Reuse and Configuration for Code Generating Architectural Refinement Transformations

Author version of the accepted manuscript. The final publication is available at ACM via http://doi.acm.org/10.1145/2489861.2489866.

Michael Langhammer[*]
Karlsruhe Institute of Technology, Karlsruhe, Germany
langhammer@kit.edu

Sebastian Lehrig[†]
Software Engineering Group & Heinz Nixdorf Institute, Paderborn, Germany
sebastian.lehrig@uni-paderborn.de

Max E. Kramer
Karlsruhe Institute of Technology, Karlsruhe, Germany
max.e.kramer@kit.edu

ABSTRACT

The transformation of component-based architectures into object-oriented source code for different platforms is a common task in Model-Driven Software Development. Reusing parts that are common to all supported target-platforms for several model-to-text transformations is challenging. Existing approaches, like parameterized transformations and modularity concepts for transformations, make the reuse of transformations parts easier, but cannot be used to visualize design decisions that are common to all supported target-platforms. In this paper, we propose that platform-independent design decisions and their transformation results should be made explicit in an intermediate view. A single parameterized transformation should yield a common object-oriented model before individual transformations for specific platforms are executed. We argue that the additional view makes it possible to analyze decisions on how a component architecture is implemented and increases the maintainability by decoupling the involved transformations.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: [Object-oriented design methods]; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

Keywords

Model-to-text transformation, Model Driven Software Development, View-based Engineering, Component-Based Software Engineering

[*]This work was supported by the German Federal Ministry of Economics and Technology (iZeus, grant 01 ME12013).
[†]The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

1. INTRODUCTION

In Model Driven Software Development (MDSD) the generation of object-oriented source code from a component-based Architecture Description Language (ADL) is a common task. These model-to-text transformations (M2T) usually produce method stubs that support developers in implementing the architecture. For platform-independent ADLs, it is possible to generate source code for multiple platforms. In addition, it is possible to generate different variations of source code for one platform. In such scenarios, however, a separate transformation is needed for each target platform and for each possible variation. Reusing transformation parts that are common for all transformations is an important, yet unresolved problem [13]. In this paper, we describe two problems that have to be solved when parts of transformations should be reused: (1) the *inter-platform problem* pointing to issues when transforming to distinct platforms, and (2) the *intra-platform problem* describing the challenges to support distinct features for one platform. We discuss two possible solutions for these problems. The first solution configures the transformation that generates the source code in terms of an annotation model that is used as additional input. The second solution introduces an object-oriented intermediate model that divides the code generation process into two steps. First, a platform-independent transformation from an architectural model to an intermediate model is executed. Second, a platform-specific transformation generates source code from the object-oriented intermediate model. The general idea of separating platform specifics from other models is also part of the Model-Driven Architecture (MDA) [10] approach that was specified by the Object Management Group (OMG). In Section 7 we discuss how the Platform Independent Model (PIM) of MDA relates to our solutions.

The contribution of this paper is a combined solution that represents the results of platform-independent design decisions in an intermediate model while keeping the transformations separated and configurable with annotation models. A platform-independent annotation model is used as input for a parametrised transformation that generates an object-oriented intermediate model from an architectural model. Then, a platform-specific annotation model and the intermediate model are used in a parametrised M2T transformation.

The remainder of this paper is structured as follows: in Section 2, a detailed explanation of the *inter-platform* and *intra-platform problem* is given. In Section 3 and 4, we discuss two possible solutions for these problems. Section 5 presents a solution that combines the previous two proposals. In Section 6 we illustrate an application example of our combined solution. Section 7 gives an overview about related work and Section 8 concludes the paper.

2. PROBLEM STATEMENT

Developers typically implement the architecture modeled in component-based ADLs. The automated generation of source code based on the ADL model via model transformations is a promising approach to lower the effort for this implementation task. For instance, a model transformation generates source code for an architectural component.

Source code is generally specific to a concrete platform like Java SE or Java EE. Consequently, transformation code also needs to be specific for a concrete platform. Therefore, scenarios that support multiple platforms commonly provide one transformation per platform. For instance, Palladio [2] provides a separate transformation of its *Palladio Component Model (PCM)* ADL to Java SE and Java EE, respectively. Due to their complete separation, these transformations cannot profit from reusing common transformation parts applicable for several platforms. For example, the transformation logic of first transforming components and connecting them afterwards is common to all platforms. This lack of reuse leads to redundant transformation code as well as an increased maintenance effort. A main problem for this is that common decisions, i.e., inter-platform commonalities, are hidden inside each transformation implementation. We call this problem the *inter-platform problem*.

We identified a similar problem when the target platform does *not* change, hence, called the *intra-platform problem*. For instance, when transforming an ADL model to Java SE, decisions related to the package and class structure for components or to implementation strategies for roles constitute variation points for a transformation. Roles, for example, can be implemented via the component context pattern of Stahl et al. [9] that decouples a component implementation from its roles or simply by letting one (component) class implement all roles. These variations can have a different impact on the flexibility and maintainability of the resulting code. Instead of making these variation points explicit and selectable during transformation, typically only one variation is selected and realized. This selection is again implicitly hidden inside transformation code. An example for this intra-platform problem is Palladio's Java SE transformation that only realizes roles according to the context pattern [9].

3. ANNOTATED REFINEMENT TRANSFORMATIONS

One technique to approach the inter- and intra-platform problem (cf. Section 2) is to use an annotation model for a transformation that refines an ADL to source code (see Figure 1a). The annotation model references elements of the ADL and configures how these elements are realized at the source code level. The transformation takes the annotation model in addition to the ADL model as input and provides the configured refinement as an output.

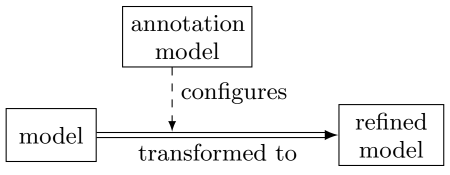
An annotation model targeting the inter-platform problem configures a transformation for a concrete target platform, e.g., Java EE. Therefore, a transformation that uses an ADL model and an annotation model as input is explicitly required to consider possible target platforms. Here, the configurability of the target platform is the central point, not the transformation implementation itself (black-box view). Typical types of annotation models targeting configurability are simple configuration files and feature models [4]. The actual implementation of the transformation (white-box view) is a complementing step. Common approaches to this implementation are visitor-based or based on simple if-then-else expressions to iterate over the annotation model (cf. [9]). Other approaches are based on Gamma et al.'s [5] template method pattern (e.g., Palladio follows this concept [1]) or higher-order transformations that generate a set of transformation rules for a configured platform (cf. [6]).

For the intra-platform problem, the situation is similar: configurability is again the central point. Therefore, the same techniques as for the inter-platform problem can be applied. Instead of allowing to configure the *type* of platform, such as Java EE or Java SE, concrete *features* within a fixed platform are configured. Example configuration options are how roles are implemented (cf. Section 2) and whether Remote Method Invocation (RMI) or the Simple Object Access Protocol (SOAP) is used for communication between components. It may be necessary to configure such features connector-wise instead of globally. Therefore, the annotation of single connectors is a viable option.

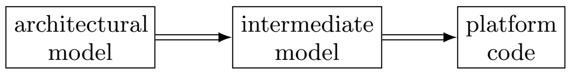
The main advantage of using annotation models is a clear separation of concerns: they solely add information needed for the transformation, i.e., they do not repeat information of the ADL that stays unchanged. Therefore, metamodel changes of the ADL do not affect the annotation model if no annotated metaclasses are deleted and if new metaclasses that should be annotated are added. A further consequence of this observation is that parts of the transformation can be realized as if no annotation model would be present. These parts are exactly the issues that are *not* covered by the annotation model, e.g., the idea of first mapping components to code and then adding the connectors. Due to this separation of concerns of configuration and realization, the model transformation becomes reusable and more maintainable.

The downside of annotation models is that transformations and analyzes requiring information from both an ADL model and an annotation model also need to navigate in both models. This necessity becomes crucial if existing transformations or analyzes that cannot cope with more than one input model are applied. Model checkers, for example, commonly operate only on one input model. Furthermore, navigating in both models can result in complicated transformations. If the transformation needs to access both models in most situations, repeatedly switching between the two models becomes a tedious task and results in a lower understandability and maintainability of the transformation. Such repeated switching is a sign that either the transformation itself lacks an appropriate modularization or that annotation models are not a proper solution for the given situation. These issues of navigating between different models (or viewpoints) relate to the viewpoint consistency of the considered scenario. Viewpoint consistency is commonly understood as the degree to which "different viewpoints impose contradicting requirements" [3] on the scenario. While Boiten et al. [3] consider viewpoint consistency in the context of Open Distributed Processing (ODP), we identified the importance of this characteristic when using annotation models for ADLs.

As alternative to annotation models, the information can be included in the ADL model itself or an intermediate model as described in the next section can be used.



(a) Solution with annotation refinement transformations



(b) Solution using an intermediate model for refinements

Figure 1: Comparison of the solutions to solve the *inter-* and *inter-platform problems*

4. USING AN INTERMEDIATE MODEL FOR REFINEMENTS

Another approach to solve the *inter-* and *intra-platform problems* is to use an intermediate model between the ADL and the target platform (see Figure 1b). The intermediate model contains all relevant platform-independent information and has to be generated by a transformation from the ADL. All transformations that generate platform-specific code have to use the intermediate model as source model. Hence, the inter-platform problem can be solved with an intermediate model. To solve the intra-platform problem, more than one transformation from the ADL to the intermediate model is needed. In fact, we need one transformation for each mapping that should be used. For example, we need one transformation to generate Java SE code and one for Java EE code.

The main advantage of using an intermediate model is that results of platform-independent design decisions become explicit and persistent. This has the advantages that the intermediate model can be used in analyzes and platform-independent mapping rules are not hidden in a transformation. In addition, the intermediate model can be shown in a separate view. This view can be generated from the intermediate model and helps developers, e.g., by analyzing how ADL components are connected to object-oriented source code. In addition, the view supports developers by discovering errors occurred during the transformation from the intermediate model to the source code. Developers can analyze the intermediate model and decide whether the error occurred in the transformation from the ADL to the intermediate model or during the transformation from the intermediate model to source code. Another advantage is that transformations that produce the intermediate model are completely decoupled from transformations using the intermediate models to generate platform-specific code. This increases the modularity and, therefore, maintainability of the software system.
One disadvantage of having an intermediate model is that information that have to be present in the intermediate model as well as in the ADL has to be copied in the transformation from the ADL to the intermediate model. Another disadvantage is that the intermediate model has to be expressive enough to represent all possible variations resulting from different design decisions.
The difference of our proposed intermediate model compared to the PIM and Platform Specific Model proposed by the OMG [10] is that our object-oriented intermediate model is not completely platform-independent since it is restricted to object-oriented platforms. However, it does not contain further platform-specifics. Hence, it can be considered between a PIM and PSM.

5. COMBINED SOLUTION

Based on our experience with transformations for multiple platforms and the advantages and disadvantages discussed in the previous sections we propose a combined solution. This combined solution separates those transformation parts that are affected by the inter-platform problem from those parts that are affected by the intra-platform problem. We suggest that platform-independent design decisions are added to an ADL model using an annotation model. This annotation model and the ADL model serve as input for a parameterized transformation that produces an instance of an object-oriented intermediate model. Platform-specific design decisions are added to this intermediate model in a second annotation step. Another parameterized transformation consumes this platform-specific annotation model and the intermediate model to generate platform-specific code. The domain and annotation models as well as transformations of our combined solutions, are also depicted in Figure 2.

Our solution combines advantages of both approaches: platform-independent design decisions and the knowledge how to map them to object-oriented code can be reused for multiple platforms in the form of the intermediate model. The result of these decisions and their mapping can be specifically presented in an additional view that resides on an abstraction level between component-based architectures and object-oriented code. With this view, users can analyze and modify the platform-independent implementation of their architecture. Additionally, the configurability of the transformation that leads to this intermediate view is made explicit. This helps in separating transformation parts that pertain to different concerns and, therefore, increases the maintainability of the transformations.

Our combination of annotation models and an intermediate model also avoids some of their disadvantages: the intermediate model can be populated using a single-structured transformation instead of multiple transformations that encapsulate platform-independent design decisions implicitly. This is possible because these decisions are explicitly used as an input for the transformation in the form of an annotation model. The results of platform-independent design decisions are directly available in the intermediate model. This makes it unnecessary to indirectly access these decisions from platform-dependent transformation logic as it would be the case without an intermediate model. Furthermore, the second platform-specific transformation does not use the platform-independent design decisions as an additional input. It only consumes the corresponding results. As a result, platform-independent and platform-specific transformation parts, which would be part of the same transformation if no intermediate view would be provided, are completely decoupled from each other.

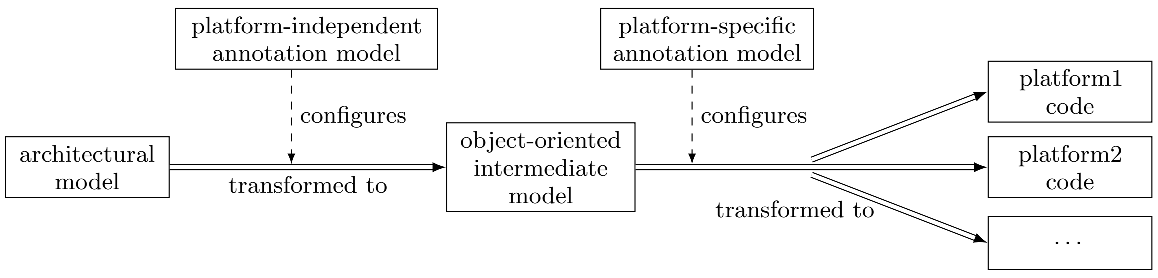

platform-independent annotation model
platform-specific annotation model
configures
configures
architectural model
transformed to
object-oriented intermediate model
transformed to
platform1 code
platform2 code
. . .

Figure 2: Models, annotation models, and transformations for our combined solution

6. EXAMPLE

In this section we illustrate the combined solution we explained in Section 5 with a concrete transformation from an ADL to an intermediate model. In our example we use the architectural relevant parts of the PCM as an ADL and a UML class diagram as intermediate model. The architectural model of our example represents an extract of a simplified version of the Media Store case study for Palladio [7], which enables users to upload and download media files from a server. The excerpt from the repository for this Media Store system contains two components and two interfaces and is shown in Figure 3. A user of the Media Store system can use the *HTTPDownload* and *HTTPUpload* methods of the *IHTTP* interface to download and upload media files. To realize this functionality the *WebGUI* component uses the *MediaStore* component via the *IMediaStore* interface.

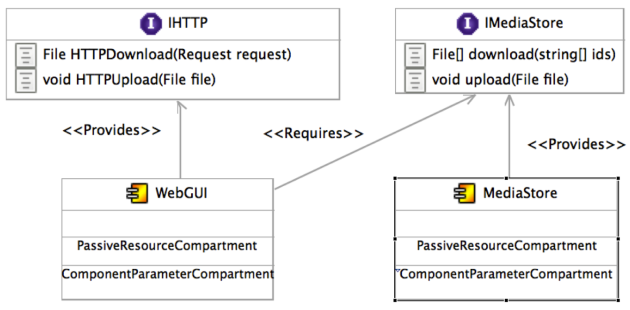

IHTTP
File HTTPDownload(Request request)
void HTTPUpload(File file)
IMediaStore
File[] download(string[] ids)
void upload(File file)
<<Provides>>
<<Requires>>
<<Provides>>
WebGUI
PassiveResourceCompartment
ComponentParameterCompartment
MediaStore
PassiveResourceCompartment
ComponentParameterCompartment

Figure 3: An excerpt from the architectural model of a simplified Media Store, with two components *WebGUI* and *MediaStore* and the interfaces *IHTTP* and *IMediaStore*.

The application of the combined solution that we presented before yields two annotated transformations. The first transformation transforms a PCM instance into an UML class diagram. For this first transformation, we have to provide annotations that describe how ADL components are mapped to UML elements. For instance, it has to be decided how provided roles of components are mapped to UML classes and interfaces. One alternative is to use the component context pattern discussed by Stahl et al. [9], which we mentioned in Section 2. Another alternative is to make the classes that represent components directly implement the interfaces. A UML class diagram representing the intermediate model resulting from the second alternative is depicted in Figure 4. It uses the following mapping rules: a PCM repository is transformed into a UML package and each PCM component is transformed into a subpackage in this repository package and a class. PCM interfaces and their signatures are transformed into UML interfaces and operations. Required roles are represented by a property in the class for the component and by a dependency from this class to the interface. Provided roles lead to implementation relationships between the corresponding class and interface. This mapping for provided and required roles can be used instead of the component context pattern whenever connectors between component instances do not need to be changed at runtime and if no component provides the same interface more than once.

The second transformation from UML to Java uses the UML model as source. It takes the target platform of the resulting source code, e.g. Java SE or Java EE, into account. The intermediate model, however, does not need to be changed when the target model is changed.

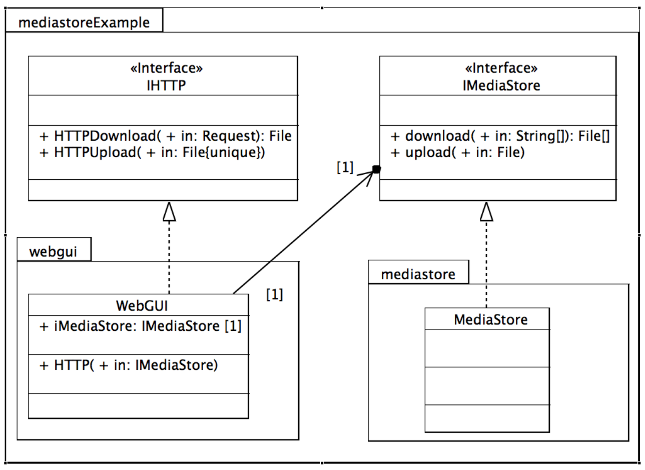

mediastoreExample
«Interface»
IHTTP
+ HTTPDownload(+ in: Request): File
+ HTTPUpload(+ in: File{unique})
«Interface»
IMediaStore
+ download(+ in: String[]): File[]
+ upload(+ in: File)
[1]
webgui
WebGUI
+ iMediaStore: IMediaStore [1]
+ HTTP(+ in: IMediaStore)
[1]
mediastore
MediaStore

Figure 4: A UML class diagram for the simple media store that represents the intermediate model between PCM and source code.

7. RELATED WORK

In this section, we provide a short list of approaches to reuse and modularity in model transformations and state their main deficiency.

Wimmer et al. [13] provide a comparison framework for reuse in rule-based transformation languages. The superimposition mechanism they discuss replaces complete rules, which makes it difficult to use it for fine-grained design decisions. Genericity in the sense of type-parameters is not available for standard transformation languages and only applicable for some reuse issues. A transformation DSL would add artificial complexity and is inappropriate in our scenario because the transformation targets and transformation logic are far from stable and therefore would result in a series of DSL changes. Transformation product lines correspond to the annotation mechanism discussed in Section 3.

Sánchez Cuadrado et al. [8] factorize and compose transformation definitions organized in phases with the transformation language RubyTL. Voelter and Groher [11] achieve variants of code-generators with aspects for the M2M transformation language Xtend and the M2T transformation language Xpand. Wagelaar and Straeten [12] compare different variants for adding refinement information using the Web Ontology Language Description Logic, DSLs, and feature models but do not consider how this information is used.

All these reuse and modularization solutions for transformations do not help in making design decisions visible and do not allow developers to analyze the result of platform-independent design decisions. Even if platform-independent design decisions can be reused using one of the approaches, they cannot be realized independent of the platform-specific parts without an intermediate step.

As mentioned in Section 1, the MDA specification also mentions the generation of source code from models. In the MDA approach, a Computation Independent Model (CIM) is created by domain experts to model the system. This CIM is transformed in a Platform Independent Model (PIM). This PIM is transformed into a Platform Specific Model (PSM). Finally, the PSM is used as an input model for a M2T transformation that generates source code. The MDA specification also mentions the idea of marking a model for transformations from a PIM to a PSM. But in contrast to the solutions that we discussed in this paper, the MDA neither describes how this information is used to guide transformations nor how it can be reused for multiple platforms. Another difference between the mark models proposed by the MDA approach and our combined solution is that we differentiate between platform-specific and platform-independent annotation models. The last difference concerns the interpretation of the term platform-independent. Although ADL models can be seen as platform-independent in the MDA sense, the object-oriented intermediate model can be considered between a PIM and a PSM, as it is restricted to object-oriented platforms but does not contain further platform specifics.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed approaches to reuse transformation code when object-oriented source code is generated from a component-based Architecture Description Language (ADL). We identified the *intra-platform* and the *inter-platform problem* that have to be solved when transformation code should be reused. To solve these problems, we have discussed the ideas of having an annotation model or an object-oriented intermediate model. If we consider an annotation model for solving the problems, the annotation model is used as an additional input model for the model-to-text transformation. If an intermediate model is used, the transformation from the ADL to source code becomes split-up in two transformations: one that transforms from the ADL to the intermediate model and one that executes the model-to-text transformation from the intermediate model to source code. During our discussion, we came to the conclusion that we need both: annotation models and an intermediate model. Hence, we presented our combined solution that uses an intermediate model to solve the *inter-platform problem* and a platform-independent annotation model to solve the *intra-platform problem*. Two transformations are needed: firstly, a transformation that uses a platform-independent annotation model and an ADL model as input and the intermediate model as output, and, secondly, another transformation that uses a platform-specific annotation model and the intermediate model as input to produce source code. The combined solution combines the advantages of the annotation model with those of the intermediate model, e.g., an explicit view to the intermediate model can be generated, and avoids some of their disadvantages.
As future work we plan a prototypic implementation of the combined solution using Palladio as ADL that should be transformed to Java EE and Java SE.

References

[1] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Vol. 1. Universitätsverlag Karlsruhe, 2008.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (Jan. 2009), pp. 3–22.

[3] Eerke Boiten et al. “Viewpoint consistency in ODP”. In: *Computer Networks* 34.3 (2000), pp. 503–537.

[4] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. “Formalizing Cardinality-based Feature Models and their Specialization”. In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29.

[5] Erich Gamma et al. *Design Patterns – Elements of Reusable Object-Oriented Software*. 1st ed. 37. Reprint (2009). Amsterdam: Wesley, 1995.

[6] Jens Happe et al. *A Pattern-Based Performance Completion for Message-Oriented Middleware*. 2008.

[7] Heiko Koziolek, Steffen Becker, and Jens Happe. *Predicting the performance of component-based software architectures with different usage profiles*. 2007.

[8] Jesús Sánchez Cuadrado and Jesús García Molina. *Approaches for Model Transformation Reuse: Factorization and Composition*. 2008.

[9] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[10] Frank Truyen. *The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture*. 2006.

[11] Markus Voelter and Iris Groher. *Handling variability in model transformations and generators*. 2007.

[12] Dennis Wagelaar and Ragnhild Straeten. “A Comparison of Configuration Techniques for Model Transformations”. In: *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, 2006.

[13] Manuel Wimmer et al. “Fact or Fiction – Reuse in Rule-Based Model-to-Model Transformation Languages”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2012.