Ecoreification: Making Arbitrary Java Code Accessible to Metamodel-Based Tools

Heiko Klare, Erik Burger, Max Kramer, Michael Langhammer, Timur Sağlam, Ralf Reussner
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{heiko.klare|burger|max.e.kramer|michael.langhammer|timur.saglam|reussner}@kit.edu

***Abstract*—Models are used in software engineering to describe parts of a system that are relevant for the computation of specific analyses, or the provision of specific functionality. Metamodeling languages such as Ecore make it possible to realize analyses and functionality with model-driven technology, such as transformation engines. If models conform to a metamodel that was expressed using Ecore, numerous Eclipse-based tools can be reused to directly analyze, display, or transform models. In many software projects, models are, however, realized with objects of plain-old Java classes rather than an explicit metamodel, so these popular tools cannot be used.**

In this new ideas paper, we present an *Ecoreification* approach, which can be used to automatically extract Ecore-conforming metamodels from Java code, and a code generator that combines the benefits of both worlds. The resulting code can be used exactly as before, but it also uses the modeling infrastructure and implements all interfaces for Ecore-based tooling. This way, arbitrary non-standard models can be displayed and modified, for example using graphical Sirius editors, or transformed with well-proven transformation languages, such as QVT-O or ATL.

I. Introduction

Models and metamodel-based tools are an important part of modern software engineering. The original vision of model-driven software development as defined in the OMG MDA standard [1] contained a promise that, with the shift from code to models as the primary development artifact, writing program code would become a secondary activity in software development, or could even be completely automated. Although code generation is widely used in specific domains, such as embedded software for automobiles, it has not been adopted by the majority of software developers. Among the reasons for this are poor quality of generated code, and a lack of synchronization mechanisms for manual changes that are applied to code [2]. Although most developers use models at some point in the development process, truly model-driven development of software, in which the generated program code is not modified manually, is still limited to specialized domains because of the aforementioned shortcomings.

These factors already hinder the adoption of model-driven technologies during the development of new systems that are created from scratch. For existing systems with a large code base, for example in Java, the introduction of model-driven technologies may even lead to further problems: Although technologies exist to enrich Java code with information so that a metamodel can be created from an existing class structure, heavy modifications are often necessary so that the code conforms to a metamodeling standard, such as the Eclipse Modeling Framework. These modifications may even change the API of the existing Java code, which triggers additional modifications in all software that interacts with the existing code base. Especially for long-living software systems, which may already have been developed before model-driven methods were available, such breaking changes are not acceptable. This is why many developers refrain from modifying code, even though the information that is expressed by it would constitute a metamodel that could be used in a platform-independent way once extracted, with the added value of being able to use, e.g., model transformation languages.

In this paper, we present our vision of an approach to make arbitrary Java code accessible to the modeling tools of the Eclipse Modeling Framework using its metametamodel *Ecore*. Thus, we call our approach *Ecoreification*, since Java classes with their methods and fields are turned into metaclasses of a metamodel conforming to Ecore. The process of extracting the metamodel is automated with the possibility for developers to take decisions manually at certain interaction points. We present our preliminary results and a prototypical implementation of the extraction process.

After a description of foundations and the introduction of a running example in Section II, we discuss the addressed problem and introduce our novel Ecoreification concept in Section III. Then, we present a proof-of-concept prototype and initial results in Section IV. Finally, we discuss key challenges in Section V, present related work in Section VI, and conclude the paper in Section VII.

II. Foundations and Running Example

In this section, we describe the Eclipse Modeling Framework with a special focus on the interaction with Java code. We also define a running example.

A. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [3] is one of the most common approaches for Model-Driven Software Development (MDSD). EMF provides a metametamodel called *Ecore*, which conforms to the Essential Meta-Object Facility (EMOF) [4]. Thus, Ecore-conforming metamodels and EMOF-conforming metamodels are interchangeable, as they can be transformed into each other without information loss.

Ecore-conforming metamodels consist of metaclasses, which in turn consists of structural features and a functional interface. A structural feature can be an attribute, whose type is

primitive or an enumeration, or a reference, whose type is another metaclass, and which describes the relations between metaclasses. The functional interface of a metaclass is defined by operations, analog to methods in programming languages. Nevertheless, metaclasses usually only define the signatures of methods but not their implementation.

EMF provides functionality to generate Java code from metamodels. In that code, metaclasses are represented as classes, which define the access and modification logic for their features, for example, with accessors and mutators for attributes and references, and which specify methods according to the operations defined in the metaclasses. The functionality of metamodel operations is usually defined using Java code and can be implemented within the generated code. To avoid that the implementation gets overwritten after a metamodel has been modified and code generation has been executed again, the operations can be provided with specific annotations.

The code that EMF generates from a metamodel follows a specific, predefined structure, so that tools that are developed for EMF can expect and rely on a certain structure of metamodel code. For example, every metaclass is represented as a pair of interface and implementation class with a certain naming schema, and for every package in the metamodel, a factory for metaclasses of that package is generated, whose factory methods also follow a specific naming schema. Several tools that rely on this specific code structure are already deployed with EMF, for example, a generic, tree-based model editor, which uses reflection that assumes the code structure.

Several model-driven tools have been developed for EMF. Examples are transformation languages such as QVT-O [5], textual model editors based on Xtext [6], a framework for developing DSLs, or graphical model editors based on Sirius [7].

B. Running Example

As an example of software that can be developed using a model-driven approach, we have chosen a graph library that describes graphs as a metamodel with metaclasses for vertices and edges. An example for such a graph-processing library is Apache Flink Gelly.[1] It provides classes for describing graphs and a library with algorithms to run on them, such as clustering, link analyses and shortest paths. The project has not been developed with model-driven techniques, but the graph structure could of course be expressed as an Ecore-conforming metamodel, which would be beneficial for several applications. For example, graphical editors for graphs could be created with low effort using a framework such as Sirius. It would then only be necessary to specify how instances of certain metaclasses and their features shall be represented by predefined graphical elements, such as nodes and edges. Another example for a benefit is the possibility to define transformations into other metamodels. These metamodels may specify different graph representations or even completely different formalisms, for which further algorithms and tools exist.

A simplified metamodel for graphs, based on the description in Apache Flink Gelly, is depicted as a UML class diagram in Figure 1. It shows how metaclasses for vertices and edges are used to model three different types of graphs: *Directed graphs* may have several edges from a source s to a target t. *Oriented graphs* may only have a single edge that connects s and t. *Ordered trees* are acyclic, oriented graphs that are connected and for which a root vertex as well as an ordering for all vertices is given. To account for these properties the two graph metaclasses and the interface of the example have different fields and methods. We only show those that are necessary to explain our approach and omit, for example, mutators.

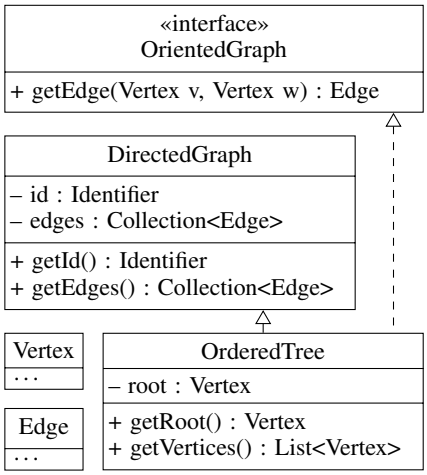

Fig. 1. Representation of graphs as a metamodel, based on the graph-processing library Apache Flink Gelly

III. Ecoreification

In this section, we discuss the problem which our novel Ecoreification approach addresses and present its overall concept and envisioned benefits. More details on the challenges of the approach, the technical realization and results of first experiments with a proof-of-concept prototype are presented in the subsequent sections.

A. Implicit Metamodels in Software Code Bases

Many software projects are long-living, which means that they may already have existed before model-driven techniques and tools were developed. The source code of such projects often contains parts that model domain information or other structures which can also be understood as an implicit metamodel, but are not explicitly denoted as such.

For applying model-driven tooling, implicit metamodels have to be made explicit, since only an explicit metamodel provides the information for model-driven tooling according to the structure prescribed by the metametamodel and on the level of abstraction that is necessary. For example, all features of metaclasses in an Ecore-conforming metamodel is publicly accessible. In the source code, it is, however, hard to determine if a field of a class is only responsible for representing internal state, or if it is a property that is publicly accessible.

[1] https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html

B. From Implicit to Explicit Metamodels

A straightforward approach to make implicit metamodels explicit is to manually create Ecore-conforming metamodels that reflect the implicit metamodels in the source code. Nevertheless, such manually created metamodels are completely independent from the originally developed code. That code does usually not conform to the structure that is automatically generated by the EMF code generator as required by model-driven tooling. As a consequence, one of the following approaches is necessary to integrate EMF functionality into the original source code: either code has to be generated using the EMF code generator, and the functionality of the existing code has to be integrated into that generated code, potentially resulting in a changed API, or the existing code has to be modified such that it conforms to the structure of the code generated by EMF.

This gap between manually written source code and code generated from metamodels by EMF usually requires metamodels to be used from the beginning of a project, as it is difficult to manually extract metamodels from source code later on. Tools like Xcore [8] try to close this gap between modeling and programming by providing a textual syntax for metamodels and a way to define the functionality of methods within a metamodel using the Java dialect Xtend [9]. Nevertheless, Xcore is not a programming language suited for the productive development of complete software systems, as its functionality is still constrained to the capabilities of Ecore. Private or protected methods, for example, cannot be defined. Furthermore, Xcore still generates ordinary EMF code from the metamodel, which results in the same difficulties when trying to extract explicit metamodels from existing code. Even Annotated Java [3], a mechanism for defining metamodels by annotating source code, does not simplify this extraction process, because the annotated source code still has to follow the specific structure of generated EMF model code.

To summarize the problem: the source code of existing software often contains implicit metamodels so that model-driven tools that need an explicit metamodel cannot be applied. To close this gap between manually written code and explicit metamodels, a mechanism is needed that makes implicit metamodels of existing source code explicit, and adapts the code so that existing model-driven tools can be reused.

C. Ecoreification Concept

To overcome the aforementioned problem, we present an Ecoreification approach for arbitrary Java code. This process turns Java classes with appropriate methods and fields into metaclasses of an Ecore-conforming metamodel. Based on this extracted metamodel, the Ecoreification process produces Java code that provides the original custom API and all interfaces that are used by EMF-based tools. For our running example, this means that existing Java classes, such as `DirectedGraph` or `Edge`, are identified as metaclasses, and the methods `getVertices` etc. are identified as references. From this extracted graph metamodel, new code is generated and integrated with the existing code. All existing applications and libraries that used the old code will continue to work as before. In addition, EMF-based tools can be used, for example, to display graph models or to transform them into instances of other metamodels, for which analyses or code generators already exist.

The approach consists of three individual steps for

1) extracting a metamodel from existing source code,
2) generating code to support modeling tools, and
3) integrating this code with the original code to preserve the API and all code based on it.

For the first step, we propose a semi-automated extraction algorithm, in which we map Java constructs to Ecore elements. Since Ecore-conforming metamodels have a more restricted structure than manually written code, not all information can be transferred from source code to metamodels directly. Furthermore, Ecore-conforming metamodels contain information that is not made explicit in the source code and not all information of the code should be represented in an abstract model. From these differences between metamodels and source code, we have identified four key challenges for the extraction of metamodels from source code, which are all concerned with identifying modeling constructs within code: i) multiplicities of features, ii) containment references, iii) non-default constructors, and iv) features to be abstracted away in models. We propose solutions for these challenges in Section V.

In the second step, the ordinary EMF code generator can be reused. The resulting code can be used to create instances of the extracted metaclasses, which represent objects of the analyzed Java classes.

In order to automatically obtain objects for the original Java classes *and* for the extracted metaclasses, the original code and the generated metamodel code have to be integrated in the third step. For this, we propose a code structure in which the original Java classes extend unification classes. These classes unify the original code and the model code by delegating method calls to model classes and by implementing model interfaces.

D. Envisioned Benefits

The presented Ecoreification approach is beneficial for software development projects with an existing Java code base and for the development of new code. On the one hand, developers that apply the approach to an *existing code base* can profit from all EMF-based tools without the need to explicitly define a metamodel. As a result of this reuse, new model-based functionality can be added to a software system at relatively low costs, depending on whether EMF-based tools are directly reused or adapted. On the other hand, developers of *new software* can benefit from the integration of model code and non-model code: Custom code neither needs to be attached to a metamodel nor is it necessary to modify generated code. Model parts of the code are automatically detected, and all additional code is kept as it is during the code integration.

As soon as the Ecoreification is able to extract all model-relevant features from code, the further evolution of the software can be performed within the extracted model instead of the original code. Only if the code is maintained by third parties and thus evolves independently from the extracted metamodel, the Ecoreification has to be performed again after modifications.

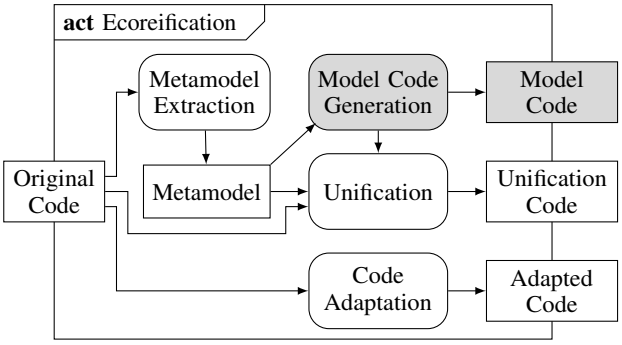

Fig. 2. Ecoreification process combining metamodel extraction, unification and adaptation with ordinary model code generation (in gray)

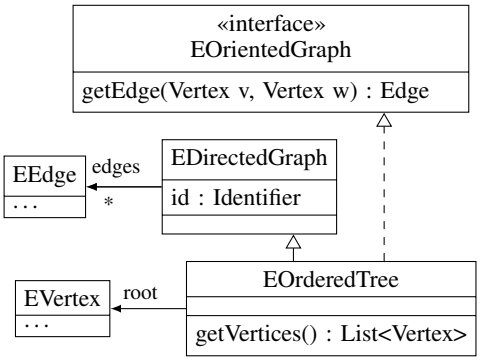

Fig. 3. Extracted metamodel of the running graph library example

IV. Proof of Concept

We have performed first experiments with a proof-of-concept implementation of our Ecoreification approach to investigate its feasibility [10]. There are, however, important challenges that we have not addressed yet, such as the extraction and integration of code for containment references. Therefore, we describe the current realization of our prototype and our plans for fully implementing and evaluating the approach. This section is structured along the three steps of extracting (IV-A), generating (IV-B), and integrating code (IV-C). The Ecoreification process and relevant artifacts are depicted in Figure 2.

A. Metamodel Extraction

The extraction of a metamodel from original Java code is performed in two passes: First, all packages and all the types they contain are identified together with the class inheritance and interface realization relations between these types. Then, all fields and methods of the Java classes that were discovered as metaclasses are analyzed to identify attributes and references.

In the first pass, all Java packages, classes, interfaces, and enumerations, as well as generalization and realization relations between them, are extracted as corresponding Ecore elements (`EPackage`, `EClass`, and so on). The only two exceptions from this one-to-one mapping are the relations of enumerations and nested classes. Generalization and realization relations between enumerations are possible in Java, but not in Ecore. Therefore, our prototypical metamodel extractor currently ignores such relations. To support them in the future, it would be possible to map Java enumerations with such relations to ordinary Ecore metaclasses. Furthermore, Ecore does not support nested classes, which is why a nested class is extracted as a metaclass in a subpackage that has the name of the nesting class.

In the second pass, features are extracted: Primitive types, generic type parameters, methods, and transient modifiers for fields are extracted as their direct Ecore counterparts. Accessor and mutator methods have, however, a special role with respect to fields: A field for which an accessor method exists is either mapped to an attribute or to a reference, depending on the type of the field. If a mutator is not present for the field, then the resulting attribute or reference is extracted as unchangeable. This means accessor and mutator methods are not extracted but only influence the decisions to extract fields as features and to make them changeable or not. Java fields marked as *final* are mapped to unchangeable features. Modifiers for access, abstract methods, and static elements are not extracted as they have no counterpart in Ecore-conforming metamodels on purpose. Arrays are supported using data types with special creation methods in the package factory of the metamodel.

In our prototype, users can specify an extraction scope to limit the extraction to certain Java types. The remaining Java code cannot be ignored, because types in the extraction scope may reference types outside the scope. For these referenced external Java types, we extract data types for the metamodel to ensure that the final integrated code behaves as expected.

We illustrate the metamodel extraction step in the class diagram of Figure 3, which shows the extracted metamodel for the Java code of our running example introduced in Figure 1. The diagram shows the metamodel after the first step of the Ecoreification process. In this and all subsequent figures, the letter *E* is prepended to all names of metaclasses and interfaces so that they are not confounded with the code elements that have the same name. In our prototype, this demonstrative differentiation by name is not necessary and thus not performed, because generated and original classes have different packages.

The interface `OrientedGraph` and its methods are directly extracted without any differences. The classes `Edge` and `Vertex` are extracted as metaclasses omitting details for brevity. The fields and methods of the two Java classes `DirectedGraph` and `OrderedTree` lead to different extraction results: As the Java class `Identifier` is outside the extraction scope, a data type is extracted for it. Thus, an attribute for the metaclass `EDirectedGraph` is extracted for the field `id` and its accessor `getId` of the Java class `DirectedGraph`. Note that we neither display accessors nor visibilities in the metamodel, as Ecore always makes all attributes and references publicly accessible via accessors. For `EDirectedGraph`, a reference to `EEdge` is extracted for the field `edges` and its accessor `getEdges`. The same applies to the Java class `Vertex`, the field `root`, and the accessor `getRoot`. The method `getVertices` of the Java class `OrderedTree`, however, is not an accessor for a field, so it is extracted as an operation for the metaclass `EOrderedTree`.

B. Code Generation and Unification

After the metamodel has been extracted from the original Java code, model and unification code are generated to prepare the integration of the model code with the original code. To obtain the *model code*, the ordinary EMF code generator is used: For every metaclass that was extracted for a Java class, a model implementation class and a model interface are generated. We already showed the extracted metamodel for the code of our running example in Figure 3. In Figure 4, we present the resulting adapted code, model code, and unification code. We consider only a subset of the original code: the field `edges` and its accessor as well as the two classes `Vertex` and `Edge` are omitted to simplify the discussion. The generated model implementation class for the metaclass resulting from the Java class `DirectedGraph`, for example, is `EDirectedGraphImpl`, and the model interface is `EDirectedGraph`.

In addition to the ordinary model code, we also generate code to unify model and non-model code. This *unification code* consists of a unification class for every metaclass. Every unification class implements the model interface of its metaclass by delegating all model methods to the implementation class of the ordinary model code. In our running example, the unification class `UnifiedDirectedGraph` implements the model interface `EDirectedGraph` by delegating the model method `getVertices()` to `EDirectedGraphImpl`. Generating such delegation classes is just the first step of the code unification process, because it is only concerned with model code, but not with non-model code.

In the second step of the code unification process, every unification class is made a subclass of the superclass of the original class. In the example, the original Java class `OrderedTree` extends `DirectedGraph`. Therefore, the unification class `UnifiedOrderedTree` is generated as a subclass of the `DirectedGraph`. This is necessary because the unification class is inserted into the inheritance hierarchy in the last step of the Ecoreification process. In this last step, the original code is adapted, which we describe in the following.

C. Code Integration by Adaptation

As a last step of the Ecoreification process, the original code is integrated with the generated model code by adaptation, which includes the replacement of extension relations and imports, and the removal of those fields, accessors, and mutators that resulted in features of the corresponding metaclasses.

First, all extension relations between classes of the original code are replaced with extension relations to the unification classes, because the corresponding unification classes already have extension relations to these superclasses. In our running example, the extension relation between `OrderedTree` and `DirectedGraph` is replaced with an extension relation to `UnifiedOrderedTree` during the adaptation (see again Figure 4). As the unification class `UnifiedOrderedTree` already has an extension relation to `DirectedGraph`, this replacement results in an insertion of `UnifiedOrderedTree` in the inheritance hierarchy. Implementation relations between classes and interfaces as well as extension relations between interfaces in the original code are, however, not removed. Although the model interfaces define all methods of the original code interfaces, these relations have to be kept to ensure backward compatibility.

Second, the fields, accessors and mutators of Java classes that resulted in features of the corresponding metaclass are removed. These accessors and mutators are still available in the adapted Java classes as they extend the unification classes, which define these methods by delegating them to the model implementation class as described above. For example, the field `root` and its accessor in the original Java class `OrderedTree` are no longer present after the adaptation. The accessor is, however, implemented by the superclass `UnifiedOrderedTree` via delegation to an instance of `EOrderedTreeImpl`. This model implementation class obtained the accessor and the corresponding field `root` during the ordinary code generation for the metaclass `EOrderedTree`. If a method is not a pure accessor or mutator because it contains additional code, we want to preserve this additional code in the future by keeping such methods and only replacing the accessor or mutator part with a call to the appropriate model methods.

Third, imports in the model code are replaced to ensure the compatibility of the types of parameters and return values of model methods. Without this adaptation, the declared types of the parameters and return values of all model methods would be model classes and model interfaces. Therefore, we replace these model types with the types of the original code, which specialize these types after the adaptation step. The result of these replacements is, for example, that the method `getRoot` does not return an `EVertex`, but a `Vertex`.

D. Preliminary Results

Our proof-of-concept prototype realizes a part of the above mentioned steps to show the general feasibility of our approach. It is capable of extracting simple features and operations from source code into an Ecore-conforming metamodel. We applied this prototype to the abstract syntax tree of the Eclipse Java Development Tools (JDT) and extracted a valid metamodel from which valid model code can be generated. The application to the JDT also led to the identification of some of the presented challenges. It contains, for example, non-default constructors and fields that have to be represented as multi-valued features. We could not yet evaluate the integration of the original and the generated code for the JDT but will do this as soon as we finished the automation of the code integration.

To show that the concepts for integrating the code generated by EMF into existing source code are realizable, we applied our extraction prototype to two implicit metamodels in source code. They represent the families and persons metamodels, which are well-known from model transformation evaluations [11]. We generated code from the extracted metamodels and performed the steps for integrating generated and original code manually. Afterwards, we successfully applied a QVT-O model transformation, which transforms families models into persons models. This QVT-O transformation accesses the model features using Ecore functionality, which means that the integration of model and source code was successful in this exemplary scenario.

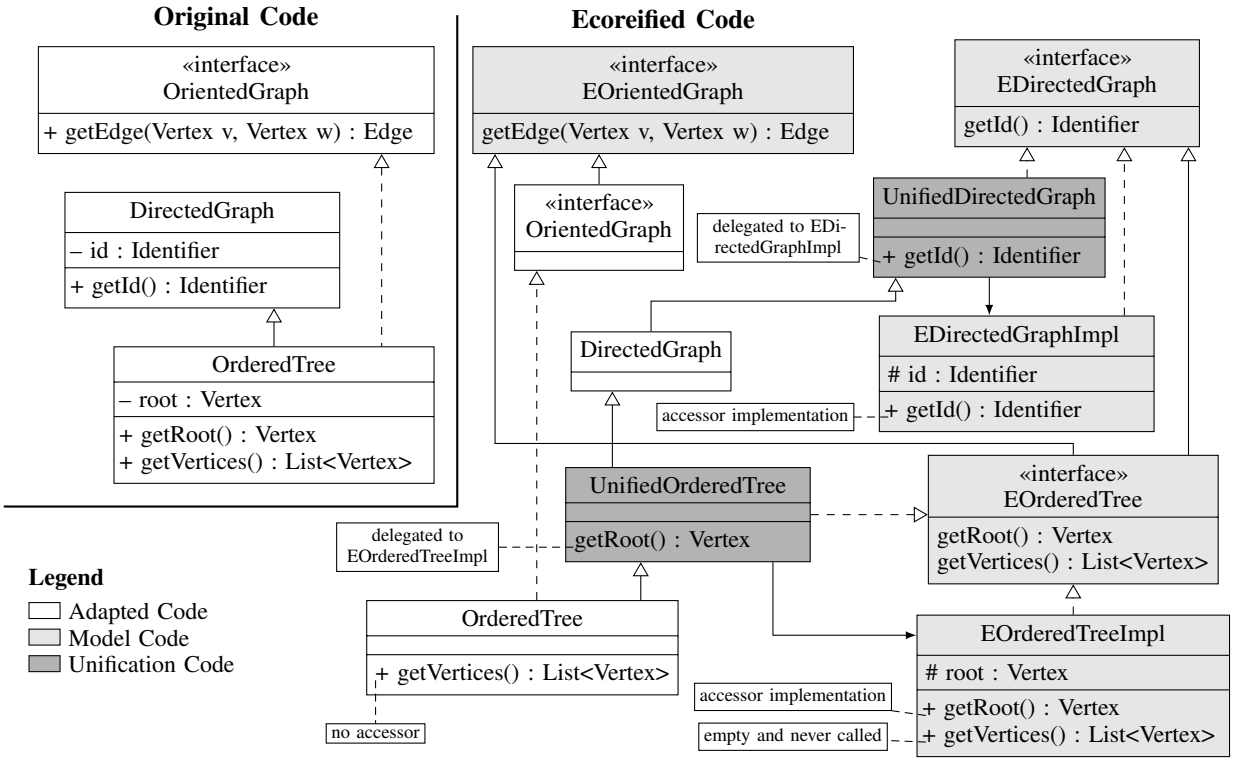

Fig. 4. Class diagram showing parts of original code and ecoreified code for two metaclasses and an interface of the running graph library example

V. Challenges

In Subsection III-C, we briefly mentioned four key challenges for extracting Ecore-conforming metamodels from Java code: i) multiplicities of features, ii) containment references, iii) non-default constructors, and iv) features to be abstracted away in models. We now discuss them in more detail.

The first challenge is to deal with *multiplicities of features*. In Ecore-conforming metamodels, users need to define the multiplicities of features, which especially determines if only a single element or a collection is referenced. Those cases are treated differently by EMF tooling. Single-valued features only allow to modify the referenced element, whereas multi-valued features allow the modification of the contents of the collections that realize them. In Java code, multiplicities cannot be defined explicitly, but are implicitly indicated by the use of collection classes or arrays. Although every type reference in Java code could be interpreted as a single-valued reference, extracting references to collection types as single-valued references would prohibit the ability to modify the collection contents by EMF tools. Therefore, it is necessary to extract multi-valued features from source code if possible. Since most collections used in Java code implement the `Collection` interface, references to classes implementing that interface can be interpreted as multi-valued references. In the code generated by EMF, all multi-valued features are represented using the Ecore collection implementation `EList`. Ecore also offers the option to mark lists as unique, which can be used to represent a `Set`. For other collection types, such as stacks or queues, `EList` is not appropriate, as it does not offer necessary methods such as `pop` and `push`. To overcome this challenges, it is possible to create counterparts for those collection types in Ecore.

The second challenge is to deal with *containment references*. In Ecore-conforming metamodels, references need to be modeled explicitly as containment references if a class should be contained within this reference. If an instance of a metamodel is saved using the standard Ecore mechanism, each object within the instance needs to be referenced by one containment reference. In source code, especially in languages with a garbage collector such as Java, it is not necessary to provide an explicit containment for objects. Objects in Java are garbage collected if they have no incoming reference, i.e., every hard reference to a Java object can be considered a containment reference. Hence, it is difficult to extract containment references from source code. Although static code analysis can identify objects that are definitely contained in only one other object, this does not work for all objects. We therefore propose the following approach: The extracted metamodel has an explicit container element, in which all model elements will be contained. Upon insertion of an element into the model, detected using the EMF notification mechanisms, it has to be inserted into that container as well if necessary. Upon removal from the last reference, the element has to be removed from that container. This initial approach can be enhanced with static code analysis and heuristics to identify containments.

The third challenge is to extract and to deal with non-default constructors. Having a default constructor is necessary, because EMF tooling uses default constructors to create instances of

classes. Ecore, furthermore, does not permit the creation of constructors for metaclasses. Java classes, however, often have parametrized constructors and do not offer a default constructor. To deal with non-default constructors in the extraction process, we propose the following approach: If a default constructor does not exist for a class that needs to be extracted, we first create a default constructor. For each non-default constructor, we can automatically create an initialization method. This initialization method contains the code of the non-default constructor and can be called from the appropriate constructor as well as from code created with Ecore tooling. This approach has the advantage that already existing code does not need to be changed, because the existing non-default constructors are still offered.

The last of the four challenges is to only select those metaclasses and features that should be part of an *abstract* model from all metaclasses and features that could technically be extracted. Abstraction is a central goal of modeling and therefore the potential advantage of using a metamodel that was automatically extracted from the source code may strongly depend on what is *not* part of the models. We envision an interactive process in which the user of the extraction tool can define the desired level of abstraction by selecting metaclasses and features. The result of not selecting a metaclass would be the same as described above, i.e. a data type that links to the original Java class. If a feature is not extracted, then the result has to be that the field as well as the accessor and optional mutator method remain untouched in the original class and neither appear in the model class nor in the model interface. To support the user during this decision, especially if the number of extractable elements is high, we propose to analyze for every metaclasses and for every feature how often it would be used in the code that corresponds to the current extraction scope.

In addition to these fundamental challenges, further challenges such as the extraction of bidirectional references or more restrictive multiplicities could be addressed in the future, but are not mandatory for the applicability of our approach.

VI. Related Work

The integration of Ecore functionality into existing source code has, to the best of our knowledge, not been explored yet. In Subsection III-B, we already mentioned approaches that try to reduce the gap between modeling and programming, like Xcore [8] and Annotated Java [3]. Such tools do, however, only support the development of *new* projects, but not the integration of *existing* source code.

Most related to our approach is the extraction of UML models from source code. This research field is also called *UML reverse engineering* and a survey on it was conducted by Kollmann, Selonen, Stroulia, *et al.* [12]. Challenges that we presented, such as the extraction of reference multiplicities or the specification of an appropriate extraction scope, are also relevant in this context. A generic framework for reverse engineering different types of UML models, like class diagrams or interaction diagrams, was proposed by Tonella [13]. A difference to our work is that the extracted UML models are only a more abstract representation of the source code. In our approach, code must be generated from these models and integrated into the existing code, which requires a specific structure for the extracted models that is not needed for UML reverse engineering.

UMPLE [14] is an approach for combining modeling and programming based on the UML. It is a specialized programming language that can be used to combine textual definitions of several UML concepts, such as associations with multiplicities, with ordinary source code. In consequence, it raises the abstraction of programming languages for these concepts to a modeling level. The developers of UMPLE also presented reverse engineering approaches for integrating existing source code into UMPLE [15], [16]. In contrast to our approach, UMPLE does, however, not integrate source code into a modeling framework so that existing tools can be reused. Instead, the integration target is the UMPLE language which can be compiled to plain Java code. Furthermore, this language extends ordinary programming languages, so the complete original code can be translated to UMPLE, which is not possible when extracting Ecore-conforming metamodels.

VII. Conclusion

In this paper, we have presented the Ecoreification idea for integrating the functionality of Ecore-conforming metamodels into source code that contains implicit metamodels. This functionality is presumed by model-driven tooling and comprises, for example, uniform access to metaclass features. With our approach, it is possible to apply model-driven tools, such as transformation languages or graphical editor frameworks, to existing source code that was not developed using explicit metamodels, and from which the required functionality is automatically integrated into the generated source code.

We have proposed a three-step process for integrating Ecore functionality into existing code: First, an explicit metamodel that represents the source code structure has to be extracted. Second, code has to be generated from that metamodel, which contains the functionality required by model-driven tools. Third, the generated code has to be integrated into the existing code to provide the same API as before, but enriched with the functionality of the extracted Ecore-conforming metamodel.

We have presented a proof-of-concept for the metamodel extraction and for integrating generated model code with existing code to show that our approach is applicable. It was possible to extract metaclasses with attributes and references for all accessible fields. This way, existing code was made accessible to model-driven tools while keeping its original API.

To encourage further research, we have identified key challenges, which have to be addressed in the future to close the gap between existing source code and model-driven tooling. Those challenges comprise the handling of parametrized constructors, the realization of containments and the extraction of multiplicities. We have presented our plans for solving these challenges, but could not yet completely validate their technical effectiveness. In future work, we will realize the Ecoreification approach and validate it using open-source projects such as the already mentioned graph library Apache Flink Gelly.

References

[1] Object Management Group (OMG), *Model Driven Architecture - Specifications*, 2006.

[2] A. Forward and T. C. Lethbridge, "Perceptions of software modeling: a survey of software practitioners," School of Information Technology and Engineering, University of Ottawa, Tech. Rep. TR-2008-07, 2008.

[3] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, second revised, ser. Eclipse series. Addison-Wesley Longman, Amsterdam, 2008.

[4] Object Management Group (OMG), *MOF 2.5 Core Specification (formal/2015-06-05)*, 2015.

[5] ——, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.3*, 2016.

[6] S. Efftinge and M. Völter, "oAW xText: A framework for textual DSLs," *Proceedings of Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[7] T. E. Foundation. (2017). Sirius - the easiest way to get your own modeling tool, [Online]. Available: https://eclipse.org/sirius/ (visited on 2017-04-13).

[8] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[9] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, R. Von Massow, and M. Hanus, "Xbase: Implementing Domain-specific Languages for Java," *ACM SIGPLAN Notices*, 48, no. 3, pp. 112–121, 2013.

[10] T. Sağlam, "Automatic Integration of Ecore Functionality into Java Code," Bachelor's Thesis, Karlsruhe Institute of Technology (KIT), 2017, p. 76.

[11] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Springer Berlin Heidelberg, 2012, pp. 399–437.

[12] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, Institute of Electrical and Electronics Engineers (IEEE), 2002, pp. 22–32.

[13] P. Tonella, *Reverse Engineering of Object Oriented Code*. Springer New York, 2005.

[14] M. A. Garzón, H. Aljamaan, and T. C. Lethbridge, "Umple: A framework for model driven development of object-oriented systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 494–498.

[15] T. C. Lethbridge, A. Forward, and O. Badreddin, "Umplification: Refactoring to incrementally add abstraction to a program," in *2010 17th Working Conference on Reverse Engineering*, 2010, pp. 220–224.

[16] M. A. Garzón, T. C. Lethbridge, H. Aljamaan, and O. Badreddin, "Reverse engineering of object-oriented code into umple using an incremental and rule-based approach," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '14, IBM Corp., 2014, pp. 91–105.