Tool Independent Code Generation for the UML
Closing the Gap Between Proprietary Models and the Standardized UML Model

Arne Noyer[1], Padma Iyenghar[2], Elke Pulvermueller[3], Florian Pramme[1], Joachim Engelhardt[1], Benjamin Samson[2] and Gert Bikker[1]

[1]*Institute for Distributed Systems, Ostfalia University, Salzdahlumer Strasse 46/48, Wolfenbuettel, Germany*

[2]*Institute of Computer Engineering, University of Applied Sciences Osnabrueck, Osnabrueck, Germany*

[3]*Institute for Software Engineering, University of Osnabrueck, Osnabrueck, Germany*

Keywords: Model-driven Software Engineering, Code Generators, Model Transformation, Meta-models, Unified Modeling Language (UML), Eclipse Modeling Framework (EMF), Model Facade.

Abstract: Embedded software development is moving towards the model-based paradigm to support the complexity of today's embedded systems, as they become more and more important and omnipresent in our daily lives. In this context, the Unified Modeling Language (UML) is a widely used standard. Code generators can be executed to generate source code from UML models. Usually the code generators are proprietary for one UML tool. If code generators for different targets or programming languages have to be supported by various modeling tools, the wheel must be reinvented. Code generators could use the standardized Extensible Markup Language Metadata Interchange (XMI) format of the UML as a basis. However, tools export their data to XMI differently. Therefore, the paper shows how the proprietary models of UML tools can be mapped to a standardized UML model. This is realized by using techniques for model to model transformations. These techniques need a meta-model for the source and the target model. Hence, an approach is introduced for creating meta-models for Application Programming Interfaces (APIs) of UML-tools, which act as a facade. Then the code generators can work with the standardized UML model to generate the source code. This results in an improved scalability of the code generators.

1 INTRODUCTION

Nowadays embedded systems are omnipresent and increasingly used in a wide variety of application scenarios. The number of functionalities they have to provide and the number of systems they interact with continue to grow. For instance, automobiles have more and more comfort features that are executed on different Electronic Control Units (ECUs). For example, in luxury vehicles there are already up to 100 control devices in use (Hergenhan and Heiser, 2008).

In embedded software engineering, classically, the software is implemented in a 3rd generation language (3GL), such as ANSI C/C++. As the embedded systems are developed with additional features, the complexity of the underlying software increases, necessitating new and automated ways of software development, such as model-based approaches. In the Automotive Safety Norm ISO 26262 the usage of model based approaches is highly recommended.

The Unified Modeling Language (UML) (Object Management Group, 2013d) is one among the widely used industry standards for Model Driven Development (MDD) (France et al., 2006). Apart from the general UML elements and diagrams, UML profiles can be used to address certain application areas such as the real-time embedded systems (Krichen et al., 2013).

MDD is considered as a key solution for structured embedded software engineering and automation. Thereby it is inevitable for future projects.

The hardware in the embedded system has often only limited resources (e.g. memory) (Sestoft et al., 2002) which is imperative for its efficient operation and cost optimization. This results in stringent and special requirements during embedded software development. Therefore, code generators, which create code from UML models, have to be highly efficient. The development of a code generator comes with a lot of effort and the generated code may be specific

Noyer A., Iyenghar P., Pulvermueller E., Pramme F., Engelhardt J., Samson B. and Bikker G..
Tool Independent Code Generation for the UML - Closing the Gap Between Proprietary Models and the Standardized UML Model.
DOI: 10.5220/0004870701170125
In *Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2014), pages 117-125
ISBN: 978-989-758-030-7
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

for a certain target platform. For supporting different platforms, many code generators have to be realized or adaptions have to be made.

Unfortunately, the code generators are in general proprietary and specific for a given modeling tool. This paper addresses the aforementioned gaps and outlines an approach to create tool independent code generators for the UML.

In this context, an approach for creating model facades for Application Programing Interfaces (APIs) is introduced. The remaining of this paper is organized as follows. Section 2 presents state of the art techniques for code generation and coupling of different modeling domains. It also outlines related work. Section 3 presents an approach for tool independent, UML-based code generation. A prototype implementation of the proposed approach is presented in section 4. Section 5 concludes the paper.

2 STATE OF THE ART AND RELATED WORK

This section discusses the state of the art and related work pertaining to model based development, code generation in MDD from UML tools, meta-models for UML, model to model transformation, meta-model generation for APIs and model to text transformation. Since code generators are basically model to text transformations, different techniques for model to text transformations are discussed. The input for tool independent code generators has to be a uniform model. Therefore, standardized UML meta-models are analyzed. In order to transform the proprietary model of modeling tools into a standardized UML model, a model to model transformation mechanism is needed. However, for using many of these mechanisms, a meta-model is needed not only for the target model, but also for the source model. Since the data of modeling tools can often be accessed by using an Application Programing Interface (API), possibilities for creating a meta-model for an API are presented. This section concludes with a short summary, which also includes the related work pertaining to the state of the art techniques presented so far in this section. Last but not the least, the gaps in the existing methodologies pertaining to the main goals of this paper are enlisted.

2.1 Code Generation from UML Tools

The huge advantage of the UML can be fully realized if the source code is generated directly from the models (Burke and Sweany, 2007). Therefore, many modeling tools such as Rational Rhapsody (IBM, 2012) and Enterprise Architect (Sparx Systems, 2012) have code generators that generate the source code automatically from the models.

However, many producers of modeling tools only focus on code generators for a selection of applications and target platforms. For example, there are code generators for large systems, such as Windows CE, Linux and VxWorks, in Rhapsody (IBM, 2012). For many potential users with special application areas, e.g. VHDL for embedded systems (Moreira et al., 2010), this is not sufficient. Also for specialized systems with limited resources, the code generators that are included in the modeling tools often can not be used. Therefore, companies like (Willert Software Tools GmbH, 2013) are offering frameworks for different target platforms, and developed their own code generators for modeling tools that use these frameworks. Such development of new code generators for certain fields of application, e.g. MISRA compliant C (Motor Industry Software Reliability Association (MISRA), 2012) for a Cortex M3 (ARM, 2013), and their integration in a specific modeling tool results in significant effort. Furthermore, the code generators are typically proprietary to a modeling tool since they access the model from the tool by using its interfaces. Nevertheless, code generators are working very similar if they have the same field of application, e.g. the same target language. They differ primarily in how they read the model.

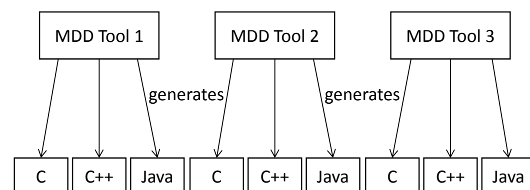

Figure 1: Similar code generators for different modeling tools.

For supporting a specific kind of code generation in different modeling tools, (at least) one proprietary code generator must be developed for each modeling tool (Fig. 1). These are very similar, as code shall be generated in the same way. They differ mostly only in reading the data (models) from the tools. So, a significant effort and time are needed for developing different code generators that also have many redundancies. For example, if three code generators for the languages C, C++ and Java should be developed and these should be compatible with three different modeling tools, a total of nine code generators would be needed (Fig. 1). This also results in the risk of errors such that something is not correctly implemented in a new code generator, which already worked in

another code generator (redundancy issues). Also, changes in the application field of similar code generators would cause the need for adapting those changes to all of these similar code generators (e.g. for considering MISRA-rules). The same is of course also applicable if different code generators for embedded systems are required that support various target platforms. In contrast, the approach in this paper allows to use one specific code generator with different modeling tools. Therefore, a uniform meta-model is needed with which such a code generator can work.

2.2 Meta-models for UML

The Meta Object Facility (MOF) (Object Management Group, 2011) is a standard by the OMG (Object Management Group, 2013b) for defining meta-models. It is also used for defining the UML. Furthermore, the OMG provides an XMI format for exporting/importing UML models (Object Management Group, 2013d). In theory, this format can be used to exchange UML models between different UML tools. Unfortunately, different UML tools export the same model to XMI differently. For example, Rhapsody exports the reference from an UML attribute to a datatype as a XMI attribute, while Enterprise Architect creates a XMI child element, which contains the reference. For the realization of tool independent UML code generators, this also makes it very hard to just export the model as XMI file from UML tools and then using the XMI file as input for a code generator. The need for a transformation from models of UML tools into a generic model still exists.

There is an implementation of the UML2 meta-model for Eclipse (Eclipse Foundation, 2013e), which is based on the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2013b). The EMF is basically an implementation of the reduced Meta Object Facility (MOF), the essential Meta Object Facility (eMOF). It is widely used for creating meta-models and there are many frameworks with a variety of functions, which can be used in combination with EMF-based meta-models. Benefits of the EMF itself are for example the easy creation of meta-models, the Java code generation from the meta-models as implementation and the possibility to generate a graphical tree editor for working with instances of the meta-model. They are discussed in detail in (Bzivin et al., 2005). In summary, there already are implementations of the standardized UML meta-model, which could be used as input for code generation. The question emerges, how a transformation from different UML tools to a standardized model can be realized. There are already many mechanisms for model to model transformations. Concepts for model to model transformations are analyzed below.

2.3 Model to Model Transformation

In the literature, there are different techniques available for model to model transformations. Significant concepts are Triple Graph Grammars (TGG) (Kindler and Wagner, 2007), Query View Transformation (QVT) (Object Management Group, 2013c) and the Atlas Transformation Language (ATL) (Jouault et al., 2006). It is noticeable that many implementations of these techniques are based on Eclipse and use the EMF. In order to execute a model to model transformation with such an implementation, for both sides an EMF-based meta-model is needed just as there is one for the UML2 meta-model. Different Triple Graph Grammars (TGG) are compared with each other in (Hildebrandt et al., 2013). These techniques allow a model to model transformation to the EMF-based UML2 model, if there is an EMF-based meta-model for the source model. Unfortunately, most UML tools do not provide a (EMF-based) meta-model for their internal model. Many tools, such as IBM Rational Rhapsody (IBM, 2012) and Enterprise Architect (Sparx Systems, 2012), allow to access their internal data model by providing an API, but in most cases there is also no meta-model for their API.

One of the major aims of this paper is to address this gap and propose an approach towards automatic generation of an EMF-based meta model. This acts as a facade for accessing an API of a modeling tool. This means that model elements are like a view on the objects of the API. If operations of model elements are called, the call is forwarded to the representing operations of API objects (see section 4.1).

2.4 Meta-model Generation for APIs

An approach for generating a meta-model for an existing API is API2MoL (Izquierdo et al., 2012). For generating a meta-model the existing source code of an API and its structure must be analyzed. For each class inside the API a corresponding element in the meta-model is created. Moreover, the approach aims at creating bridges between model driven engineering and APIs automatically. Therefore, two operations are needed, after the meta-model is generated. One operation is to obtain model elements from the objects of the API. The other operation is to create API objects from model elements. API2MoL provides operations for both sides, which can be executed when such a conversion is needed. There is an EMF-based implementation of API2MoL, which is available at

(Atlanmod and Modelum research groups, 2013).

API2MoL could be used for converting the internal data of an API into a model. Then a model to model transformation could be executed for converting the API model into a standardized UML2 model. The transformation from UML2 model elements to API model elements can be useful for realizing reverse engineering functionalities. If API objects and their model representations with their attributes are stored on both sides, synchronization problems may occur. This can happen if API objects and their model representations are both changed before the changes are applied to the other side.

In order to avoid synchronization problems, another approach for creating a meta-model facade for an API is presented in chapter 4 of this paper. Furthermore, that approach allows the seamless integration with other technologies, which need an EMF based model. These include technologies for model to model transformations (see section 2.3). As soon as models of modeling tools can be accessed as a standardized UML2 model, it should be enabled to generate source code from it. Therefore, the next section briefly discusses approaches for model to text transformations.

2.5 Model to Text Transformation

Code generators are basically model to text transformations. They can be implemented by programmatically concatenating strings or by using one of many template based approaches. The clarity of the source code of huge code generators can suffer, if they are implemented in an ordinary programming language. Template based approaches make the programming easier and improve the maintainability. In (Rose et al., 2012) different concepts for model to text transformations are discussed in detail. There are also many frameworks for model to text transformations for Eclipse, such as Acceleo (Eclipse Foundation, 2013a), Xpand (Eclipse Foundation, 2013f) and Xtend (Eclipse Foundation, 2013g). Acceleo is an implementation of the standardized MOFM2T (Object Management Group, 2013a). Furthermore, there also exist code generators for Eclipse, which generate code from EMF-based UML2 models, such as (Obeo, 2013), which is based on Acceleo. There are many frameworks for developing code generators and there are already existing code generators, which can be used for an EMF-based UML2 model.

2.6 Summary

Most modeling tools provide their own code generators, which can only generate code from their own models. Besides the tool manufacturers there are also other companies (like (Willert Software Tools GmbH, 2013)) that develop code generators. In most cases, the code generators in general are developed in a similar way to those of the manufacturers by using tool specific mechanisms. If one type of code generation is realized for different modeling tools, they develop a proprietary code generator for each of them. This results in a lot of effort for developing several similar code generators. In order to improve this, an approach for the development of tool independent code generators for UML is introduced and discussed in this paper.

There is already a standardized XMI format for UML, in which most UML tools can export their models. Unfortunately, there are differences in how they export their data, which makes it difficult to just use it as input for code generators. There is a gap between the proprietary models of different UML tools and the standardized UML model.

A technique for model to model transformation could be used for converting the proprietary models of UML tools into a standardized UML model. However, most techniques need a meta-model for the source and a meta-model for the target in order to allow a mapping between two models. Most modeling tools do not provide such a meta-model.

Many UML tools allow to access their models by using an API. In order to allow the usage of model to model transformation techniques on an API, a novel approach of creating a model facade for accessing an API is introduced in this paper. After converting a model of a tool into a standardized UML tool by using a model to model transformation, one of many techniques for code generation can be used.

3 PROPOSED APPROACH

This section presents the approach for tool independent code generation for UML. In the first subsection, the general approach is discussed. Afterwards, its scalability is analyzed. Subsection 3.3 discusses different approaches for accessing the models of modeling tools and converting them into a standardized model. In this context, a new approach is introduced for creating a model facade for APIs of modeling tools. The next subsection introduces a prototype implementation.

3.1 Tool Independent Code Generation for the UML

A tool independent code generation can be realized by first converting the UML-tool-specific model into a standardized UML model. Therefore, a model to model transformation can be used. Afterwards code generators can generate code from a standardized model (cf. Fig. 2).

When a code generator is developed that works with the standardized UML2 model, it is automatically compatible with all tools for which a model transformation to the generator model has been created. In other words, of course, also all code generators are compatible with a tool for which a model transformation is defined. This approach is illustrated in Fig. 2. Each of the three modeling tools has a model transformation in order to transform its data into the standardized model. Then the real code generators are used to generate the source code from this model. In case another type of code generation is needed (for example for generating code in C#) only one code generator needs to be implemented that works with the standardized UML2 model. It is automatically compatible with each of the three modeling tools.

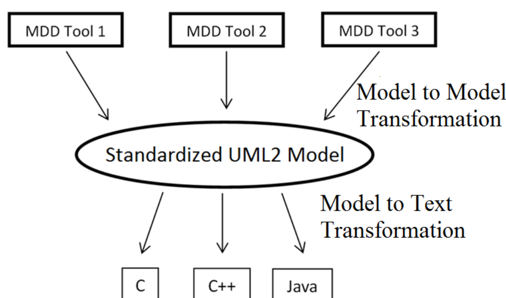

Figure 2: Code generation with a standardized UML model.

3.2 Improved Scalability

Let us assume that a transformation of a model from a modeling tool into a standardized model is also some kind of code generation, since there is also an input and an output. In this case the difference is that the transformation results in a model instead of code. In the example shown in Fig. 2, six code generators are developed in total. In contrast, in Fig. 1 nine code generators have to be developed for allowing the same types of code generation for the same amount of modeling tools.

Let C be the sum of all code generators (and model transformations), which have to be developed, T the number of tools that should be supported and G the number of types of code generations (e.g. Java, C, C++) that should be realized. Then, when tool specific code generators are developed, the calculation is $C_{old} = T * G$.

When working with a standardized UML model in the middle, let G_M and G_T further be the number of model to model transformations and the number of model to text transformations, which are needed. In this approach, $C_{new} = G_M + G_T$. Since for each modeling tool a model to model transformation G_M must be realized, it can also be expressed as $C_{new} = T + G_T$. This is because, each model to text transformation G_T realizes a different type of code generation, $C_{new} = T + G$.

If code generators are developed which should be compatible with only one UML tool ($T = 1$) it is always $C_{old} < C_{new}$. This means that less transformations have to be realized if the code generator is just proprietary for a certain tool. If at least two code generators are developed for at least two different tools, it starts to get complex. In case $T = 2$ and $G = 2$, the number of transformations that have to be realized is in both cases four. If T and/or G are greater than 2, $C_{old} > C_{new}$ is always true. Since for calculating $Cnew$ the sum of T and G is used and for C_{old} their product is used, the amount of required transformations rises much faster for C_{old}. As an example Fig. 3 shows the scalability for $T = 3$.

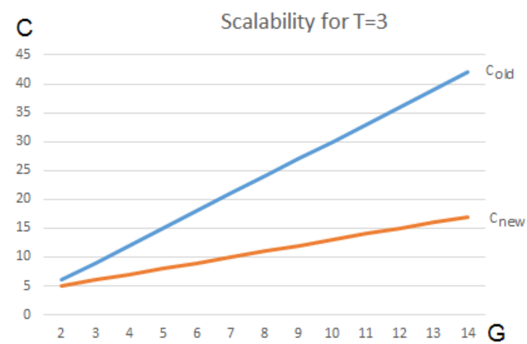

Figure 3: Scalability for T=3

Of course, the amount of effort for realizing model to model transformations for different tools and for realizing different types of code generators by using model to text transformation techniques varies. It strongly depends on which kind of technique is used and aspects such as how similar the model of a tool and the standardized UML model are. In the examples discussed above, it is assumed that the effort for developing a model to model transformation for a specific tool is the same as the effort to develop a code generator. This can assumed to be the worst case, since a model to model transformation could also be realized as model to text transformation, if a model

can be serialized into some kind of text. In reality, mapping a model, which is similar to the UML model, to a standardized UML model by using a suitable mapping technique results in many cases in less effort. The question arises, how the data (the models) of modeling tools can be accessed and converted to the standardized model in a suitable way. This is discussed in the next section.

3.3 Converting Proprietary Models into a Standardized Model

As stated in section 2.2, it is not an efficient to just use the XMI export of available UML tools to get a standardized input for code generators. Of course, the XMI files could be analyzed and further transformed into a standardized model. Then there would be a further step before the actual code generation could take place, in which data has to be written on a hard drive. This would result in more time for running code generators. Because of this, it is desirable to access the data of modeling tools directly.

Many UML tools allow to access their internal models by providing an API (see section 2.3). Unfortunately, in most cases they do not provide a meta-model for their API, which would allow the usage of many model to model transformation techniques. However, a meta-model can be extracted from existing source code and/or compiled units by analyzing their structure. Therefore, for each class a meta-class has to be created in the meta-model with its attributes and operations. Afterwards, operations could be implement for creating model elements (of this meta-model) for objects of the API and vice-versa. In order to avoid synchronization problems, another approach is introduced in the following.

The model elements have to refer to the API objects, which they represent. This is realized by storing a associated API object in an attribute for the model element. For example, *Model_APIObject* could be the model representation of an API object *APIObject*. Then an attribute is needed for *Model_APIObject* to store a reference to the corresponding API object. This attribute could be named *originalAPI_Object*.

Now, every call of an operation of the model element can just be delegated to the equivalent operation of the contained API object. In this context, only public operations have to be considered and created in the meta-model, since other operations of an API cannot be accessed. There is also nothing else to do for private attributes and for attributes for which getter and setter operations exist. If there is a public attribute inside an object of the API for which no getter and setter operations exist getter and setter operations should nevertheless be created for the representing model elements. They should, again, delegate the access to the attribute. So, it is not needed to store attribute values inside model elements. The access of all attributes is also just delegated to the original API objects. Section 4 elaborates on this methodology and discusses a prototype implementation. When this model facade is realized for an API, many existing model to model transformations can be used for converting the internal model of a tool into a standardized UML model.

In some cases, the internal models of tools, such as Merapi-Modeling (Ostfalia University, 2013) and the open source tool Papyrus (Eclipse Foundation, 2013d) can be accessed directly and there is also a meta-model available for it. Here, model to model transformations can be used directly.

If a modeling tool does not provide a mechanism (e.g. an API) to access their model elements from the outside, one has to achieve this goal in a laborious way. In any case, modeling tools somehow store their models. This can be inside a database or inside files. In order to understand the structure of the stored data, a large amount of reverse engineering has to be performed. Then, a meta-model can be created manually, in which the operations of model elements access the content of the database or the files. This was never necessary for the implementation of a prototype, in which four modeling tools were considered.

4 PROTOTYPE IMPLEMENTATION

A code generation framework for the proposed approach is implemented as a prototype using Eclipse technologies. As mentioned in section 2.2, there already is an implementation of the standardized UML2 model in Eclipse based on the Eclipse Modeling Framework (EMF). This is used as model in the middle and input for code generators. Furthermore, EMF allows to create other meta-models. Based on EMF there are well established frameworks for model to model and model to text transformations. For the prototype, the UML tools IBM Rational Rhapsody (IBM, 2012), Enterprise Architect (Sparx Systems, 2012), Merapi-Modeling (Ostfalia University, 2013) and Papyrus (Eclipse Foundation, 2013d) are considered.

4.1 Standardized UML2 Model for UML-tools

Rhapsody and Enterprise Architect both provide an Java API for external programs to access their inter-

nal model elements. For creating a EMF-based meta-model of the APIs it can be analyzed by using the Java Reflection API (cf. Fig. 4). For each class, a class in the meta-model (called EClass in EMF) is created with an attribute referring to the original API object. Furthermore, representations of its public operations are created for the class in the meta-model. The implementation of these operations is done in a way that they always delegate the call to the operation of the referred, original API object. This is realized by creating an EMF-based annotation for the operation inside the meta-model that contains code for delegating the calls. Afterwards, an EMF mechanism is used for creating Java code for the meta-model. This code is executed, whenever model elements of this meta-model are used. A prototype is implemented for automatically creating model facades in this way. The facade generation can also be used for any other Java API.

There is a difference between this approach and API2MoL (Izquierdo et al., 2012). API2MoL provides operations, which can be manually executed, to convert the API objects into model elements and vice versa. In the proposed approach, the model acts as a facade for an API, i.e. a corresponding model is just used as a layer for accessing the API. This also allows a seamless integration for APIs with other EMF-based frameworks, such as the Graphical Modeling Framework (Eclipse Foundation, 2013c). There is no need for executing operations for the conversion. Therefore, no synchronization problems can occur. Model representations of API object are created automatically on demand.

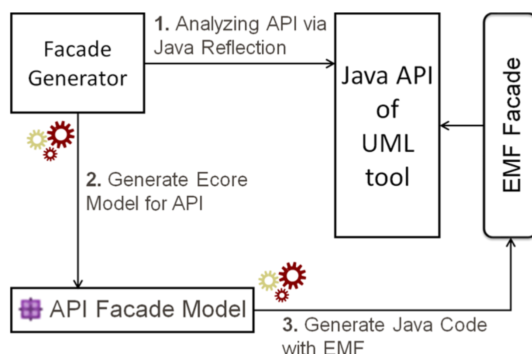

Figure 4: EMF facade generation.

The access of operations, which return API objects, have to return model representations of these when working on the model level. Every time such an operation is called on the model level, a model representation of the API object is created and its attribute for containing the original API object is set. In order to avoid that several model representations of a single API object are created, the API object is stored together with its model representation inside a Map data structure.

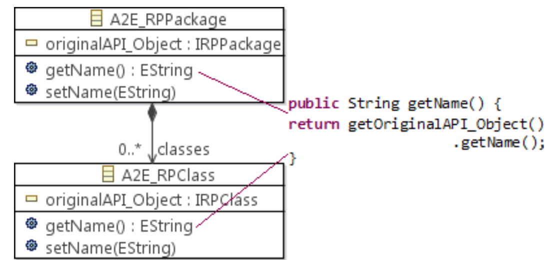

Figure 5: EMF facade for IBM Rational Rhapsody.

As an example Fig. 5 shows a small part of an EMF facade for the Rhapsody API. There are two EMF classes, which act as a facade for packages and classes from Rhapsody. Their name has the prefix *A2E*, which stands for *API to Ecore*. Both of them have an attribute *originalAPI_Object* for storing a reference to the object, which they represent. Furthermore, the EMF classes have an operation for each method of the *originalAPI_Object*. If such an operation is called, it just forwards the call to the represented operation of the *originalAPI_Object*. The same applies for accessing and setting attributes via getter and setter methods, i.e. the attribute *name*.

An aggregation was created from the EMF class *A2E_RPPackage* to *A2E_RPClass*, because the Rhapsody API class *IRPPackage* has an operation for accessing contained API objects of type *IRPClass*. When the code is executed for accessing the contained classes of the EMF facade class *A2E_RPPackage*, at first, the contained classes of the *originalAPI_Object* are resolved. Then it is checked, if already objects of type *A2E_RPClass* were created, which represent theses classes. If necessary, new facade objects will be created and their *originalAPI_object* will be set. Finally, the facade objects of type *A2E_RPClass* will be returned.

Now a model to model transformation can be used for converting API models into the EMF-based UML2 model. Any technique from section 2.3 can be used for this purpose. However, to have the possibility of implementing reverse engineering functionalities later, the technique should allow to make bidirectional mappings and transformations. QVT would be adequate, but there is a lack of a suitable implementation which supports QVT-Relations for bidirectional transformations completely (Macedo and Cunha, 2013). An interesting upcoming implementation is presented in (Macedo and Cunha, 2013). However, it is only recommended for medium size models for now, because of its execution performance. TGGs are also very well suited for our purpose. It was

shown in (Rose et al., 2012) that the TGG implementation eMoflon has the most advanced functionalities for bidirectional transformations. Therefore, eMoflon is used in our prototype for model to model transformations.

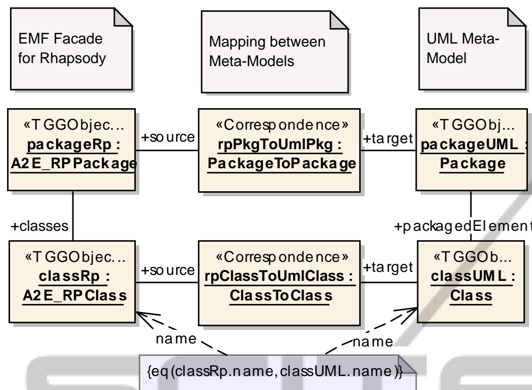

Figure 6: TGG for mapping the Rhapsody facade to UML.

Fig. 6 shows a part of a TGG for mapping Rhapsody classes to UML classes, which are contained in Rhapsody packages or UML packages. A constraint is used for mapping the attribute *name*.

Merapi-Modeling uses its own, UML-like meta-model. Since it is already based on the EMF, eMoflon can directly be used for realizing a model to model transformation to the EMF-based UML2 model.

Papyrus even uses the EMF-based UML2 model for storing its models. There is no need for implementing a model to model transformation.

Now, any existing code generator for the EMF UML2 model can be used for creating source code. Further, code generators can be realized by using mechanisms for model to text transformations (cf. section 2.5).

5 CONCLUSION

Code generators, which are used to directly generate source code from UML models, are in general proprietary for a specific tool. This results in a significant effort, if a certain type of code generation has to be realized for different tools. This implies that there is a need for an approach towards development of tool independent code generators.

On the other hand, in order to make code generators compatible with different tools, they need a uniform input model. There is already a standardized XMI format for the UML. Many tools can export their model in this format. Unfortunately, they export their models differently.

Techniques for model to model transformations could be used for transforming the models of different tools into a standardized UML model. The models of most tools can be accessed by using an API. However, in order to use such techniques, a meta-model is needed for the source and the target model. However, there is a gap between the models of different UML tools and the standardized UML model.

In order to address this gap, an approach for generating a meta-model for APIs is introduced in this paper. Unlike in existing approaches, the model elements of this meta-model are acting as a facade for the original API objects. This allows the direct usage of many techniques for model driven development. Then the model can be transformed into a standardized UML model. The proposed approach is evaluated using a prototype implementation. In the empirical evaluation, four different modeling tools are transformed into an EMF-based UML model. Furthermore, the experimental results indicate that the proposed approach increases the scalability of code generators significantly, if more than one type of code generation has to be realized for more than one UML tool.

MDD for embedded systems benefits from the improved scalability. In order to support different target platforms, many code generators have to be realized or adaptions have to be made. It is more efficient to implement a code generator, which can be used for several modeling tools, instead of implementing proprietary code generators for each tool. Developers of code generators can optimize and improve one code generator rather than spending time on implementing the same code generator again and again for different tools.

There are still possibilities for further optimizing the proposed approach. One aspect is to create UML facades, which interact with the APIs of modeling tools directly. This means to have a seamless UML view on the proprietary models of different modeling tools. Then additional transformations between the meta-model of APIs and the standardized UML models could be avoided. In this context, the need arises for a (new) mechanism to make a mapping between an existing model and code (of an API) and an affiliated facade code generation mechanism. Another aspect is to analyze how the effort of recreating or adapting existing transformations can be reduced, when the meta-model resp. the API of a modeling tool changes.

Tool independent code generation is only the beginning of the potential uses of the presented approach. Future work is about the certification of tool independent code generators and the check of MISRA modeling guidelines. Developers can select the UML

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

tool of their choice and benefit from all those features. Furthermore, the model facade generation can be used to seamlessly integrate any API into MDE.

ACKNOWLEDGMENT

This work is supported by a grant from BMWi (Federal Ministry of Economics and Technology, Germany). This project work is carried out in close cooperation with Willert Software Tools GmbH, Ostfalia University of Applied Sciences and University of Osnabrueck.

REFERENCES

ARM (2013). Cortex M3 processor product website. http://www.arm.com/products/processors/cortex-m/cortex-m3.php.

Atlanmod and Modelum research groups (2013). API2MoL Project Website. https://code.google.com/a/eclipselabs.org/p/api2mol/.

Burke, P. W. and Sweany, P. (2007). Automatic Code Generation Through Model-Driven Design. In *Software Technology Conference (STC)*.

Bzivin, J., Hillairet, G., Jouault, F., Kurtev, I., and Piers, W. (2005). Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *In Proceedings of the conference for Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*.

Eclipse Foundation (2013a). Acceleo Project website. http://www.eclipse.org/acceleo/.

Eclipse Foundation (2013b). Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/.

Eclipse Foundation (2013c). Graphical Modeling Framework Project website. http://www.eclipse.org/modeling/gmp/.

Eclipse Foundation (2013d). Papyrus UML Product website. http://www.eclipse.org/papyrus/.

Eclipse Foundation (2013e). UML2 Project website. http://www.eclipse.org/modeling/mdt/?project=uml2.

Eclipse Foundation (2013f). Xpand Project website. http://www.eclipse.org/modeling/m2t/?project=xpand.

Eclipse Foundation (2013g). Xtend Project website. http://www.eclipse.org/xtend/.

France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model-driven development using UML 2.0: promises and pitfalls. *Computer*, 39:59 – 66.

Hergenhan, A. and Heiser, G. (2008). Operating Systems Technology for Converged ECUs. *Software Systems Research Group*.

Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schfer, W., Lauder, M., Anjorin, A., and Schrr, A. (2013). A Survey of Triple Graph Grammar Tools. In *In Proceedings of the Second International Workshop on Bidirectional Transformations (BX 2013)*.

IBM (2012). Rational Rhapsody Product Website. http://www-01.ibm.com/software/ awd-tools/rhapsody/.

Izquierdo, J. L. C., Jouault, F., Cabot, J., and Molina, J. G. (2012). API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering. *In Information and Software Technology*, 54:257–273.

Jouault, F., Allilaire, F., Bzivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: A QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06). ACM, New York, NY, USA, 719-720.*

Kindler, E. and Wagner, R. (2007). Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. *Technical Report, University of Paderborn*.

Krichen, F., Hamid, B., Zalila, B., Jmaiel, M., and Coulette, B. (2013). Development of reconfigurable distributed embedded systems with a model-driven approach. *Concurrency Computat: Pract. Exper. doi: 10.1002/cpe.3095*.

Macedo, N. and Cunha, A. (2013). Implementing QVT-R Bidirectional Model Transformations using Alloy. In *FASE 2013*.

Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Petin, J., and Levrat, E. (2010). Automatic code generation for embedded systems: From UML Specifications to VHDL code. In *Industrial Informatics (INDIN) 8th IEEE International Conference*, pages 1085 – 1090.

Motor Industry Software Reliability Association (MISRA) (2012). MISRA C website. www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx.

Obeo (2013). UML to Java Generator Project. http://marketplace.obeonetwork.com/module/uml2java-generator.

Object Management Group (2011). Meta Object Facility Specification 2.4.1. http://www.omg.org/spec/MOF/.

Object Management Group (2013a). MOFM2T 1.0 Specification. http://www.omg.org/spec/MOFM2T/1.0/.

Object Management Group (2013b). OMG Website. http://www.omg.org/.

Object Management Group (2013c). QVT Specification 1.0. http://www.omg.org/spec/QVT/1.0/.

Object Management Group (2013d). Unified Modeling Language Specification. http://www.uml.org/.

Ostfalia University (2013). Merapi-Modeling Product website. http://www.merapi-modeling.de.

Rose, L. M., Matragkas, N., and Kolovos, D. S. (2012). A feature model for model-to-text transformation languages. In *In Proceedings of Modeling in Software Engineering (MISE)*.

Sestoft, P., Hulgaard, H., and Wasowski, A. (2002). Code Generation for Embedded Systems. http://www.itu.dk/ wasowski/papers/poster.pdf.

Sparx Systems (2012). Enterprise Architect product website. http://www.sparxsystems.com/products/ea/index.html.

Willert Software Tools GmbH (2013). Company website. http://www.willert.de.