On A-posteriori Integration of Ecore Models and Hand-written Java Code

Thomas Buchmann and Felix Schwägerl
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany

Keywords: Model-Driven Development, Ecore, Code Generation, Java, Reverse Engineering, Model Transformation.

Abstract: Model-driven software engineering emphasizes using models as primary development artefacts. In many cases, the static structure of a software system can be automatically generated from static models such as class diagrams. However, hand-written source code is still necessary, either for specifying method bodies or for integrating the generated code with already existing artefacts or frameworks. In this paper, we present a concept and the corresponding technical solution, which allow to lift up hand-written code for method bodies to the model level and tightly integrate it with the Ecore model. Furthermore, we demonstrate the feasibility of our approach with the help of a concrete use case.

1 INTRODUCTION

Model-Driven Software Engineering (MDSE) (Völter et al., 2006) is a discipline which receives increasing attention in both research and practice. It puts strong emphasis on the development of high-level models rather than on the source code. Models are not considered as documentation or as informal guidelines how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, MDSE aims at the development of *executable* models. The resulting models are then transformed in a series of subsequent transformation steps (Frankel, 2003) into source code which can be compiled and executed on the respective target platform.

Ideally, software engineers operate only on the level of executable models such that there is no need to inspect or edit the actual source code (if any). In this sense, models are the code (now written in a high-level modeling language). However, practical experiences have shown that language-specific adaptations to the generated source code are frequently necessary.

Object-oriented modeling is centered around class diagrams, which constitute the core model for the structure of a software system. From class diagrams, parts of the application code may be generated, including method bodies for elementary operations such as creation/deletion of objects and links, and modifications of attribute values. However, for user-defined operations only methods with empty bodies may be generated which have to be filled in by the programmer.

The Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) has been established as an extensible platform for the development of MDSE applications. It is based on the Ecore metamodel which is compatible with the OMG Meta Object Facility (MOF) specification (OMG, 2011).

In EMF, for instance, only structure is modeled by means of class diagrams, whereas behavior is described by modifications to the generated source code. However, EMF is already tuned for efficient programming, as it demands for hand-written Java code for method bodies. Users are able to annotate the respective parts and the Eclipse Modeling Framework uses a code-merging generator to preserve these fragments on subsequent code generation steps.

We apply our approach to a specific problem scenario — the derivation of products in model-driven software product lines. Nevertheless, the approach can also be used in single system development.

This paper is structured as follows: In Section 2, we discuss our contribution. A detailed example showing a use case for our approach is described in Section 3. Related work is discussed in Section 4 before Section 5 concludes the paper.

2 EMF - JAVA INTEGRATION

In this section, we describe our approach to unify EMF modeling and Java programming. After giving a

Buchmann T. and Schwägerl F..
On A-posteriori Integration of Ecore Models and Hand-written Java Code.
DOI: 10.5220/0005552200950102
In *Proceedings of the 10th International Conference on Software Paradigm Trends* (ICSOFT-PT-2015), pages 95-102
ISBN: 978-989-758-115-1

brief overview about EMF and MoDisco (Bruneliere et al., 2010), we discuss our solution in detail.

2.1 EMF Overview

The Eclipse Modeling Framework (EMF) with its metamodel Ecore is the standard platform for model-driven software development in Eclipse, and it is especially wide-spread in the academic community. It follows a minimalistic and pragmatic modeling approach. Ecore only comprises the core concepts of object-oriented modeling and thus allows for a straightforward mapping to Java code.

Figure 1 depicts the development process imposed by EMF. Typically, modelers use Ecore class diagrams to describe the static structure of the software system. The code generator provided with EMF maps the class model to a set of corresponding Java interfaces and implementation classes. Furthermore, the correct semantics of references between the classifiers is ensured.

However, for user-defined operations only the headers of the corresponding Java methods are generated. Body implementations have to be specified directly in Java afterwards. EMF uses a code-merging generator in order to deal with these additions to the generated code supplied by the user. Corresponding Javadoc-tags mark regions in the source code which should be preserved on subsequent EMF code generation runs. Unfortunately, these Javadoc-tags have to be specified before the corresponding method declaration. Thus, if a method declaration is annotated with such a Javadoc-tag because it contains a user-defined body, it will never be modified by the EMF code generator. In particular the following operations may cause problems:

Deletion. In case the corresponding EOperation is deleted from the class model, the resulting Java code will still be present in the source code after subsequent generation steps.

Modification. If the EOperation is modified in the Ecore class diagram (e.g., renaming, changing parameters or return type), these changes cannot be propagated to the generated code, since the corresponding code fragments are protected. Instead, a new method declaration with empty body is generated.

2.2 Integration of Method Bodies

To avoid the problems described above, integrating the hand-written method bodies in the Ecore model is a possible solution. If the bodies are already present

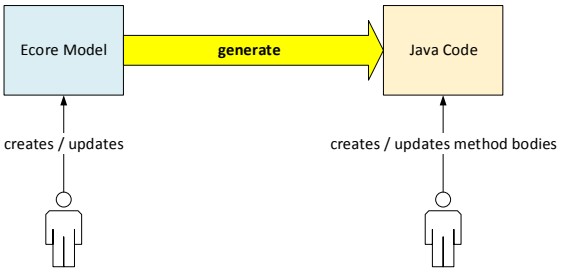

Figure 1: The typical MDSE Process in EMF.

when the EMF code generator starts its work, there is no need to protect certain methods with corresponding Javadoc-tags. As a consequence, changes in the Ecore class diagram are propagated to the generated code correctly:

Deletion. If an EOperation is deleted from the class model, its corresponding body is deleted as well.

Modification. All changes to the definition of an EOperation (i.e. return types, parameters, or name) are propagated to the generated code.

In the following, we describe how this integration can be achieved automatically.

The EMF code generator is always invoked on a so called generator model. The generator model wraps the Ecore model and adds some generation-specific meta information, e.g. target directories, code style, and so forth.

Ecore models may be annotated with so called EAnnotations for a number of different purposes. These annotations are used to store information which is not explicitly supported in the Ecore metamodel (Steinberg et al., 2009). EMF's standard annotations may be classified into different categories based on the source it uses for each kind:

Ecore. Annotations of this kind are attached to EModelElements in order to specify additional information which is relevant at both runtime and code generation.

GenModel. GenModel-sourced annotations are used to attach information that is only relevant when generating code to respective EModelElements.

Extended Metadata. Annotations of this kind are used in models that were created from XML Schema.

EMOFTags. Tags in EMOF are used for the same purpose as EAnnotations are used in Ecore. Since EMF provides an interchange between Ecore and EMOF, EMOFTags are required to map EAnnotations to corresponding tags.

Since the Ecore metamodel does not cover method bodies, annotations are required to make them part of

a model. Since this information is required only during code generation, *GenModel* annotations have to be used. EMF already provides a pre-defined annotation for this purpose: A GenModel-sourced annotation for the type EOperation exists. It allows to specify a key value pair. The key *body* indicates that the corresponding value field contains Java code that implements the operation. However, iterating through the model and adding corresponding annotations for each user-defined methods is a cumbersome and time-consuming task. Thus, we strived for a complete automation of this process.

To this end, we use the MoDisco[1] framework. Originally, MoDisco is dedicated to software modernization projects. It provides a set of discoverers for different types of artefacts, e.g. a discoverer for Java source code. Thus, it allows to parse existing Java source code into an Ecore-based model, which resembles the abstract syntax tree (AST) of the Java language (c.f. Fig. 2).

MoDisco also provides a code generator based on Acceleo[2], which allows to generate Java code from the Ecore-based AST model.

In order to integrate hand-written method bodies into the Ecore model automatically, first the complete Java code is discovered using MoDisco, resulting in a Java AST model (based on Ecore). Then the procedure iterates over the Ecore model and checks for each EOperation if a corresponding body implementation is present in the AST model. If this is the case, a corresponding GenModel-EAnnotation is created. The EAnnotation uses a key-value pair to specify the purpose. The value contains the Java code that realizes the corresponding method implementation. In order to obtain this Java code from the AST model, the code generator is invoked only for the desired fragment (in this case the Block which is contained in the corresponding MethodDeclaration) and the resulting string is stored in the value field of the newly created EAnnotation. Using our approach, the user has the full benefits of his preferred modeling environment (e.g. the Ecore class diagram editor) and his preferred programming environment (e.g. the Eclipse JDT Java editor). Furthermore, code completion and syntax highlighting are available when the hand-written body implementations are created, which is not the case if the standard EMF annotation editor is used.

The resulting Ecore model now contains the static structure of the software system as well as all hand-written body implementations for corresponding EOperations. All subsequent changes to the structural model may now be propagated correctly to the generated code, as there is no need to protect these hand-written parts any longer.

[1]https://eclipse.org/MoDisco/

[2]http://www.eclipse.org/acceleo

3 EXAMPLE

In this section, we present a use case for our solution in the context of model-driven software product line engineering (MDPLE). Please note that the problem is specific to MDPLE. However, the solution which is presented here can be used in general for every model-driven software project which is realized with the help of EMF.

3.1 Problem Description

One area of our research is dedicated to model-driven *software product line engineering* (SPLE). SPLE (Clements and Northrop, 2001) addresses the organized reuse of software artifacts. Feature models (Kang et al., 1990) are used to capture the commonalities and differences of members of a product line, while feature configurations describe the characteristics of a specific member thereof. Software product line engineering is divided into two levels. (1) *Domain engineering* is used to analyze the domain and capture the commonalities and variabilities in a feature model. Furthermore, the features are realized in a corresponding implementation. In model-driven software product lines, models represent the implementation at a higher level of abstraction. (2) *Application engineering* deals with binding the variability defined in the feature model and deriving concrete products.

In SPLE, basically two different approaches exist to realize variability in the corresponding feature implementation: (1) In approaches based on *positive variability*, product-specific artifacts are built around a common core. During application engineering, composition techniques are used to assemble the final product using these artifacts. (2) In approaches based on *negative variability*, a superimposition of all variants is created. The derivation of products is achieved by removing all fragments of artifacts implementing features which are not contained in the specific feature configuration for the desired product.

Several approaches exist to associate elements of the feature model with artifacts part of the domain model; in previous publications (Buchmann and Schwägerl, 2012) FAMILE has been presented, an EMF-based tool chain for model-driven software product line development using negative variability. All common approaches for model-driven software product line engineering only support homogeneous artefacts. In (Buchmann and Schwägerl, 2014) an

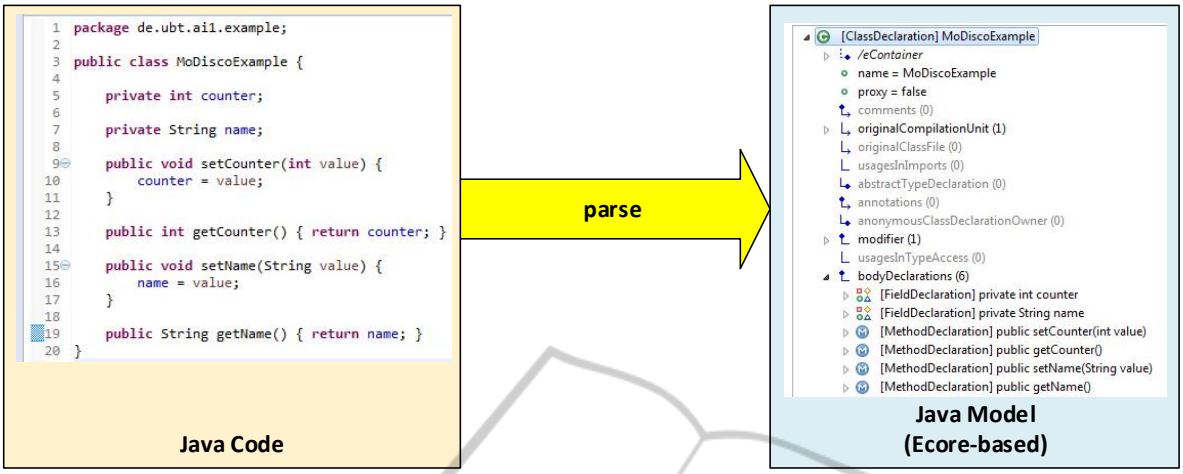

Figure 2: Using MoDisco to parse Java code into an Ecore-based model.

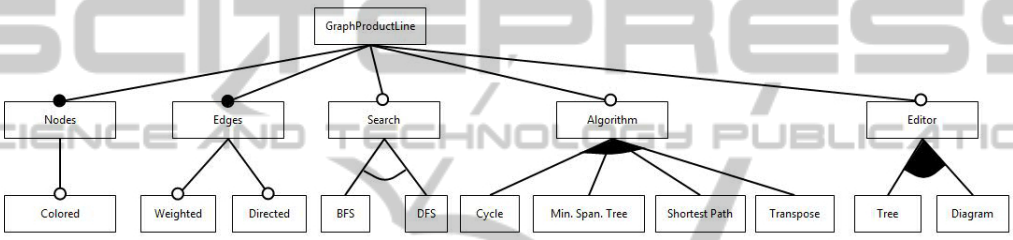

Figure 3: Feature model for the graph product line.

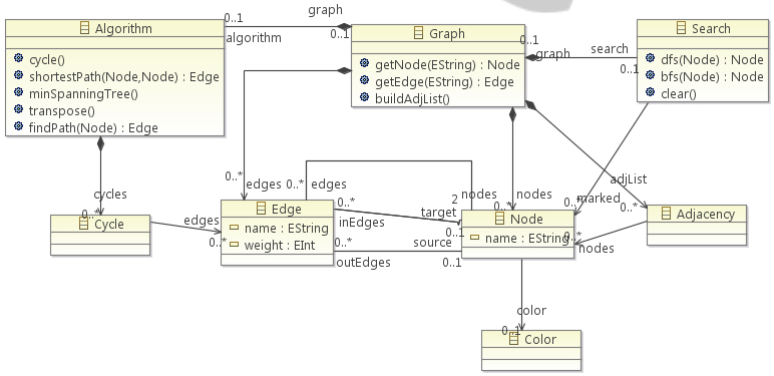

Figure 4: Ecore model for the graph product line.

extension to FAMILE allowing for heterogeneous projects is described. In this case, the platform of the product line may consist of models and hand-written source code. However, when deriving products, the variability information needs to be kept consistent across all artefacts.

A prominent example in literature on software product lines is a product line for graphs. The corresponding feature model is shown in Fig. 3. A graph always consists of nodes and edges (filled dots) an optionally (unfilled dots) one (unfilled arc) search strategy (bfs or dfs) and an arbitrary number (filled arc) of algorithms. Furthermore, edges may have a weight or they may be directed.

Figure 4 depicts the multi-variant domain model of the graph product line. Following the model-driven approach, an object-oriented decomposition of the underlying data structure is applied: A Graph contains Nodes and Edges. Furthermore it may contain a Search strategy and Algorithms operating on the graph data structure. For performance reasons, the data structure may be converted into an Adjacency list, to

speed up certain algorithms. As the model depicted in Fig. 4 is the superimposition of all variants, the relation between nodes and edges is expressed in multiple ways: (1) In case of undirected graphs, an edge is used to simply connect two nodes, expressed by the reference nodes. (2) Directed graphs on the other hand demand for a distinction of the respective start and end nodes of an edge. This fact is expressed by two single-valued references named source and target, respectively.

```
246     /**
247      * <!-- begin-user-doc -->
248      * <!-- end-user-doc -->
249      * @generated NOT
250      */
251     public EList<Node> dfs(Node node) {
252         if (getMarked().contains(node))
253             return getMarked();
254         getMarked().add(node);
255
256         // undirected graphs
257         for (Edge e : node.getEdges()) {
258             for (Node n : e.getNodes()) {
259                 if (n != node)
260                     dfs(n);
261             }
262         }
263
264         // directed graphs
265         for (Edge e : node.getOutEdges()) {
266             dfs(e.getTarget());
267         }
268
269         return getMarked();
270     }
```

Figure 5: Example for a multi-variant method body written in Java.

As stated above, Ecore only allows for structural modeling, i.e., it does not provide support to model method bodies. Thus, the standard EMF development process (Steinberg et al., 2009) demands for a manual specification of an EOperation's body by completing the generated source code. In the example, hand-written Java source code for all operations contained in the class diagram shown in Figure 4 has been supplied. A small cut-out of a method implementation for the class Search is shown in Figure 5. In the corresponding Ecore model (cf. Fig. 4), the Search class defines three EOperations. While the EMF code generation only creates Java code for the method header, the body implementation depicted in Figure 5 was supplied manually. In this case, the method implementation also contains variability as the corresponding references between nodes and edges are different depending on the presence or absence of the feature *Directed* in the current feature configuration. Please note that the level of granularity supported by FAMILE's variability annotations is arbitrary, ranging from single Java fragments over statements, blocks, methods, or even classes and packages.

As mentioned earlier, FAMILE supports the development of software product lines based on negative variability. Thus, when deriving specific products based on a concrete feature configuration, all fragments and artefacts which implement unselected features have to be removed. Figure 6 depicts the situation that would occur if only standard EMF technology without the mechanism described in this paper would have been used.

During *Domain Engineering*, the platform containing all variants is created. This can be done in a model-driven way using Ecore models to describe the static structure of the software. Then the Ecore code generator is invoked (cf. step 1 in Fig. 6) and hand-written Java code is used to supply the method bodies. The hand-written code is added to the generated one and then discovered into an Ecore-compliant AST model in order to be able to use FAMILE for variability management on the source code fragments. The user now may annoate the respective artefacts with variability information. Annotations concerning the structure are performed on the level of the Ecore model. For example, class Search shown in Fig. 4, contains the operations dfs(Node) and bfs(Node) respectively, which represent the different search strategies which are used in graphs and which are annotated with the corresponding features from the feature model (cf. Fig. 3). Feature annotations in the Ecore model can be used on any level of granularity. E.g. classes, attributes, methods, parameters or references may be annotated.

As the variability information (for the static structure) that has been added to the Ecore model in domain engineering is not present in the generated code, the discovered Java AST model also does not contain it. Furthermore, annotating for example an EAttribute in the Ecore class diagram would require the user to annotate the corresponding field declaration and the respective accessor methods in the generated source code. Of course this is not feasible, since one of the goals of the FAMILE tool chain is to keep the annotation effort for the user as small as possible. Furthermore, in order to consistently annotate the Ecore model and the generated parts in the Java model, the user would require knowledge about the Ecore code generator. However, the hand-written body implementations may also require variability. E.g. the implementation of the method dfs(Node) contains different fragments which are used for directed and undirected graphs respectively. The user may annotate these blocks with corresponding feature annotations. Please note that these annotations are performed di-

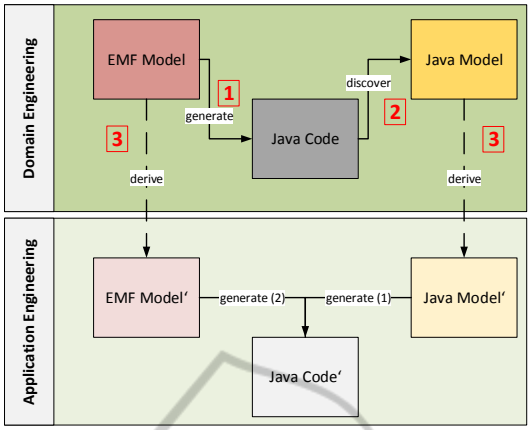

Figure 6: The interplay between model and hand-written code in (heterogeneous) model-driven software product lines.

rectly from the Eclipse JDT editor. The FAMILE tool chain maps these annotations to the discovered MoDisco Java model as FAMILE operates on Ecore-based models only (cf. step 2 in Fig. 6).

During *Application Engineering*, when unused fragments are filtered from the multi-variant models, the corresponding target models are derived (EMF Model' and Java Model' respectively, cf. step 3 in 6). In an ideal world, i.e. if both models are in sync in terms of variability information, the user could invoke the code generation for the Java model and afterwards the code generation of the EMF model in order to obtain the final source code for the desired product. However, reality is different: The EMF code merging generator does not remove files. For example, an annotated class of the Ecore model has been filtered during the derivation process, but it is still present in the Java model. If the code generation for the Java model is invoked first, corresponding Java code for this class is generated which is not deleted on a subsequent run of the EMF code generation. The same holds for operations: The EMF code generation requires that hand-written code is marked in order to preserve it during subsequent generation steps. In case an EOperation that has been extended with a hand-written body is filtered in the Ecore model, this mechanism prevents it from being deleted.

A possible solution would be to add information about the conceptual links between Ecore models and the corresponding generated Java code to the FAMILE tool chain. However, due to the following reasons, this is not feasible: (1) FAMILE is as general as possible, as the only requirement is that the domain models are based on Ecore. No assumptions about certain concepts are made and thus FAMILE may be used for a vast variety of artefacts. (2) The EMF code generation could be modified by the EMF developers which would then require modifications of FAMILE.

Thus, the approach as described in this paper offers a tool-independent solution for this problem.

3.2 Solution

Figure 7 shows how the approach discussed in this paper solves the problem shown above and related problems of the same class. The domain engineering phase is carried out as described earlier. Once both models are derived in application engineering, there is no need to invoke two different code generation steps. In detail, our automated solution performs the following tasks on both models:

Traverse Ecore Model. First, the Ecore model is traversed and for each EOperation the following tasks are performed:

Locate Method Implementation. In the Java model, the corresponding MethodDeclaration is located. We make sure, that it really contains a respective body implementation and the appropriate Javadoc tag.

Invoke Code Generation Template. If this is the case, we need to invoke MoDisco's Java code generation only for this model fragment (i.e. the block containing the body implementation).

Add EAnnotation. Finally, the EOperation from the Ecore model is extended by an appropriate EAnnotation: A GenModel-source annotation with key *body* is created. The corresponding *value* is the result of the code generation run in the previous task described above (cf. Fig. 8).

After the Ecore model has been traversed and the above steps have been excuted for all user-defined EOperations, the models are in sync and the user only needs to invoke the EMF code generator (just as in a

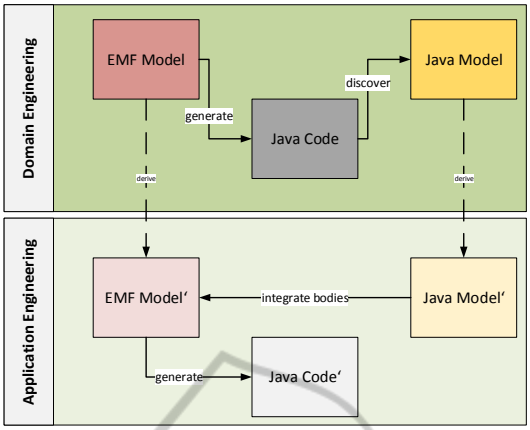

Figure 7: Our solution applied for model-driven software product line engineering.

regular EMF project) in order to obtain the final and consistent Java source code for the desired product.

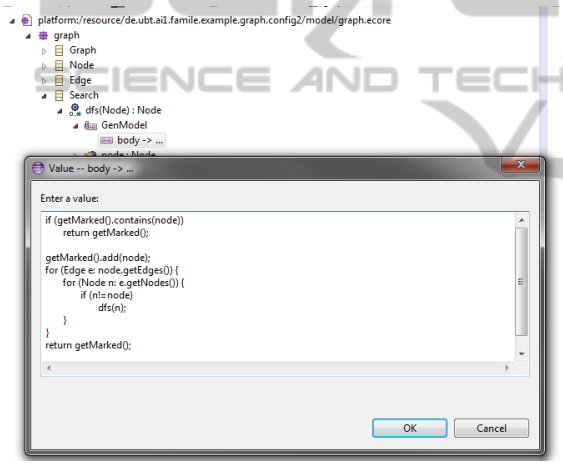

Figure 8: The final Ecore model, after applying our solution.

Please note that the proposed solution is the only feasible way to achieve consistency and to allow variability on arbitrary levels of granularity. E.g. a solution which would not use the Java model, but would integrate the method bodies into the Ecore model directly in domain engineering does not allow to specify variability in body implementations as shown above.

4 RELATED WORK

During the last few years, several approaches have been published which extend EMF with capabilities for behavioral modeling. Graph rewriting techniques are used in Henshin (Arendt et al., 2010), MDELab (Giese et al., 2009), or ModGraph (Buchmann et al., 2011). While the first ones are based on an interpreter, i.e. there is no generated code, ModGraph on the other hand generates method bodies for user-defined operations out of graph transformation rules.

However, while graph transformation rules provide a benefit for some complex operations, they even reduce the level of abstraction in terms of control flow or in case the problem which has to be solved demands for an imperative solution rather than a declarative one (Buchmann et al., 2012). Thus, hand-written Java code is still required for large parts of a software system.

Xcore[3] is a textual DSL which allows to define both the static structure and the behavior of Ecore models. It provides a code generator that allows to generate Java code from the Xcore specification. However, although Xcore has a Java-like syntax, it is another new language which users have to learn.

The problem described in this paper could also be solved using an incremental bi-directional Model-to-Text transformation. So far there is no tool which meets these requirements. There are incremental code generators (e.g. JET, Xpand or Acceleo) in the EMF context, but they only operate in one direction.

To the best of our knowledge, our approach is the only one which offers the possibility to combine Ecore modeling and standard Java programming on the modeling level.

5 CONCLUSIONS

In this paper, we presented an innovative approach for bridging the gap between the model level and the source code level. Furthermore, as a proof of concept, we presented an implementation for the Eclipse Modeling Framework. EMF requires hand-written

[3]https://wiki.eclipse.org/Xcore

method bodies, since it only allows for structural modeling. The EMF code generation engine is able to preserve these user-supplied code fragments on subsequent generation steps. However, using this mechanism, modifications and deletions in the Ecore model are no longer propagated to the corresponding code fragments.

We presented an approach that makes use of so called *GenModel*-Annotations, allowing to specify information which is not originally supported by the Ecore metamodel. In advance, we use MoDisco to automatically parse the Java source code into a corresponding AST model. Model transformations are applied to extract the required information from the AST model and to add it to the Ecore model using EAnnotations.

Furthermore, we explained in detail how this approach provides a significant improvement in model-driven software product line engineering (MDPLE): Since we use a generic tool chain for MDPLE, conceptual links between different models, e.g. an Ecore model and a corresponding Java model containing body implementations cannot be hardcoded in the tool. In order to provide consistency between these types of models, the information stored in both of them has to be integrated using the approach discussed in this paper.

ACKNOWLEDGEMENTS

The authors want to thank Bernhard Westfechtel for his valuable and much appreciated comments on the draft of this paper.

REFERENCES

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place EMF model transformations. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, volume 6394, pages 121–135, Oslo, Norway. Springer Verlag.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174, New York, NY, USA. ACM.

Buchmann, T. and Schwägerl, F. (2012). FAMILE: tool support for evolving model-driven product lines. In Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., and Tolvanen, J.-P., editors, *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, CEUR WS, pages 59–62, Building 321, DK-2800 Kongens Lyngby. Technical University of Denmark (DTU).

Buchmann, T. and Schwägerl, F. (2014). A model-driven approach to the development of heterogeneous software product lines. In Mannaert, H., Lavazza, L., Oberhauser, R., Kajko-Mattsson, M., and Gebhart, M., editors, *Proceedings of the Ninth International Conference on Software Engineering Advances (ICSEA 2014)*, pages 300–308, Nice, France. ICSEA.

Buchmann, T., Westfechtel, B., and Winetzhammer, S. (2011). MODGRAPH - A Transformation Engine for EMF Model Transformations. In *Proceedings of the 6th International Conference on Software and Data Technologies*, pages 212 – 219.

Buchmann, T., Westfechtel, B., and Winetzhammer, S. (2012). The added value of programmed graph transformations — a case study from software configuration management. In Schürr, A., Varro, D., and Varro, G., editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, Budapest, Hungary. Presented at AGTIVE 2011, currently under review for publication in the post-proceedings.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Boston, MA.

Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, IN.

Giese, H., Hildebrandt, S., and Seibel, A. (2009). Improved flexibility and scalability by interpreting story diagrams. *ECEASST*, 18.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute.

OMG (2011). *Meta Object Facility (MOF) Core*. Object Management Group, Needham, MA, formal/2011-08-07 edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Boston, MA, 2nd edition.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.