An approach for evolving transformation chains[⋆]

Andrés Yie[1,2], Rubby Casallas[1], Dennis Wagelaar[2], Dirk Deridder[2]

[1] Grupo de Construcción de Software, Universidad de los Andes, Colombia
{a-yie, rcasalla}@uniandes.edu.co

[2] Software Languages Lab, Vrije Universiteit Brussel, Belgium
{ayiegarz, dennis.wagelaar, dirk.deridder}@vub.ac.be

Abstract. A transformation chain (*TC*) generates applications from high-level models that are defined in terms of problem domain concepts. The result is a low-level model that is rooted in the solution domain. The evolution of a TC is a complex and expensive endeavor since there are intricate dependencies between all its constituent parts. More specific, an evolution problem arises when we need to add an unanticipated concern (e.g., security) that does not fit the expressiveness of the high-level metamodel, because such an addition forces us to adapt existing assets (i.e., metamodels, models, and transformations). We present a solution that adds a new concern model to the TC, in an independent way.

1 Introduction

Model-Driven Engineering (*MDE*) implementations promote the use of models expressed in terms of problem domain concepts (e.g. Bank Account, Insurance Claim) as the prime artifact to develop software. These models, to which we refer as high-level models, are used as input for a transformation chain (*TC*). A TC is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model, which is rooted in the solution domain. In addition to the translation from problem domain concepts to solution domain concepts (e.g., mapping a Business Entity onto a Java Class), the TC adds implementation details in every transformation step.

For high-level models, the metamodels are rooted in the problem domain. These metamodels define the abstract syntax of a domain-specific modeling language (*DSML*) that is suitable to be used by domain experts [1]. For the low-level models, the metamodels are rooted in the solution domain. These metamodels are typically closer to the definition of general-purpose languages (*GPLs*).

The particular problem we address is the addition of a new concern (e.g., security, monitoring, etc.) that was not anticipated in the existing MDE implementation. No real problem arises if the new concern can be cleanly expressed using the existing high-level metamodel. However, if this is not the case, then a

[⋆] This research was performed in the context of the Caramelos project (VLIR), the VariBru project (ISRIB), and the MoVES project (IAP, Belgian Science Policy)

number of problems arise when trying to extend the existing high-level metamodel with new concepts (e.g., the notion of security in a business domain metamodel): 1) the existing metamodel will be polluted with concepts that do not belong to its main problem domain, 2) including all the new elements in the core application model produces a single monolithic model which is detrimental to the overall maintainability, and 3) the new concepts will impact the TC by imposing intricate changes (adding, updating or deleting TC elements) to its existing implementation, which increases the complexity and the number of dependencies within the TC. These changes increase the dependencies among the steps in the TC. These problems make it hard to evolve an existing MDE implementation and to maintain applications.

To overcome these problems we propose a strategy that consists of specifying the new concern in a separate high-level model. This leaves the original model unaltered and oblivious of the added concern. The concern-specific model can thus be specified using concepts close to its domain which is expressed in a separate meta-model. Therefore, we have two high-level models that conform to two different metamodels. Consequently, to obtain the final application, it is necessary to compose both models. If we perform a high-level composition, then we face a *heterogeneous composition* because both models conform to two different metamodels. A heterogeneous composition is a complex task and requires a particular composition mechanism for every added concern. Therefore, we chose to align the high-level models using a *Correspondence Model* (CM) [2], which explicitly describes the relationships among the elements of different models. We use these correspondence relationships to identify the elements to compose.

We have developed a mechanism to automatically derive the CM through the various steps in the TC. The actual composition is postponed until the lowest level. At this level, every model conforms to the same metamodel (e,g,. Java metamodel), or to metamodels that are extensions of this metamodel. Having models that conform to the same low-level metamodel and a low-level CM relating these models allows us to perform a *homogeneous composition* (e.g., composition of two Classes). This reduces the complexity of the composition and it gives the means to use a single composition mechanism for multiple concerns. In our case study, we use a model composition strategy based on the *UML Package Merge* [3] mechanism that composes the low-level models into a single model that conforms to the existing low-level platform metamodel.

2 Approach overview

The overall approach is to add a new TC next to the existing MDE implementation that takes a high-level concern-specific model as input and produces a low-level concern model as output. We align the new high-level model with the original one by using a *Correspondence Model* (CM), which needs to be propagated through the TC. The main challenge is to define a mechanism to automatically derive the new correspondence relationships, having in mind that the TC increments the complexity of the models by adding elements at each step. Once we reach the lowest level, the models conform to the same existing metamodel (e,g,. Java metamodel), or conform to an extension of it. Therefore, both TCs produce two complementary low-level models that can be composed using a common composition mechanism.

To derive a low-level CM it is necessary to trace back the elements of the low-level models and to check if they come from pairs of related elements in the high-level. With a trace model (TM) [4] we determine the elements in both low-level models that come from a couple of related elements in the high-level. For instance, an Attribute in the business model is transformed into an Attribute, a GetterMethod and a SetterMethod in the low-level model. In the security model a ResourceAttribute with a ReadPermission is transformed in a private Attribute and an annotated ReadMethod in the low-level security model. Therefore, it is necessary to trace back all these low-level elements and verify that the high-level source element (Attribute) from which they originate, is related with a correspondence relationship to the high-level concern-specific element (ResourceAttribute).

Once the elements in the low-level models that have a pair of correspondent elements as sources are determined, we have to relate these elements by identifying the correct match for each one. For instance, a GetterMethod (in the low-level application model) can be related to a ReadMethod (in the low-level security model) but not to a WriteMethod. To avoid, erroneous correspondences, the modeler has to specify some constraints. A constraint is a relationship between two metaclasses that defines if the correspondence link between the concepts that conform to them can be established or not. In our solution this set of constraints is called a Derivation Model (DM).

Figure 1 presents the general schema of our approach. The original TC is in the left (MM_{bus}, M_{bus}, MM_{java}, M_{java}, T_1)[1]. The concern TC is presented in the right (MM_{sec}, M_{sec}, $MM_{sec-java}$, $M_{sec-java}$, T_2). $CM_{high-level}$ is the high-level correspondence model that aligns the two high-level models. TM_A and TM_S are the trace models that relate the high-level models with the low-level models. The DM relates the low-level metamodels with constrains between their metaclasses. The DM is used to generate the transformation T_3, that uses the trace models and the $CM_{high-level}$ to generate the $CM_{low-level}$. Finally, the low-level models are composed and transformed into code by the original model-to-text transformation (G_1).

3 Derivation of Correspondence Model and Composition

The key element in our approach is the derivation of the low-level CM in order to perform an homogeneous composition which we will briefly detail below.

We align the two high-level models using the $CM_{high-level}$ which relates the elements to be composed. For example, the business model (M_{bus}) contains the Attribute *dueDate* and the security model (M_{sec}) contains the Resource *date*

[1] MM = Metamodel, M = Model, T = Transformation chain

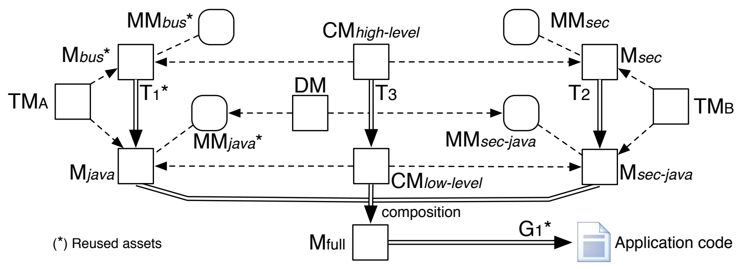

Fig. 1. General Schema

that needs to be protected. These two elements are related by a correspondence relationship in $CM_{high-level}$. The modeler creates these correspondence links because he knows the meaning of the relationships between elements.

Low-level correspondence relationships are derived automatically by the transformation (T_3). For instance, two elements a' and b', from M_{java} and $M_{sec-java}$ models respectively, will have a correspondence relationship if: 1) There is a CM relationship at the higher level between a and b, where a' was produced from a by T_1, and b' was produced from b by T_2. 2) The metaclasses ma' and mb' where a' conforms to ma' and b' conforms to mb', allow for a correspondence relationship between their instances. Intuitively, the first condition establishes that elements a' and b' trace back to a pair of elements that have a high-level correspondence relationship between them. The second condition means that the metaclasses ma' and mb' are the same metaclass or extensions of the same one. Therefore, it is permitted to define correspondence links between their instances and finally to compose them. If both conditions are satisfied for an element a' and b', T_3 will produce a correspondence link between a' and b'.

In order to fulfill the first condition, we need *traceability.* For instance, when T_1 is applied to the Attribute *dueDate*, it is transformed into the Attribute *dueDate*, the GetterMethod *getDueDate* and the SetterMethod *setDueDate*. To make this information available to T_3, we generate trace links between target elements and source elements. The same happens in the T_2 side, T_3 needs to know if the ReadMethod traces back to a related Resource. Once T_1 and T_2 are executed, two tracing models are generated (TM_A and TM_S); with these links, T_3 can find the elements in both lower-level models that trace back to the pair of related elements in both higher-level models.

To fulfill the second condition, the modeler has to define a Derivation Model (DM) to make explicit if the instances of two metaclasses can be related by a correspondence link. Furthermore, the modeler has to decide constraints stating if a couple of metaclasses can be composed. We have defined different types of constraints in the Derivation Metamodel. These types are: *Inheritable constraint* (to allow submetaclasss), *Final constraint* (to reject submetaclasses), *Incompatible constraint* (to explicitly reject two metaclasses), and *Composition constraint* (to allow composites). Due to space restrictions the details of the semantics of these constraints are out of the scope of this paper.

To generate the CM Transformation (T_3), the DM is processed by a High Order Transformation (HOT). This HOT analyzes the constraints in the DM and generates the CM transformation T_3. Therefore, it is not necessary to develop a new transformation for every pair of metamodels. The developer only requires defining the constraints between them.

The final step is the composition of both low-level models, which uses the generated $CM_{low-level}$. This CM model has the information of *what* will be composed. For instance, Classes in the application low-level model M_{java} will be composed with the annotated Classes in the security low-level model $M_{sec-java}$, the Attributes in M_{java} with the private Attributes in $M_{sec-java}$, and the Methods in M_{java} with the annotated methods in $M_{sec-java}$. By using the correspondence links it is possible to identify every pair of elements to be composed. To perform the composition we use a mechanism based on the *UML Package Merge* [3].

4 Conclusions

Our approach facilitates the modeling of multiple concerns in separated models each one close to the problem domain. The different concern models are aligned using a CM, which explicitly capture the overlapping and dependencies among their elements. Our approach offers an automatic derivation mechanism to maintain both models aligned from the high-level until the lowest level through the TC. This is one factor that differentiates our approach from others approaches where the correspondence relationships are only defined as an input, but not maintained during the TC. As a result of delaying the composition to the lowest level, where all the models conform to the same metamodel, it is possible to perform a homogeneous composition using a single composition mechanism.

Summarizing, our approach offers several advantages: 1) it facilitates the modeling of multiple concerns in separated models and close to the problem domain, 2) it offers an automatic derivation mechanism to identify the elements to compose in the low-level models based on relationships defined in the high-level, 3) it eases the use of a single composition mechanism at low-level of abstraction, 4) it reuses the existing assets (metamodels, models and transformations).

References

1. Tolvanen, J.P., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: Collected experiences. Software Product Lines (2005) 198-209
2. Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.F.: A canonical scheme for model composition. ECMDA-FA (2006) 346-360
3. Dingel, J., Diskin, Z., Zito, A.: Understanding and improving UML package merge. Software and Systems Modeling **7**(4) (2008) 443-467
4. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Systems Journal **45**(3) (2006) 515-526