Applying Metamodel-based Tooling to Object-oriented Code

Heiko Klare, Timur Saglam, Erik Burger and Ralf Reussner
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Germany

Keywords: Model-driven Engineering, modelling – Programming Gap, Reverse Engineering, Model Extraction.

Abstract: Model-driven development processes mainly from the capabilities of modelling frameworks, since these offer an *explicit* representation of model elements and features, as well as reusable tooling, such as transformation languages or editor frameworks. Nevertheless, most software systems are not developed in a model-driven way although they do contain *implicit* models encoded in their object-oriented design. Adaptation to model-driven tooling imposes high manual effort and easily breaks compatibility with dependent code and existing tooling. We present an automated and minimally intrusive approach that makes implicit models in software systems explicit. We adapt existing object-oriented code so that it provides an explicit model representation, while preserving its original API. As a result, capabilities of modelling frameworks, such as persistence and change notifications, can be integrated into object-oriented code, and enable the application of reusable tools. We provide a classification of requirements that existing code has to fulfill, usable as an indicator for the applicability of modelling tools to them. An evaluation based on one artificial and two open-source case study systems shows the correct preservation of the API, as well as the ability to apply tooling to the modified code.

1 INTRODUCTION

Models are commonly used to encapsulate knowledge of a certain domain, so that further artifacts can be derived from them, e.g., program code (Object Management Group (OMG), 2006). Developers use modelling frameworks to define *explicit* domain metamodels. Hand-written object-oriented program code also contains *implicit* domain metamodels in the form of domain knowledge that is encoded in the object-oriented design. This implicit definition obstructs the reuse and evolution of program code. For example, a graph library may define interfaces and classes for vertices, edges, and their relations. If this library is extended to support further algorithms, which requires converting the graphs into a representation of another library that provides those algorithms, this must be implemented in verbose and hand-written code. The same is true if a visual editor for creating and modifying graphs is added. If a graph metamodel is defined explicitly using a modelling framework, developers can apply *metamodel-based tooling*, such as transformation languages and graphical editor frameworks.

In general, modelling frameworks provide two essential benefits. First, they provide *abstraction* by encapsulating domain knowledge in metamodels of classes, their properties, and operations. Hand-written code often provides no clear separation between domain metamodels and, for example, infrastructure logic and utilities. Second, a modelling framework induces a *formalism* for defining metamodels and a mapping to a representation in program code. Generators produce code that provides a specifically structured *application programming interface (API)*, i.e., a set of interfaces with specific methods providing well-defined functionality, which can be seen as a contract on which metamodel-based tools can depend. This API especially provides a specific way to create model elements, access their features, and persist them. Hand-written code does usually not provide such a specific API, but at most follows design conventions.

To enable the application of metamodel-based tools to implicit metamodels, the gaps of missing *abstraction* and *formalization* have to be closed. First, domain metamodels have to be identified in code, either by manual element selection, or by model extraction heuristics. Second, the code has to be modified to provide the API required by the modelling framework, which, if manually performed, is costly and error-prone, and easily breaks its original API.

In this paper, we present an approach that automatically adapts existing code containing an implicit metamodel, so that it preserves its original API, but also provides the API required by a modelling framework. The proposed solution extracts a domain metamodel from existing code, executes the code generator of the modelling framework, and integrates the existing with the newly generated code. It reuses both the original code to maintain the existing API and the existing code generator to avoid a reimplementation of the genera-

Klare, H., Saglam, T., Burger, E. and Reussner, R.
Applying Metamodel-based Tooling to Object-oriented Code.
DOI: 10.5220/0007246202170228
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 217-228
ISBN: 978-989-758-358-2

tion patterns. The benefit of this approach is that it enables developers to apply metamodel-based tools to implicit metamodels in program code. It closes the *formalization* gap and eases the shift from code-centric to model-driven development, which is necessary to achieve a common adoption of such processes (Meyerovich & Rabkin, 2013).

We presented the initial approach idea in prior work (Klare et al., 2017). In this paper, we complete the concept, discuss solutions to identified challenges and provide an evaluation based on an implementation for the Eclipse Modeling Framework. We make the following contributions to the topic of applying metamodel-based tools to object-oriented code:

Concept Completion: We provide an advanced concept on how to extend code to enable the application of metamodel-based tools, while preserving its API. The concept minimizes code modifications, reuses the code generator of the modelling framework, and contains solutions to previously identified challenges (Klare et al., 2017).

Requirements Classification: We provide a classification of problem-inherent requirements that the code has to fulfill to enable the application of metamodel-based tooling on it. The requirements are supposed to be an indicator whether such tooling is applicable to a specific system.

Applicability Evaluation: We evaluate the applicability of the approach by applying it to one artificial and two existing open-source systems, all three containing domain metamodels. It shows the correct preservation of code behavior and the introduced applicability of metamodel-based tooling by taking the example of a graphical editor.

We present our concepts on a simplified employee management system. The structure of the code representing a domain metamodel is depicted in Figure 1. It consists of an interface for employees, one class for ordinary employees, and one for managers. They have features for their salary, supervisor and position, which are accessible by appropriate methods. Additionally, each employee has a method to get fired.

2 TERMINOLOGY & FOUNDATIONS

Metamodel Formalisms. Models are based on an implicitly or explicitly defined *metamodel formalism*, which defines the elements a model can contain and how they can be connected. Each model is based on a metamodel, which in turn has its own metamodel. The topmost, self-describing metamodel is defined by the

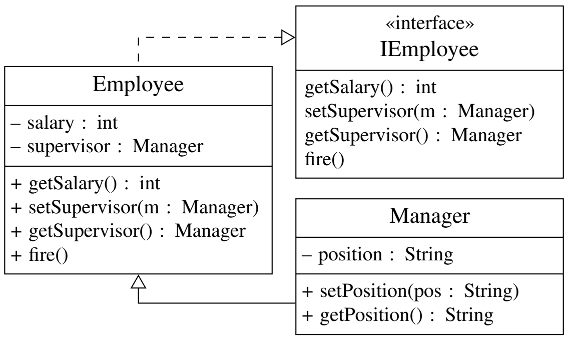

Figure 1: Simplified employee management system.

metamodel formalism, such as OMG's *Meta-Object Facility (MOF)* (ISO/IEC 19508:2014(E), 2014). We assume a metamodel formalism that conforms to the *Essential MOF (EMOF)* and has four modelling levels. We refer to the model on the topmost level as the *metametamodel*, to models on the underlying level as metamodels and to those on the third level as models.

Modelling Frameworks. A *modelling framework* is built on top of a metamodel formalism and defines how models of that formalism can be used. It provides reusable tools, such as editor frameworks or transformation languages, in the following referred to as *metamodel-based tools*. A framework especially includes a *code generator* that derives object-oriented code from a metamodel, the so called *metamodel code*. The generated code provides a specific *API*, which defines a contract on which metamodel-based tools can rely, e.g., for instantiating elements or accessing features. We refer to it as the *metamodel code API*.

Eclipse Modeling Framework. The Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) is a modelling framework, which provides the EMOF-conform metametamodel *Ecore*. It defines packages, classes, interfaces, operations and features (attributes and references). Metamodels define explicit containment references, which induce a tree hierarchy of model elements. The code generator maps metamodel packages and interfaces to their Java counterparts. It maps a metamodel class to an interface and a *metamodel implementation class* with the suffix "Impl". Features are mapped to instance variables with *accessors* (getters) and *mutators* (setters). Additionally, a factory class is generated for each metamodel package, providing methods that call the non-parametrized constructor of each metamodel implementation class in that package. Several metamodel-based tools have been developed for EMF, e.g., transformation languages such as QVT-O, textual editor frameworks like Xtext, or graphical editor frameworks such as Sirius.

3 Ecoreification

In this section, we discuss implicit domain metamodels and how to make them accessible to metamodel-based tooling.

3.1 Implicit Domain Metamodels

Software consists of data and behavior that operates on it. The underlying domain metamodel structures the data, in most systems implicitly represented in an object-oriented class design and not denoted according to a metamodel formalism. If metamodel-based tooling for a specific modelling framework shall be applied to code, that code has to provide the metamodel code API for that framework, on which the tools rely (see Section 2). This especially includes the provision of certain factories, feature accessors and mutators. Furthermore, certain information provided by an explicit metamodel is only contained implicitly in object-oriented code, such as features, usually encoded by accessors and mutators, or their multiplicities.

Existing code will usually not conform to the metamodel code API that the code generator of a modelling framework produces for a metamodel. Therefore, it is necessary to modify the code if metamodel-based tools shall be applied. That modification must fulfill two central properties. First, it must be *API preserving*, so that dependent code still works properly. Second, it must be *API extending* in such a way that the modified code provides the additional metamodel code API of the modelling framework. To perform those modifications, two essentials solution options, *reimplementation* and *refactoring*, can be identified. In a reimplementation approach, an explicit metamodel representing the original implicit domain metamodel has to be defined and enriched with the original logic. This option requires high manual effort as dependent code must be adapted to the new API generated from the metamodel. In a refactoring approach, the existing code has to be adapted such that the existing API is preserved but also extended with the API that would have been generated from a metamodel in a modelling framework. Extending the API also requires high manual effort and especially extensive knowledge about the structure of the metamodel code API to provide.

3.2 A Hybrid Approach

To circumvent the drawbacks of the two solution options, we present a hybrid approach that combines both in two steps. In a first step, an explicit metamodel is extracted from the implicit domain metamodel of the existing code in a way that the code generated from it can later be integrated with the existing code. It uses the code generator of the modelling framework to generate metamodel code from it. In a second step, we adapt the existing code by integrating the metamodel code generated in the first step. We therefore generate so called *unification classes*, which are inserted into the original inheritance hierarchy and delegate access of extracted metamodel features to the metamodel implementation classes. Using such an integration scheme instead of moving all logic from the existing to the generated metamodel code ensures that only necessary parts, especially the features, are extracted to the metamodel and the rest of the logic remains in the existing code, which eases its further evolution. The benefit of this approach is that it reuses both the original code, preserving its API, and also the existing code generator to add the metamodel code API.

We explain these two steps in detail in the subsequent sections. The complete process is depicted in Figure 2. It delivers three artifacts that together form the final code, which are *metamodel code* generated from the extracted metamodel, *unification code* that combines the metamodel code with the original code, and *adapted original code*, which is the original code with adaptations that integrate the unification and metamodel code. For ease of understanding, we explain the concept on EMF and Ecore, on which the implementation is based. This is the reason why we call it *Ecoreification*. Nevertheless, the overall concept can be used for other modelling frameworks.

3.3 Addressed Challenges

In addition to the complete approach, we will discuss solutions to the challenges identified in our previous idea presentation (Klare et al., 2017):

Multiplicities: To correctly handle element collections in a model instead of handling the complete collection as one immutable element, they have to be represented as multi-valued features.

Containment: Ordinary Java code provides no containment relations as required in Ecore models, but automatically removes elements that are not references anymore using a garbage collector.

Non-parametrized Constructors: Classes in the metamodel code must have non-parametrized constructors, as they are instantiated by generated factories (see Section 2). Classes in existing code do not necessarily provide such constructors.

An orthogonal challenge is abstraction. It concerns the identification of domain-relevant information to be extracted from code. Since this is an independent research topic, we assume a manual selection of classes and features or the usage of an existing approach.

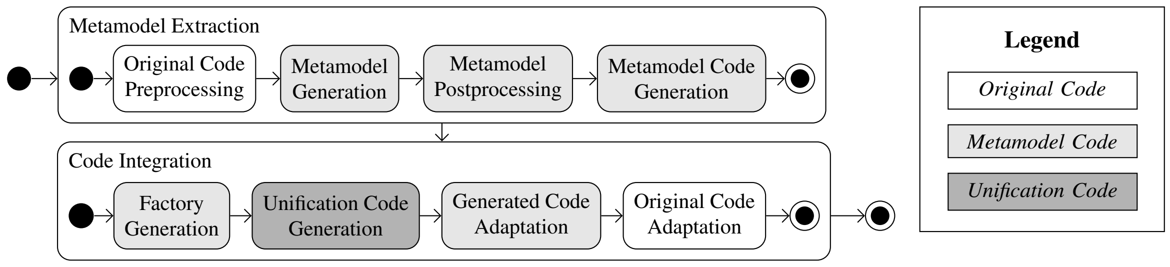

Figure 2: The essential steps of the Ecoreification process, each affecting one of three involved artifacts.

4 METAMODEL EXTRACTION

In a first step, our approach extracts a metamodel from its implicit representation in code. It is comparable to reverse engineering UML models, but with an additional requirement. The code generated from the metamodel has to be integrable with the existing code according to the scheme presented in the subsequent section. Since the extraction was discussed in detail previously (Klare et al., 2017), we only give an overview and focus on additional pre- and postprocessing, as shown in the process in Figure 2.

The mapping of Java elements to Ecore elements is summarized in Table 1. Most elements like classes, interfaces and their inheritance are directly mapped to the equivalent Ecore elements. External type references, including types of external libraries, can be modeled as data types in Ecore (Steinberg et al., 2008). Nested types are not supported by Ecore and thus represented as types in a subpackage. Each field is transferred to an attribute or reference. Accessors and mutators are identified by an appropriate signature that follows the standard Java naming scheme having the prefixes “get”, “set” or “is” followed by the feature name. If a feature is of type List, it is interpreted as a multi-valued feature of the list content type, otherwise it is treated as a single-valued feature. The handling of non-list collections is discussed in Section 7.2. Remaining methods are extracted as operations.

The metamodel generation and the metamodel code generation steps apply the above introduced patterns and run the ordinary code generator of EMF. Applying the metamodel extraction to our employee management system example produces the metamodel shown in Figure 3. It contains the extracted attributes salary and position, as well as the reference supervisor. The generated metamodel code is shown in Figure 4. The prefix “E” is only added to the types to clearly separate them from the original types. In the implementation, they are placed in separate packages instead. In the following, we discuss necessary preprocessing of the original code and necessary postprocessing of the extracted metamodel.

Table 1: Mapping of Java elements to Ecore elements.

Java Element	Ecore Element
Top-level Type	Top-level Type
Generic Type Parameter	Generic Type Parameter
External Type Reference	External Data Type
Nested Type	Type in Subpackage
Inheritance	Inheritance
Field of Type List	Multi-valued Feature
Other Field	Single-valued Feature
Accessor & Mutator	Feature Access
Other Method	Operation

4.1 Preprocessing

Before applying the metamodel extraction, some semantics-preserving preprocessing of the existing code is necessary. Features are extracted to the metamodel and will later be removed from the original code and provided by the generated metamodel code through appropriate accessors and mutators. Therefore, access to features has to be performed through accessors and mutators instead of directly accessing a field containing the feature value. To ensure this, we perform a field encapsulation that replaces all direct field references with appropriate method calls.

To be able to integrate the metamodel code into the existing code, all top-level classes are made publicly visible. Although this potentially exposes classes that were only internally visible on purpose, this is a necessary requirement to apply our approach and, additionally, would also be necessary when manually extracting the domain metamodel to an explicit metamodel, as only public types are allowed there.

Finally, all Java classes in the generated metamodel code representing a class of the metamodel must have a non-parametrized constructor to be instantiable by the generated factories used by tooling (see Section 2). Therefore, we add a non-parametrized constructor to all classes that do not have one yet. This initially solves the *Non-parametrized Constructors* challenge. The existing constructors remain in the class and can

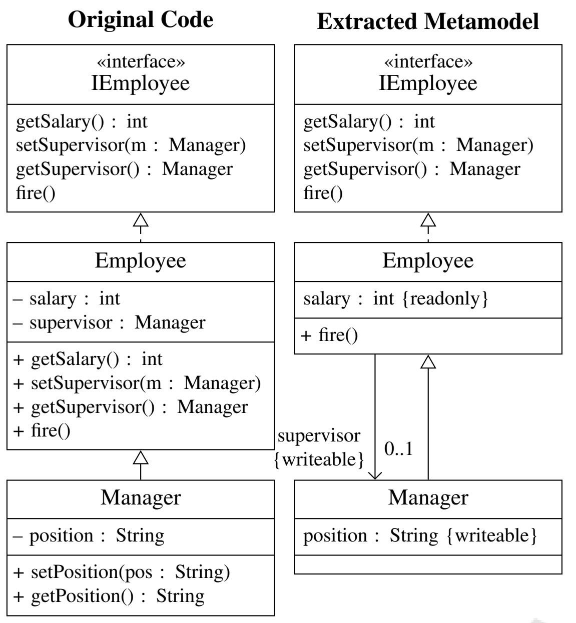

Figure 3: Original code structure and extracted metamodel.

still be called from dependent code.

4.2 Postprocessing

We finally apply a postprocessing to the extracted metamodel. Ecore requires each element to either be contained within another element or to be the model root element. Hand-written code usually has no such containment hierarchy. Objects can be referenced by any other object and have no unique container. We therefore add a root container class to the metamodel, which artificially contains all model elements that can then be referenced from any other element. This can also be enhanced with a monitor on instances of the metamodel that automatically tracks additions of references within the model and automatically adds elements to the root container upon initial referencing (and removes them vice versa). Our evaluation shows that this is a suitable solution for the *Containment* challenge.

5 CODE INTEGRATION

The integration of original and generated metamodel code is split into four steps, as depicted in Figure 2. We first generate new factories that instantiate the original classes instead of those of the generated metamodel code. Then unification code is generated that combines the original and the metamodel code and finally original and metamodel code are adapted, so that the metamodel code uses the original types and the unification code is integrated into the original inheritance hierarchy. Figure 4 shows the result of the integration process for our running example.

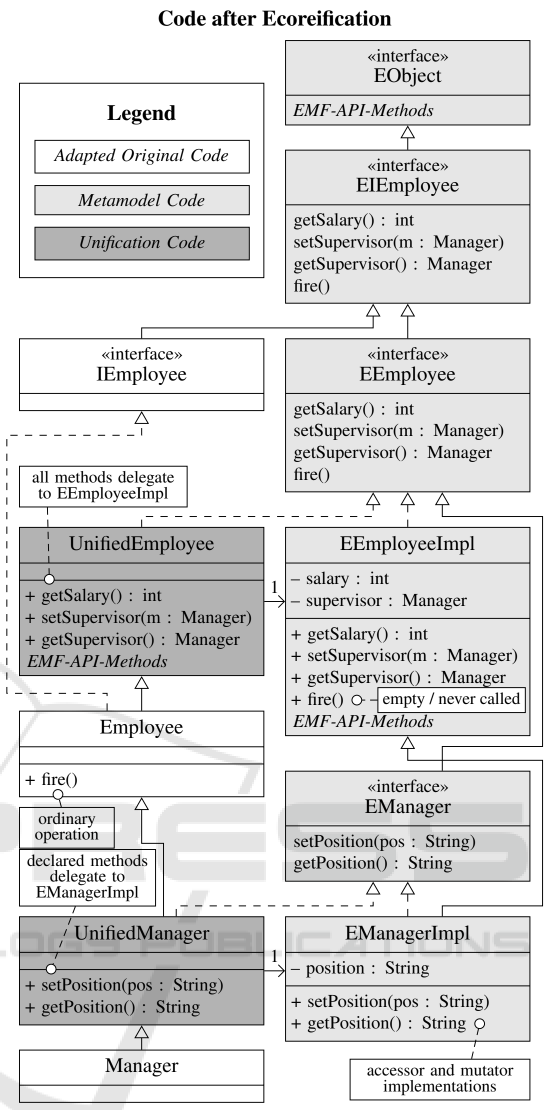

Figure 4: Code structure after applying the Ecoreification.

5.1 Factory Generation

The EMF code generator produces a factory for each metamodel package. Each factory method's return type is the interface of one metamodel class, e.g. EEmployee. It internally instantiates the metamodel implementation class, e.g. EEmployeeImpl. We replace the factories with ones that provide the same interface but instantiate the original classes, e.g. Employee, instead of the metamodel class. This is type conforming, as the original classes also implement the metamodel class interfaces after the code adaptation. The old factories are moved to be reused in unification code.

5.2 Unification Code Generation

Unification classes represent the glue code between the original code and generated metamodel code and can be seen as an implementation of the generation gap pattern (Fowler & Parsons, 2010). For each original class, one unification class is created and inserted into the original inheritance hierarchy. It delegates access to features extracted to the metamodel to instances of the metamodel implementation classes. This is less intrusive than integrating the delegation into the original code, as the original classes stay mostly unmodified.

We explain the generation and insertion of a unification class on the example of the Manager class. It is created as a subclass of the original class, thus UnifiedManager becomes a subclass of Employee. It implements the interface of the appropriate metamodel class, in the example the EManager interface. A field is added that references an instance of the corresponding metamodel implementation class. For example, UnifiedManager references an EManagerImpl object. It is instantiated in the constructors using the original factory, which was moved in the first integration step. The constructors have to be the same as those of the original superclass and just delegate to them.

The central functionality of a unification class is the delegation of feature access to the metamodel implementation class. Therefore, it implements all methods of its metamodel interface, in the example the accessor and mutator for the position feature. All unification classes delegate exactly the methods declared in the interface of the metamodel class, e.g. the accessor and mutator for position in the UnifiedManager. Other methods must not be delegated, as, for example, delegating the method fire in UnifiedManager would overwrite its correct implementation in Employee. The unification class for the topmost class in an inheritance hierarchy, such as Employee, additionally delegates methods induced by the framework, denoted as *EMF-API-Methods* in Figure 4. In EMF, those methods are declared in the EObject interface and implemented by all metamodel classes and define additional logic for, e.g., resource handling. Generic type parameters are adopted from the original class. The type parameters from the original class were transferred to the metamodel, thus the type bounds in the metamodel interface and the original class, between which the unification class is placed, are equal.

5.3 Metamodel Code Adaptation

The generated metamodel code initially uses the metamodel types for features and therefore for the appropriate method parameter and return types of accessors and mutators. For example, the EEmployee interface initially provides an accessor and mutator for the EManager type. In contrast, the original code uses the original types, in the example accessors and mutators for the Manager type. This means that the parameter and return types are covariant, as the specialized original types also use the specialized original types. While this is unproblematic for return types, only contravariant parameter types are type safe. For this reason, the metamodel code adaptation replaces all parameter and return types in the metamodel interfaces and classes with the original types. For multi-valued features, this only concerns the list content types. Figure 4 shows the code with that type adaption for our example.

5.4 Original Code Adaptation

To finally integrate metamodel and unification code into the original code, two adaptations of the original code are performed. The inheritance relations of the original code are modified and the logic of extracted features is removed from the original code.

The modification of inheritance relations comprises both classes and interfaces. The inheritance of a class is changed to its individual unification class. Since the unification classes inherit the original superclasses instead, this simply inserts the unification classes into the inheritance hierarchy. For example, the UnifiedManager class is inserted into the original inheritance hierarchy between Manager and its superclass Employee. Interfaces have no unification classes but also have associated interfaces in the metamodel code. To make an interface usable in place of its metamodel counterpart, it has to inherit its appropriate metamodel interface. In the example, the IEmployee interface therefore inherits the EIEmployee interface.

Features that were extracted into classes of the metamodel are provided by the metamodel implementation classes. Each original class has a delegation of all feature accessors and mutators through its unification class. Therefore, the code adaptation removes all fields, mutators and accessors that are replaced by metamodel features in the original code. In the example, all fields and methods of the original classes Employee and Manager are removed, except for the method fire, which is an ordinary operation and therefore preserved in the original code. This process changes the return types of accessors for multi-valued features. They originally returned instances of the List interface, whereas the generated metamodel code uses instances of the more specialized EList interface. Nevertheless, this does not break the API, as covariant replacements of return types are type conforming, and thus solves the *Multiplicities* challenge.

6 EVALUATION

We conducted three case studies to show the applicability of our approach. We applied it to an artificial employee management system, used as the running example, to an open-source graph library[1] with an implicit graph metamodel, and to an established open-source community case study, introduced in the subsequent subsection. The evaluation has two primary goals. First, we demonstrate the API-preserving property by showing that the code maintains its behavior. Second, we show the applicability of metamodel-based tools to the modified code. The evaluation is based on our implementation for EMF, available on GitHub[2].

The approach must ensure API preservation both on a syntactic and on a semantic level. Syntactic conformance with the original code is checked by compiling the modified code and all dependent code that uses its API. Semantic correctness is given, if the behavior of the system is not altered. Since this property cannot be proven easily, we execute the code and test its correct behavior with test cases and predefined usage scenarios. To show the applicability of metamodel-based tools on the modified code, we exemplarily apply an established EMF tool on it. We employ the graphical editor framework Sirius to develop a graphical editor for the creation of instances of the domain metamodels and use it to create example models.

6.1 CoCoME Community Case Study

The Common Component Modelling Example (CoCoME)[3] is a community case study system for comparing component-based software development approaches (Herold et al., 2008). It implements a trading system comprising an implicit domain metamodel with companies, products, stores, stock items and more. We depict an extract of those types considered in our evaluation in Figure 5, based on the data model from the CoCoME documentation (Herold et al., 2008, p. 19). CoCoME was extended in several ways. We use the so called *platform migration scenario* for our evaluation (Heinrich et al., 2016, p. 14), which migrates CoCoME to a Java-EE-based cloud infrastructure. Herold et al. (2008) additionally defined usage scenarios, which we use to show semantic correctness of the modified code.

6.2 Results

We applied our approach to all three case study systems. The resulting code with example tools and

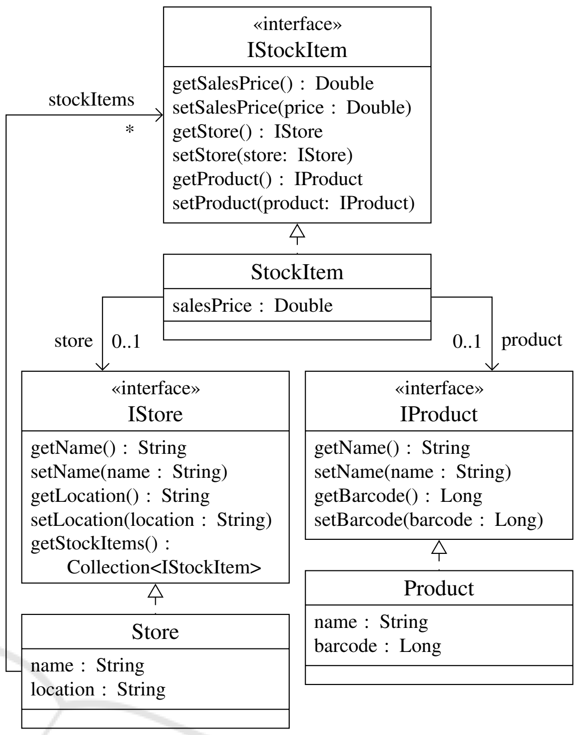

Figure 5: Implicit CoCoME domain metamodel extract.

models developed for the evaluation can be found on GitHub[4]. We had to make small modifications to the graph library, because some extracted interfaces have implementations that we did not extract to the metamodel. This results in missing method implementations in those non-extracted classes, because they finally implement the `EObject` interface and thus have to provide its methods. We manually added empty implementations of those methods to circumvent that problem, which we further discuss in Subsection 7.1.

API Preservation. All three systems compiled and deployed successfully after applying the Ecoreification. While the employee management system contains only the domain metamodel without dependent code, CoCoME consists of several projects and a complex infrastructure, which is deployed on multiple application servers in a Maven-based build process.

To show semantic correctness of the modified code, we defined test cases for all projects, in which we instantiate classes of the domain metamodel, modify their features and call their operations. As this only gives an initial indication for correctness, we also executed the existing test cases of the graph library and the use cases defined for CoCoME, which succeeded and resulted in the expected system states. We assume

[1]http://graphstream-project.org/
[2]https://github.com/Ecorification
[3]https://github.com/cocome-community-case-study

[4]https://github.com/Ecorification/CaseStudy-EmployeeManagement, https://github.com/Ecorification/CaseStudy-CoCoME, https://github.com/Ecorification/CaseStudy-gs-core

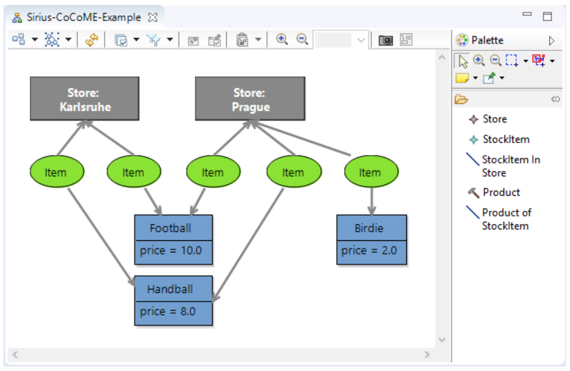

Figure 6: Example model in the CoCoME Sirius editor.

that those use cases reflect typical system usages and call most of its logic, so that their successful execution is a reliable indicator for correct system operation.

Metamodel-based Tool Applicability. We extended the test cases defined for initially demonstrating semantic correctness of the API to persist and reload the created domain models using the persistence API of EMF. To achieve that, we had to create a virtual container, as explained in Subsection 4.2, and to insert the domain model elements into that container. The models could be correctly reloaded, which shows that features were restored and thus correctly extracted, as well as that the persistence API itself is correctly provided by the modified code.

We defined a graphical Sirius editor for each of the domain metamodels to show the applicability of metamodel-based tooling to the modified code. The CoCoME editor enables the user to create products, stores and stock items with their properties and relationships, as shown in Figure 5. We successfully developed example models with the editors and loaded the models created by our initial test cases. Figure 6 shows the CoCoME editor with a simple model. To validate the correct operation of the model change notification mechanism of EMF, we performed modifications in different editors, our Sirius-based one and the EMF tree editor, as they update their contents upon external modifications using those notifications.

Summary. We were able to show the applicability of our approach by conducting typical usage scenarios in different case studies with two existing open-source systems. We applied original use cases as well as use cases based on an exemplary metamodel-based tool to demonstrate the correct operation of the modified code in all scenarios. Nevertheless, the applied tool only represents one exemplary use case of the approach. The actual benefit of the approach arises from the variety of provided mechanisms, such as persistence and change notification, and of available tools.

6.3 Validity Threats

Our evaluation revealed the general feasibility and applicability of the approach, yet we identify and discuss validity threads in this section. One threat to external validity is the limited set of scenarios. By using two open-source projects that were not developed specifically for our approach and by applying predefined usage scenarios to it, we aimed to mitigate that threat and get as generalizable results as possible. For that reason, we assume transferability of our results to other scenarios. Furthermore, we focused on correctness and applicability of the approach but did not yet validate usability properties.

Regarding internal validity, the significance of both evaluation aspects, the API preservation and the applicability of metamodel-based tooling, can be challenged. We did not provide a proof for the preservation of code semantics when applying the Ecoreification. As formally proving semantic correctness is difficult, we found our approach to apply test cases and predefined usage scenarios the best way to assure preservation of semantics. Apart from that, we already defined the steps of the approach carefully in a way that they, to the best of our understanding, preserve code semantics, as they only use semantics-preserving refactorings.

Finally, graphical editors are only one example for metamodel-based tools. It is possible that there are reasons why other tools or scenarios may not operate properly with our approach. Nevertheless, we selected the tool and the scenarios carefully, so that, to the best of our knowledge, the complete EMF API, consisting of feature access, persistence and notifications, is successfully used. The editor framework and our scenarios require proper operation of this complete API, which is why we assume generalizability of the results.

7 DISCUSSION

In the following, we discuss the limitations of our approach that we are aware of. They can be separated into requirements that have to be fulfilled by the existing code and are inherent to the problem, and challenges that can be solved by extending the approach. We finally also discuss how our approach supports further evolution of the code and its generalizability.

7.1 Application Requirements

The first category of limitations are inherent to the problem and cannot be resolved by extending the approach. Instead, they imply requirements to the code, which, if necessary, have to be fulfilled by refactorings.

Code Isolation. The approach has to assume that the code it is applied to is isolated in two ways, which we discuss in the following. First, dependent code is only allowed to use its publicly exposed interface, and second, dependent code must not directly implement interfaces that are extracted to the metamodel. Although these requirement may be difficult to fulfill, they are induced by the structure of the metamodel code and are therefore unavoidable.

If dependent code references internal representations, such as protected feature values, it may become incompatible when applying the approach. This is due to the reason that features get encapsulated, so that they are only used via accessors and mutators, as this is the way access is encapsulated in the generated metamodel code. This requirement is obsolete if all fields are already encapsulated before applying the approach or if all dependent code also gets refactored.

Interfaces of the metamodel code extend the EObject interface, which provides additional methods that were originally not implemented by classes realizing the interface. So the first non-abstract class of an inheritance hierarchy that implements the interface has to implement those methods. If that class is not extracted to the metamodel and thus not modified by our approach, because it is excluded or part of an external project, it gets inconsistent. Solving this problem by excluding all interfaces from the Ecoreification that are implemented in dependent projects or by including dependent projects into the Ecoreification is only possible if all external dependencies are known. Otherwise, the only solution is to update all dependent projects afterwards and amend their implementation, which is what we have done in the evaluation for the graph library project (see Section 6).

Non-parametrized Constructors. The handling of parametrized constructors was already discussed in Subsection 4.1. Nevertheless, the initial solution to simply add non-parametrized constructors if they are not existing is not always applicable, especially if a constructor requires data to correctly instantiate an object. As non-parametrized constructors are a strict necessity in classes of the metamodel code, the code must be manually refactored a-priori if necessary.

Although this is a rather strict limitation, it is not a limitation of our approach but of the underlying problem. Metamodel-based tools have to rely on non-parametrized constructors, as otherwise, for example, the creation of those objects in a graphical editor is impossible if the data required for instantiation depended on the element type. The only alternative would be a complete re-implementation of the metamodel to be able to apply metamodel-based tools, which makes the adaptation of dependant code hard, as discussed in Section 3. Therefore, refactoring the existing code so that non-parametrized constructors are provided promises to be the less elaborate solution.

7.2 Remaining Challenges

Another category of limitations comprises conceptual and technical challenges that can be solved by extending the approach or its implementation.

Non-default Feature Access. During feature extraction, accessors and mutators are searched by signature matching. If such a method does not contain the default logic, i.e., it modifies the field during access, provides a transient value or even defines completely different logic, it has to be preserved instead of replacing it with the default logic generated for the metamodel feature access. Therefore, it has to be renamed, so that the logic remains and the final feature accessors and mutators can be generated without conflicts.

Non-list Collections. A mechanism to extract multi-valued features from fields of type List was explained in Section 4. Other multi-valued feature types, such as sets, queues or generic collections, cannot be treated that way, as Ecore does not provide dedicated classes for such collection types but handles all of them with specific implementations of the Ecore-specific list implementation EList. Extracting fields with such types as multi-valued features would result in inconformant types. To solve this issue, it is necessary to provide further implementations of the EList interface that additionally implement the appropriate collection interfaces and to integrate them into the code generation.

Technological Challenges. Apart from the presented challenges, there are rather technology-specific ones, e.g., specific for the modelling framework, the programming language or the project configuration. Examples for such challenges are the resolution of potential name clashes or dependency management. EMF, and therefore our approach, relies on the dependency management of Eclipse plug-ins and does not natively support other systems, such as Maven or Gradle. A resolution mechanism would have to be implemented for each of the infrastructures to be used.

7.3 Code Evolution

We have explained in detail how our Ecoreification approach can be applied to existing code. However, code is usually subject to change. Regarding the support of code evolution when applying the Ecoreification, two cases have to be distinguished, depending on who develops and changes the code.

In the first case, the code is developed by the same people that apply the Ecoreification. This allows to apply the Ecoreification and evolve the adapted code further. Changes regarding classes or features extracted to the metamodel can be directly applied to the metamodel and generated into the metamodel code. Only the metamodel code adaptation (see Subsection 5.3) would have to be applied to the newly generated metamodel code again. Changes to functionality that remains in the original code can be applied to the original code as if the Ecoreification was not applied at all.

In the second case, the code is developed and modified externally. In that case, it is not possible to evolve the code with the applied Ecoreification. After each code release, the Ecoreification has to be applied again. Nevertheless, if the extraction scope, which defines which classes and features were extracted, is stored, the approach can be applied fully automated to the modified code again, resulting in code that only differs in the places where the original code was modified.

7.4 Generalizability

Although we presented our approach specific for EMF, we claim that our concept is generalizable to a certain extent and can be transferred to other modelling frameworks. In the following, we sketch necessities for transferring the overall concept to other frameworks. As motivated in Section 3, the essential steps of our concept, metamodel extraction and code adaptation, are necessary, independent from the modelling framework, or otherwise the code already fulfills the required structure and no modification is necessary at all. Since all modelling frameworks need to rely on a specific representation of modelling concepts such as features, either the programming language already supports it or a code generator has to produce it following a specific pattern. At least if that pattern is comparable to that of EMF, especially representing metamodel classes as implementation classes with a commonly implemented interface, our approach can be transferred to the framework.

Using the example of the .NET Modeling Framework (NMF) (Hinkel, 2018), our approach can be transferred completely. Some of the detailed steps can even be omitted, as the underlying language C# already provides a higher level of abstraction than Java. For example, it has *properties* as a language feature that can be used to define features and their accessibility.

Summarizing, the transferability of our concept mainly depends on the comparability of the modelling framework to EMF. The generalizability of the discussed requirements and challenges of the approach were already argued individually.

8 RELATED WORK

The application of metamodel-based tooling to implicit domain metamodels in existing object-oriented code is, to the best of our knowledge, not researched in that generality yet. Nevertheless, there are research topics related to the single steps or the overall concept of our approach, which we discuss in the following.

Reverse Engineering. Our approach starts with reverse engineering a metamodel from existing code. Reverse engineering is an essential and well-researched software engineering task (Canfora et al., 2011; Raibulet et al., 2017). Extracting models of abstract information from a low-level representation, especially code, helps to understand structures and dependencies.

A reverse engineering approach extracts information according to a specific metamodel formalism. One common area is reverse engineering of UML models from code (Kollmann et al., 2002). Especially the extraction of UML class models is comparable to our extraction. A generic framework for UML reverse engineering was proposed by Tonella (2005). Favre (2008) researched specific reverse engineering approaches that are compliant with the model-driven architecture (Object Management Group (OMG), 2006) and use the EMOF metamodel formalism (Favre et al., 2009). The tool MoDisco (Brunelière et al., 2014; Brunelière et al., 2010) extracts Ecore-based models from Java code according to a Java metamodel.

MoDisco and UML reverse engineering tools both extract models according to their own given metamodel. In contrast to that, our approach extracts a metamodel based on the Ecore metamodel formalism, thus the models reside on different modelling levels. Besides that difference, reverse engineering tools, including those presented above, do not aim to perform any integration of the extracted model with the existing code. This reduces the constraints on the extraction mechanism compared to our approach, as no specifically structured code has to be generated and integrated. Finally, our approach does not provide a major contribution to reverse engineering, but just applies an approach with specific constraints, which is why we do not discuss this research field in more detail.

Modelling – Programming Gap. Using modelling frameworks in software development always induces a gap between models and code, as they contain interdependent information that must be kept consistent. This can be achieved with code generators or with automated round-trip engineering approaches (Aßmann, 2003) that allow the simultaneous and consistent modification of code and models.

The creation of metamodels is usually a task performed with specific tools and editors. Nevertheless, approaches like Annotated Java (Steinberg et al., 2008) and Xcore (Bettini, 2016) aim to better embed the metamodel specification into the ordinary programming process by reusing Java syntax for defining Ecore metamodels or embedding operation body specifications into the language. The extension of metamodels with further code is a common problem, as code generators easily overwrite manual modifications of the generated code. Special annotations as used in Ecore or the integration of the logic specification into the metamodel definition as in Xcore try to circumvent that problem. A solution pattern for this problem is known as the "generation gap pattern" (Vlissides, 1998), which proposes to extend the class to enrich with further logic, overwrite the appropriate methods and use that class in clients. An extension of that pattern was proposed by Fowler and Parsons (2010), who suggest to also add a hand-written superclass of the generated class that contains logic independent from the generated code. This is comparable to the way we integrate unification classes.

Our approach also contributes to closing that gap, as it converts implicit metamodel representations in code to explicit metamodel definitions. The discussed approaches enhance forward engineering with a-priori defined metamodels and are therefore suitable for new projects. Instead, our approach aims to ease the transfer of existing projects to a model-driven process.

Abstraction through Modelling. The central goal of model-driven engineering is to improve abstraction. Metamodels abstract from implementation details and rely on a more strictly defined metamodel formalism with explicit definitions of features, multiplicities, etc., on which metamodel-based tools can rely. Such an abstraction can also be achieved by rising the abstraction of programming languages.

One approach for that is UMPLE (Garzón et al., 2015), which integrates modelling concepts into ordinary programming languages, such as C++ and Java. It allows to textually define UML concepts, such as associations with multiplicities, within the source code. The developers of UMPLE also presented reverse engineering approaches for integrating existing source code into UMPLE (Garzón et al., 2014; Lethbridge et al., 2010). This approach improves the abstraction and makes the code specification conform to the UML standard. In contrast to our approach, it does not rely on an existing modelling framework with an existing code generator, but defines its own one. This means that it only introduces abstraction, but does not benefit from the ability to apply metamodel-based tools for that framework afterwards. In consequence, it must not deal with the problem to integrate generated and existing code in a way such that it provides the original API as well as the one required by the modelling framework, which makes the generation and integration of unification classes as well as the modification of generated code necessary. and thus constitutes the complexity of the Ecoreification approach.

Modern programming languages provide constructs for further abstraction, such as property definitions in C#, or allow to dynamically add them to the language, such as *active annotations* in Xtend (Bettini, 2016; Efftinge et al., 2012). This can be employed by modelling frameworks to avoid the implicit encoding of required code structures in the code generated from metamodels, as it is necessary with EMF for Java. One such framework is NMF (Hinkel, 2018), which is able to handle ordinary objects as model elements, because the language already provides the required level of abstraction. But again, this only applies to new projects and does not allow to apply metamodel-based tooling to arbitrary code.

9 CONCLUSIONS & FUTURE WORK

Model-driven software development processes mainly benefit from reusable tooling, such as transformation languages or editor frameworks, based on a modelling framework. This tooling can be applied to explicit metamodels defined with that framework, and to instances of them. It relies on a specific API provided by the code generated from metamodels. Hand-written object-oriented code that contains implicit domain metamodels does usually not provide this API, making the application of metamodel-based tooling to ordinary code impossible.

In this paper, we have presented an approach that makes ordinary object-oriented code accessible to metamodel-based tools. It extracts an explicit metamodel from existing code and integrates it with the code generated from the metamodel. The original API is preserved, yet extended with the API required by the modelling framework. The approach enables developers to make explicit the domain metamodels that are implicitly encoded in object-oriented code, and to separate them from infrastructure code and utilities that are not part of the domain design. In consequence, developers can profit from modelling capabilities, such as explicit model feature representations, persistence, and change notifications, as well as the ability to apply reusable metamodel-based tooling.

We have implemented the approach with the Eclipse Modeling Framework. In our evaluation, we have applied it to three case studies: an artificial employee management system, an open-source graph library and an open-source community case study trading system. We were able to show that, in all three case studies, the modified code provides the same functionality as the original code, and metamodel-based tooling can successfully be applied. The results demonstrate the feasibility of our approach, and are a reliable indicator for its general applicability. We will, however, conduct further case studies to confirm that the approach can be transferred to other domains and to other modelling frameworks. Additionally, we plan to measure the usability of the approach, e.g., in a controlled user study.

Our approach eases the shift from a code-centric to a model-driven development process. Model-driven technologies can be introduced without breaking the compatibility to existing and dependent software and tools. Automated extraction of domain metamodels supports developers in this transition.

REFERENCES

Aßmann, U. (2003). “Automatic Roundtrip Engineering”. *Electronic Notes in Theoretical Computer Science*, *82*(5), 33–41. SC 2003, Workshop on Software Composition (Satellite Event for ETAPS 2003).

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Brunelière, H., Cabot, J., Dupé, G., & Madiot, F. (2014). “MoDisco: A model driven reverse engineering framework”. *Information and Software Technology*, *56*(8), 1012–1032.

Brunelière, H., Cabot, J., Jouault, F., & 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* (pp. 173–174). ASE ’10.

Canfora, G., Di Penta, M., & Cerulo, L. (2011). “Achievements and Challenges in Software Reverse Engineering”. *Commun. ACM*, *54*(4), 142–151.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., & Hanus, M. (2012). Xbase: Implementing Domain-specific Languages for Java. In *Proceedings of the 11th international conference on generative programming and component engineering* (pp. 112–121). GPCE ’12.

Favre, L. (2008). Formalizing MDA-Based Reverse Engineering Processes. In *2008 sixth international conference on software engineering research, management and applications* (pp. 153–160).

Favre, L., Martinez, L., & Pereira, C. (2009). MDA-Based Reverse Engineering of Object Oriented Code. In *Enterprise, business-process and information systems modeling* (pp. 251–263). Springer Berlin Heidelberg.

Fowler, M., & Parsons, R. (2010). *Domain specific languages* (1st). Addison-Wesley, Reading, MA, USA.

Garzón, M. A., Aljamaan, H., & Lethbridge, T. C. (2015). Umple: A Framework for Model Driven Development of Object-Oriented Systems. In *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)* (pp. 494–498).

Garzón, M. A., Lethbridge, T. C., Aljamaan, H., & Badreddin, O. (2014). 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* (pp. 91–105). CASCON ’14. IBM Corp.

Heinrich, R., Rostami, K., & Reussner, R. (2016). *The CoCoME platform for collaborative empirical research on information system evolution* (tech. rep. No. 2016,2; Karlsruhe Reports in Informatics). Karlsruhe Institute of Technology.

Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., … Pfaller, C. (2008). “CoCoME - The Common Component Modeling Example”. In *The common component modeling example* (Vol. 5153, pp. 16–53). LNCS.

Hinkel, G. (2018). NMF: A multi-platform Modeling Framework. In *Theory and practice of model transformations: 11th international conference, icmt 2018, held as part of staf 2018, toulouse, france, june 25-29, 2018. proceedings*, Springer International Publishing.

ISO/IEC 19508:2014(E). (2014). *Information technology – object management group meta object facility (mof) core*. International Organization for Standardization, Geneva, Switzerland.

Klare, H., Burger, E., Kramer, M. E., Langhammer, M., Saglam, T., & Reussner, R. (2017). Ecoreification: Making Arbitrary Java Code Accessible to Metamodel-Based Tools. In *Acm/ieee 20th international conference on model driven engineering languages and systems (models 2017)*.

Kollmann, R., Selonen, P., Stroulia, E., Systa, T., & Zundorf, A. (2002). 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.* (pp. 22–32).

Lethbridge, T. C., Forward, A., & Badreddin, O. (2010). Umplification: Refactoring to Incrementally Add Abstraction to a Program. In *2010 17th working conference on reverse engineering* (pp. 220–224).

Object Management Group (OMG). (2006). Model Driven Architecture - Specifications. OMG.

Meyerovich, L. A., & Rabkin, A. S. (2013). Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 acm sigplan international conference on object oriented programming systems languages & applications* (pp. 1–18). ACM.

Raibulet, C., Fontana, F. A., & Zanoni, M. (2017). “Model-driven reverse engineering approaches: A systematic literature review”. *IEEE Access*, *5*, 14516–14542.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *Emf: Eclipse modeling framework* (second revised). Eclipse series. Addison-Wesley Longman, Amsterdam.

Tonella, P. (2005). *Reverse Engineering of Object Oriented Code*.

Vlissides, J. (1998). *Pattern hatching: Design patterns applied*. Addison-Wesley Longman Ltd.