also CBFM. On the other hand, model templates can be instances of any metamodel. Therefore, we will have to specify within the MTTL transformation the metamodel to which the template conforms.

Following our experience, we have established four necessary operations for the MTTL problem domain:

- **Clear.** Through this operation all links that are part of a 1 to n relation are deleted. Its parameters are: (1) the name of the feature that triggers the operation, (2) the type (metaclass) and name of the source object of the relation, and (3) the name of the property that represents the relation within the source object.

- **Select.** Through this operation all links that are part of a 1 to n relation but one are deleted. Its parameters are the same as for CLEAR plus the name of the target object whose relation is to be maintained.

- **Remove.** Through this operation a single relation between two objects is deleted. Its parameters are the same as for SELECT.

- **Insert.** Through this operation we insert a new non-containment relation between two objects of the template. Its parameters are the same as for SELECT plus the type (metaclass) of the target object whose relation is to be created.

Figure 4 shows the MTTL metamodel. The root object of a MTTL model must be an instance of the Transformation metaclass, that contains Mapping instances representing different transformation operations. The targetMetamodelURI property of the Transformation metaclass allows us to specify the URI (Uniform Resource Identifier) [3] of the template metamodel. This enables the possibility of using MTTL to specialize model templates conforming with different DSLs. The property feature of the metaclass Mapping refers to the name of the feature (from the feature model) that triggers the operation. It is important to note that the transformation assumes that every object susceptible to be manipulated from the model template has got a unique name property to be distinctly identified. Note that there might exist other approaches to identify model template objects, such as AOP annotations [17] or EMF URIs [3].

We have specified the transformation between MTTL and ATC with another ATC model (ATC Model A in Figure 2). This transformation fills in or specializes an ATC model template that contains a function per MTTL operation. The ATC template gets specialized by introducing, as children of the Body Block of the Main object, the set of objects necessary to perform the call to those functions (see Figure 5). As said before, it is not new to use not only feature models but also any kind of DSL to specify family members of a software product line [14, 7]. It is interesting then to see this transformation as a product line for ATC models. This product line assists the SPL engineer in generating the fami-
5. AN EXAMPLE

In this example we create and execute a product line for a family of StateMachine models, a DSL for state machine modeling. The family has three members, that is, three variants of StateMachine models. These variants represent three different ways in which a hypothetical customer of an online shop would validate himself to enter the site: direct, logging or IPFilter.

First of all, the SPL engineer must develop the product line for StateMachine models, so a feature model, a model template as well as the ATC transformation to produce a family members must be created (see Figure 2). The feature model must contain three features (direct, logging and IPFilter) identifying the three variants (see Figure 6). The StateMachine model template must contain the three model variants in a superimposed form (see Figure 7). Finally, the SPL engineer must create the ATC transformation that will specialize the template. In order to do so, the engineer creates a MTTL model (see Figure 8) containing three Select objects, each object representing the operation to be executed when a feature is choosen. Figure 8 shows the properties sheet of the Select object that represents the operation to be executed when the transformation finds the logging feature within the configuration model. Finally, the MTTL model is transformed onto ATC (by ATC Model A in Figure 2) to make it executable over the ATC execution engine.

To continue with the example, we have a product engineer that desires to get the model variant identified as logging. He must choose the logging feature from the feature model and create a feature configuration, that is, a feature model that contains just the logging feature. This feeds the ATC transformation (ATC Model B in Figure 1 and 2), which is executed to specialize the model template as to get the logging variant (see Figure 9).

6. CONCLUSION

In this paper we have proposed Model Template Transformation Language (MTTL), a domain-specific transformation language (DSTL) that helps a SPL engineer to specify rules of model template specialization in a product line of models. We have taken an approach where the DSTL instance is not compiled into source code but transformed onto a generic model transformation language. In this case we have used ATC (Atomic Transformation Language) [11], which has its own execution engine.

We have shown how the creation of the transformation between our MTTL and ATC is quite straightforward, as we take advantage from the close abstraction level between the two ontologies. Although the ATC model that specifies that transformation may be considered as a compiler, the amount of programming skills required to create it is much lesser than for creating a compiler to source code. For instance, knowledge of the EMF’s API was not required, which would have been indispensable for creating a MTTL compiler. In this case study, it took an engineer (non-expert on ATC) one week to develop the MTTL metamodel, the editor (automatically generated by EMF) and the ATC model to transform MTTL instances onto ATC.

It is also important to take the benefits coming from the platform-independence of MTTL into account. Firstly, it allows us to transform and execute its instances onto different generic transformation language platforms, such as QVT (Query/Views/Transformation) [20], ATL (ATLAS Transformation Language) [15] or ATC. Secondly, the use of MTTL, which is susceptible to be transformed into a standard language such as QVT, allows us to maintain homogeneity in the specification of transformations across companies and projects.

Finally, throughout this paper we have seen the synergy between Software Product Lines and Model Driven Engineering. It is very interesting to consider some transformations between models as product line executions. In this approach, we not only use feature models but any Domain-Specific Language (DSL) to manage variants within a model family. In this work we have seen an example of using feature models to manage a family of DSL models, and MTTL models to manage a family of ATC models.
7. ACKNOWLEDGMENTS

This paper has been supported by the Ministerio de Educación y Ciencia (PTQ2004-1495 and PTR1995-0928-OP), the Fondo Social Europeo, and the DG de Universidades e Investigación del Gobierno de Canarias (PI042005/007).

8. ADDITIONAL AUTHORS

Additional authors: José Luis Roda García (ETSI Informática, Universidad de La Laguna, Santa Cruz de Tenerife, Spain, email: jlroda@ull.es).

9. REFERENCES