An Integrated Framework for Running Extended Class Models

Johannes Schröpfer
Chair of Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany

Keywords: Model-Driven Development, Projectional Editing, SOIL, Ecore, OCL, Code Generation.

Abstract: Model-driven software engineering often deals with combinations of structural and behavioral models. In this area, class models are common artifacts expressing the structure of software systems. From class models, source code can be generated which captures the structural elements. To be runnable, the generated code usually has to be completed by implementing behavior. The Eclipse Modeling Framework (EMF) is a popular environment for model-driven software development. In this context, class models are specified as instances of the Ecore metamodel. From Ecore models, Java code is generated that lacks in the implementation of method bodies (typically, only method heads are generated). Our approach supports code generation from extended metamodels comprising structural elements, behavior, and constraints. To this end, we build a projectional editor for a textual modeling language based on SOIL, an imperative extension of OCL. The editor allows for specifying extended class models from which Java code can be generated. Our goal is to reuse the standard EMF code generator. The resulting Java source code is fully executable such that after creating the extended class model, no user interaction is required any more. In this paper, we present the idea of our approach and the current state of implementation.

1 INTRODUCTION

The *Eclipse Modeling Framework* (*EMF*) (Steinberg et al., 2009) constitutes a popular ecosystem for model-driven development. This includes support for implementing models and metamodels with appropriate model editors. It serves as the basic context for many tools and frameworks regarding persistence, user interfaces, and model transformations[1]. For *model editors*, several kinds of representations are considered. The default representation within the EMF core constitutes the EMF tree editor. Frameworks as *GMF*[2] and *Sirius*[3] (Madiot and Paganelli, 2015) provide graphical editors by representing the models as diagrams. Further representations as trees, forms, or tables are supported by *EMF Parsley*[4] (Bettini, 2014).

Recently, *human-readable textual syntax* became more and more popular. The term "human-readable" refers to intuitive textual syntax that resembles the syntax of programming languages in contrast to XML, for instance, which was designed for data exchange instead. As an example for this trend towards human-readable textual syntax, *ALF* (OMG, 2017) was designed to represent UML models using textual notation. ALF models may contain both structural as well as behavioral model elements. It is limited to *fUML* (*Foundational UML*) (OMG, 2021) which refers to a subset of UML and provides execution semantics.

As another textual language, *SOIL* (*Simple OCL-based Imperative Language*) (Büttner and Gogolla, 2014) uses OCL and adds imperative elements. Thus, operation bodies of class models can be completely described. SOIL is implemented in the tool *USE* (*UML-based Specification Environment*) (Gogolla et al., 2007) that poses a validator for models described by UML and OCL and therefore helps developers to analyze models with respect to model and behavior. Using SOIL allows for specifying UML models containing structural elements as classes, associations, and attributes, behavioral elements as operation bodies, and invariants as constraints between different model elements.

For textual editors, two kinds of editor approaches are differentiated. In case of *parser-based* editors, the text is treated as the primary artifact which is persisted and directly modified using editor commands. From the text, the transient model that is represented by the text is derived to perform syntactic and semantic ana-

[1] https://www.eclipse.org/modeling/emf/
[2] https://www.eclipse.org/modeling/gmp/
[3] https://www.eclipse.org/sirius/
[4] https://www.eclipse.org/emf-parsley/

Schröpfer, J.
An Integrated Framework for Running Extended Class Models.
DOI: 10.5220/0011794800003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 75-84
ISBN: 978-989-758-633-0; ISSN: 2184-4348

Table 1: Overview of the different kinds of SOIL statements (Büttner and Gogolla, 2014).

object creation	$v :=$ new c
attribute assignment	$e_1.a := e_2$
variable assignment	$v := e$
operation call	$e_1.op(e_2,\dots,e_n)$
operation call with result	$v := e_1.op(e_2,\dots,e_n)$
block of statements	[declare $v_1 : t_1,\dots,v_n : t_n$] [begin] $s_1;\dots;s_m$ [end]
conditional execution	if e then s_1 [else s_2] end
iteration	for v in e do s end

lyzes. By contrast, *projectional* editors invert this approach and employ the underlying model as the primary artifact which is persisted. Editor commands directly affect the model and changes are propagated to the text as its representation. While for experienced modelers and programmers, projectional editors may feel less natural than syntax-based ones; on the other hand, projectional editors come along with several benefits as the guarantee of syntactic correctness – the representation is derived from the projection rules – or a more flexible tool integration – by referring to the underlying model containing definitely identifiable model elements instead of its representation.

In the EMF context, *Xtext*[5] (Bettini, 2016) constitutes a popular framework that allows for building parser-based textual editors for domain-specific languages. Xtext provides an editor to define the grammar of the language. From the grammar, the functionality of the editor is generated. The generated artifacts can be extended which facilitates implementing static semantics as scoping of named elements, validation rules, quick fixes, or value converters.

Projectional editors were devised several decades ago as components of integrated programming environments. Currently, the *Meta Programming System* (*MPS*)[6] by *JetBrains* (Campagne, 2015) poses a contemporary framework that provides support for developing projectional editors with different kinds of notation (including textual concrete syntax). In (Schröpfer et al., 2020), we presented our approach of a framework for projectional textual model editors in EMF context. By specifying projection rules describing the contextfree syntax and programming editor providers with respect to static semantics – by means of Xtend similar to Xtext –, a projectional editor can be configured for a specific domain-specific language. That approach extends the original idea of projectional editors since also the representation is persisted. That allows users to persistently supply layout information and therefore build a customized view of the underlying model.

This paper presents a projectional editor for textually represented class models based on SOIL as a new use case for the projectional editor framework. This projectional editor constitutes the start artifact of a tool chain comprising different model transformations. The goal is generating fully executable Java source code from the class models. After specifying a class model containing both structural and behavioral elements by the user in the projectional editor, Java source code can be generated which comprises completely implemented methods. Therefore, no additional code fragments have to be inserted by the user. To this end, we reuse the default code generation mechanism provided by EMF.

Section 2 gives an overview of the background of this paper. Section 3 describes the idea of the approach and Section 4 outlines details of the implementation. Section 5 discusses related work while Section 6 concludes the paper.

2 BACKGROUND

The textual language *SOIL* (*Simple OCL-based Imperative Language*) (Büttner and Gogolla, 2014) extends the *Object Constraint Language* (*OCL*) (OMG, 2014) by adding different kinds of imperative constructs. Table 1 shows the statements provided in SOIL. By adding textual notation for structural model elements, class models including operation bodies and invariants may be completely expressed in textual notation. To this end, classes, associations, and enumerations are provided. Classes may contain attributes, operations, and invariants. Associations contain attributes which pose their member ends. For the rest of the paper, we refer to SOIL models as complete models containing structural elements, behavior (expressed by SOIL statements), and invariants.

[5]https://www.eclipse.org/Xtext/

[6]https://www.jetbrains.com/mps/

```
model graph

class Graph
  attributes
    size : Integer derive:
      vertices->size() + vertices
          ->collect(outgoing)->size()
  constraints
    inv NonEmpty: size > 0
class Vertex
  attributes
    label : String init: 'New'
  operations
    addEdge(trg : Vertex) : Boolean
      body:
        declare e : Edge = new Edge.
        e.source := self;
        e.target := trg;
        result := true;
      pre: trg <> null
class Edge

assoc Contains
  graph : Graph
  vertices : Vertex [0..*] comp
assoc HasOutgoing
  source : Vertex
  outgoing : Edge [0..*] comp
assoc IsTargetOf
  target : Vertex
  incoming : Edge [0..*]
```

Listing 1: An example SOIL model (in the context of graphs) in concrete syntax.

A small example model in textual syntax is given in Listing 1. The model named graph consists of the classes Graph, Vertex, and Edge as well as the associations Contains, HasOutgoing, and IsTargetOf. The associations Contains and HasOutgoing contain composite attributes (notated by comp), i.e., they describe compositions. A graph directly contains its vertices as child elements. An edge is modeled as the child element of its source vertex. The derived attribute Graph::size describes the sum of the number of vertices and the number of edges; its semantics is given by an OCL expression. For the graph, the invariant named NonEmpty is defined that enforces a size larger than 0. A vertex has a label where New is the initial value. The operation Vertex::addEdge creates a new edge with the target vertex specified as parameter; its body is specified by a sequence of statements; furthermore, an OCL precondition is given.

As visible in Listing 1, for simplification, we slightly deviate from the SOIL syntax presented in Table 1. The modifications hardly affect the concrete syntax but make the implementation of the projectional editor more convenient. We allow expression statements, i.e., statements each of which contains one expression only; this captures the statements for object creation and operation calls (note that the shown object creation is used as the initializer expression for the local variable and not as an own statement). Attribute assignments, variable assignments, and operation calls with result are modeled as general assignments. The sequence of declarations of local variables at the beginning of a block is finished by a dot. Semicolons are used after all assignments and expression statements (and only in these cases). We completely omit the keywords begin and end as they are redundant and have only a small impact on the readability.

To build the projectional editor, we apply the framework presented in (Schröpfer et al., 2020). After creating the metamodel for the textual language, the projection rules describing the contextfree syntax are defined. Additional functionality regarding static semantics may be implemented as methods in generated class stubs; this includes rules for scoping of named elements and further validation checks which can be specified in a declarative way.

Figure 1 gives an overview of the functionality of the editor framework. Two roles of users are differentiated: While the *DSL developer* configures the editor for a specific DSL metamodel (in our case, the SOIL metamodel), the *modeler* uses the configured editor to invoke different kinds of commands. As the context is the Eclipse Modeling Framework, the basic artifact required to employ the framework is an Ecore model as the metamodel describing the abstract syntax of the DSL. Referring to the metamodel, the syntax is defined by specifying projection rules. In addition, static semantics may be implemented by completing editor provider class stubs. From the text file of projection rules, a model is derived. When the projectional editor is run, the editor providers are called and the syntax definition model is interpreted. The modeler can invoke two kinds of commands: While data

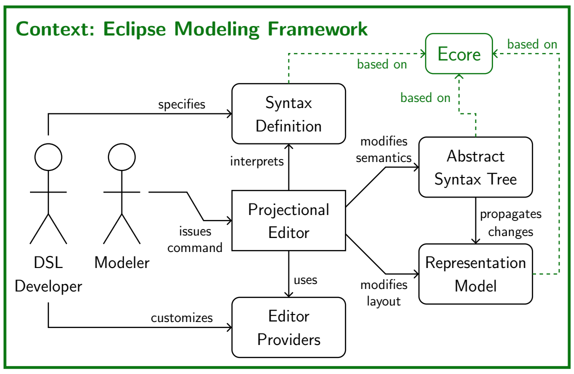

Figure 1: Overview of roles and functionality of the projectional editor framework (Schröpfer et al., 2020).

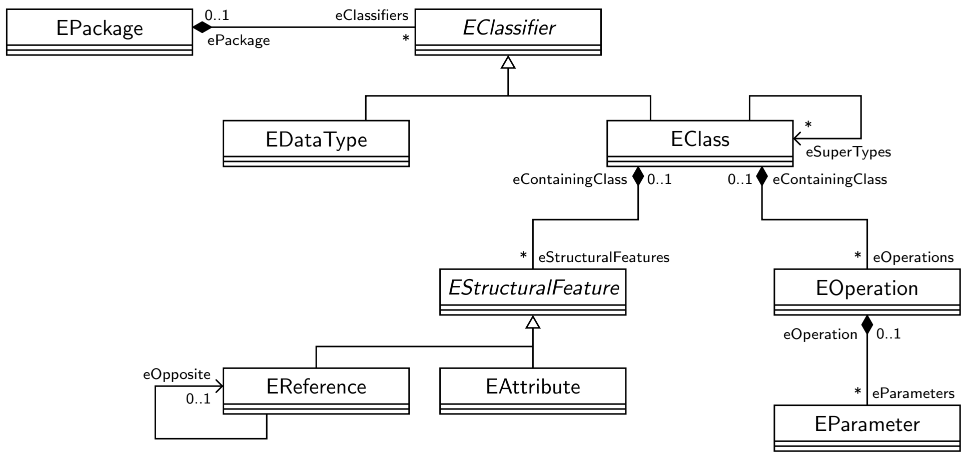

Figure 2: Cutout of the Ecore metamodel (Steinberg et al., 2009).

commands modify the underlying abstract syntax tree (in our case, the SOIL model) and the changes are propagated to its representation, view commands only affect the representation model which is also persisted in order to store the customized layout.

The context is the *Eclipse Modeling Framework* (*EMF*) (Steinberg et al., 2009) with the *Ecore* metamodel describing the abstract syntax of class models. From Ecore models, Java source code can be generated. As Ecore models only cover structural elements, all the generated Java methods have an empty body which has to be implemented by the user afterwards. In addition, EMF allows for mechamisms to configure tree editors.

Figure 2 shows a cutout of the Ecore metamodel. Packages (EPackage instances) contain classifiers, i.e., classes (EClass instances) and data types (EDataType instances). Classes can have superclasses (EClass::eSuperTypes) and contain structural features divided into attributes (EAttribute instances) and references (EReference instances); while attributes refer to data types, references are typed with classes similar to association ends in UML. A pair of two references can be grouped (EReference::eOpposite). Note that Ecore does not support the concept of associations that are commonly used in UML (and SOIL) models. In addition, classes contain operations (EOperation instances) that comprise parameters (EParameter instances).

For the model-to-model transformation, we use BXtendDSL. The framework *BXtendDSL* (Bank et al., 2021) provides support for bidirectional and incremental model transformations referring to arbitrary metamodels as Ecore models. Traces between source and target model are persisted within a correspondence model. The framework contains a DSL which allows for declarative projection rules. To express more complex transformations, generated Xtend methods can be implemented.

For the model-to-text transformation, Acceleo is applied. The tool *Acceleo*[7] poses an implementation of the *Mof2Text* standard (OMG, 2008). The text generation is built by means of templates containing static as well as dynamic fragments which refer to the metamodel elements (in our case: SOIL elements).

3 CONCEPT

This section presents the conceptual idea our approach is based upon. After an overview of the tool chain comprising the different artifacts and their dependencies, we show an example model and describe the respectively resulting models and files when the transformations in the different steps of the tool chain are applied.

3.1 Overview of the Tool Chain

Figure 3 illustrates the different conceptual steps of the workflow. The SOIL model created and modified in the editor contains all the information of the class model including structure, behavior, and constraints. A model-to-model transformation creates an Ecore model and converts the structural elements of the class model – i.e., classes, associations, attributes, and operations – to analogous elements of the Ecore model. As the initial Ecore model only contains information about the structural elements of the class model, if the EMF code generator was applied, the generated Java method bodies would be empty. For the operation bodies within the SOIL model, a model-

[7]https://www.eclipse.org/acceleo/

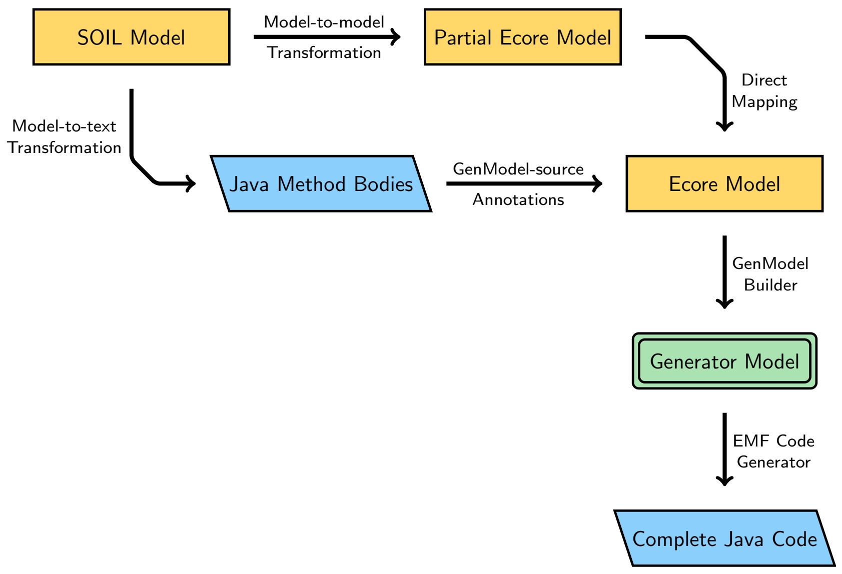

Figure 3: Overview of the tool chain.

to-text transformation creates Java source code for the method bodies. From the initial partial Ecore model, an extended Ecore model is established which provides annotations containing the information of the generated method bodies. To this extended Ecore model, the default EMF mechanism is applied and the generator model is created.

Finally, Java source code is generated by the default EMF code generator. The resulting code contains Java interfaces and Java classes with fields and methods which correspond to the classes, structural features, and operations in the Ecore (and SOIL) model. In addition, the Java methods contain statements which correspond to the statements in the SOIL model. Pre- and postconditions of the operations are checked in the Java methods. As a consequence, the source code is fully executable and therefore does not require additional elements written by the user.

Within the SOIL model, besides classes with attributes and operations, invariants are specified, as well. For the invariants, we plan a model-to-text transformation in order to create OCL files. The generated OCL files can be directly run by the tool Complete OCL. Since the invariants are not directly connected to the generated Java code, we omit this part of the tool chain in this paper.

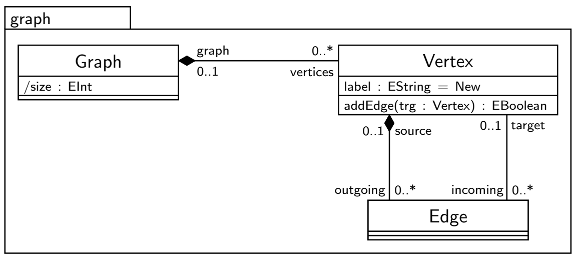

Figure 4: The partial Ecore model as target of the model-to-model transformation for the example SOIL model (Listing 1).

3.2 Demonstration for an Example Model

We refer to the SOIL model described in Section 2 (textually notated in Listing 1). From the structural elements of the SOIL model, a partial Ecore model is built. The created Ecore model as the target model of the model-to-model transformation is depicted using class diagram notation in Figure 4. For the SOIL classes in the model, corresponding EClass instances as child elements of a respective EPackage instance are created. The attributes and operations contained in the SOIL classes are mapped to respective EAttribute instances and EOperation instances. For associations, no adequate concept is given by Ecore; rather, the association ends – i.e., the attributes within the associations – are modeled as EReference instances which are contained in the respective member classes. Each pair of references that arise from attributes contained in the same association is grouped.

The SOIL statements contained in the operation bodies are converted to Java source code by means of the model-to-text transformation. The resulting Java method bodies are inserted in the Ecore model by means of GenModel-source annotations. For the resulting complete Ecore model, the generator model is built and the EMF code generator is run. Listing 2 shows the generated Java methods for the derived attribute and the operation. The precondition is checked when the method starts. If it is not satisfied, an exception is thrown (cf. lines 8–11). Analogously, postconditions lead to checks at the end of a method. The predefined result variable is mapped to a local variable (cf. line 13) that is finally returned (cf. line 20).

```
 1 public int getSize() {
 2   return getVertices().size() + Util.collect(
 3       getVertices(),
 4       v -> v.getOutgoing()).size();
 5 }
 6
 7 public boolean addEdge(Vertex trg) {
 8   if (trg == null) {
 9     throw new UnsupportedOperationException(
10         "Precondition not satisfied");
11   }
12
13   boolean result = false;
14   Edge e = GraphFactory.eINSTANCE.createEdge();
15
16   e.setSource(this);
17   e.setTarget(trg);
18   result = true;
19
20   return result;
21 }
```

Listing 2: The generated Java methods for the example SOIL model (Listing 1).

4 IMPLEMENTATION

In this section, we outline the current state of the implementation of our approach. Furthermore, we describe the ideas of the remaining steps. After details regarding the projectional editor for the language SOIL, we depict the implementation of the conceptual tool chain.

4.1 Projectional SOIL Editor

We created the metamodel for class models containing SOIL statements as an Ecore model. Based on the SOIL metamodel, a sequence of projection rules was specified in a specific editor. For each non-abstract class, a projection rule was defined. The projectional editor framework distinguishes between block patterns – e.g., the definitions of classes or operations which pose blocks with head lines and bodies – and line patterns – e.g., arithmetic OCL expressions or references to variables. From the text file of the projection rules, a model is built which is interpreted by the editor at runtime.

Listing 3 illustrates the projection rule for assignments. The projection rule constitutes a line pattern and consists of line elements. For both delimiters (operator := and final semicolon), constants are inserted. The right-hand side of the assignment is an arbitrary expression; to this end, the rule Expression is referenced. For the left-hand side, variable references (referring to local variables), attribute call expressions (referring to attributes as child elements of classes), and result expressions (used for return values of operations) are applicable; to embody alternatives, pipe characters within a pair of parentheses are used. Eventually, the keyword nospace suppresses a blank character before the semicolon; by default, blank characters are set between line elements in the editor.

```
def Assignment
  (VariableReference | AttributeCall | Result)
  ':=' Expression nospace ';'
enddef
```

Listing 3: The projection rule for SOIL assignments (line pattern).

Furthermore, static semantics are implemented by means of methods in Xtend class stubs generated by the framework. While the method for default scoping is redefined in the respective class stub, validation rules are specified in a declarative way. For each validation rule, a method is defined with the respective context model element as its parameter. A validation result is added by invoking a method. To this end, the three severities error, warning, and information are provided.

```
@Rule
def checkAssignmentTypeCompatibility(
    Assignment amt) {
  if (!amt.getRight().isCompatibleTo(
      amt.getLeft()))
    reportError(ASSIGNMENT_TYPE_COMPATIBILITY,
        "Right-hand side not compatible" +
            "to left-hand side",
        amt,
        SoilPackage.Literals.ASSIGNMENT_RIGHT)
}
```

Listing 4: The validation rule for type checking of SOIL statements (Xtend method).

Listing 4 shows the definition of a validation rule for assignments. To compare the types, an auxiliary method isCompatibleTo is used (not shown here). In case the types are not compatible, an error is fired. To this end, the method reportError has several parameters: The first parameter is a unique string ID and the second parameter poses the description that is visible in the user interface; the context object is specified as the third parameter; finally, the optional fourth parameter allows for defining the affected structural feature which allows for a more precise visualization of the error (in this case, the right-hand side of the expression).

4.2 Tool Chain of Model Transformations

The model-to-model transformation mapping SOIL elements to Ecore elements is currently implemented by means of BXtendDSL. In a text editor, the transformation rule can be specified in a declarative way. Details which cannot be expressed by the declarative rule, are implemented in an imperative way by completing Xtend methods generated from the transformation rule. The framework BXtendDSL allows for incremental and bidirectional model-to-model transformations. This will allow an additional workflow covered by future work where the Ecore model is modified and the changes are propagated to the SOIL model.

```
1  rule Attribute2EAttribute
2    src Attribute att | filter;
3    trg EAttribute eAtt;
4
5    att.name <--> eAtt.name;
6    att.ordered <--> eAtt.ordered;
7    att.unique <--> eAtt.unique;
8    att.lower <--> eAtt.lowerBound;
9    att.upper <--> eAtt.upperBound;
10   att.derived <--> eAtt.derived;
11
12   att.type --> eAtt.eType;
13   att.initializer --> eAtt.defaultValueLiteral;
14   att.derived --> eAtt.changable;
15   att.derived --> eAtt.transient;
16   att.derived --> eAtt.volatile;
```

Listing 5: Example BXtendDSL transformation rule that maps SOIL attributes to Ecore attributes.

The larger part of the SOIL model corresponds directly to the Ecore model. Nevertheless, the concept of associations is not given by the Ecore metamodel. Rather, association ends are mapped to references that are contained in the respective member classes. Listing 5 shows the transformation rule which maps attributes in SOIL classes to Ecore attributes. Since only a subset of the attributes comprised by the SOIL model are transformed – the attributes contained in classes (and not in associations) – we need a filter for the source (cf. line 2). Several values can be directly mapped notated by the bidirectional arrow (cf. lines 5–10). For the additional values – only given by the Ecore model –, for the type of the attribute – a SOIL type cannot directly mapped to an Ecore type –, and for the initial (default) value, additional mappings are required. For these cases (cf. lines 12–16), the unidirectional arrow is set which results in generated Xtend method stubs. By implementing these methods imperatively in a second step, the semantics of the more complex mappings can be specified (not shown here).

Large parts of the model-to-text transformation which generates Java source code for operation bodies have already been implemented by using Acceleo. We plan the integration of the generated Java source code in the Ecore model by means of annotations analogously to previous work (Buchmann and Schwägerl, 2015). EMF provides different kinds of annotations to store information that is not directly supported by the Ecore metamodel. Information which is relevant for code generation may be attached by using *GenModel-source annotations* that are only relevant for code generation. An annotation is implemented as a map details of key-value entries. For operations in Ecore models – i.e., EOperation instances –, the key body is provided; the value is the Java code that poses the body of the generated Java method. Analogously, for derived structural features in Ecore models – i.e., EStructuralFeature instances –, the key get is provided; the value is the Java code for the body of the generated Java getter method. Listing 6 depicts the structure of EAnnotation instances.

```
<eOperations name="addEdge" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean">
  <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="body" value="Edge e = GraphFactory.eINSTANCE.createEdge(); ..."/>
  </eAnnotations>
  ...
</eOperations>
```

Listing 6: XMI serialization of a GenModel-source annotation for an Ecore operation in the context of the graph example.

5 RELATED WORK

This section describes related work in the research area of executable models. In the EMF OCL implementation, the tool *OCLinEcore*[8] provides a textual editor for Ecore models. An Ecore model may be displayed by the editor using textual notation for the structural elements. Within the editor, invariants as well as behavior for operations and derived structural features may be specified. To this end, the editor deviates from the official OCL standard notation and

[8]https://wiki.eclipse.org/OCL/OCLinEcore

employs additional keywords (invariant and derivation instead of inv and dev). In contrast to SOIL, no additional statements may be used which prevents side effects. Furthermore, the textual editor is not projectional but parser-based and directly refers to the Ecore model; as a result, references are used instead of associations (that are familiar for modelers used to UML), for instance. Eventually, the OCL implementations do not affect the generated Java source code; rather, mechanisms are used to delegate the invocations of operations to the expressions written in OCL syntax.

Besides SOIL, the modeling language *ALF* (OMG, 2017) recently gained more and more attention. It provides a textual concrete syntax for *fUML* (OMG, 2021) that refers to a subset of UML and adds proper execution semantics. In contrast to SOIL, the language ALF does not reuse an expression language. Furthermore, ALF completely misses the concept of derived structural features. Invariants as constraints between model elements cannot be modeled, as well. An extension of the tool *Valkyrie* (Schröpfer and Buchmann, 2019) supports creating models including structural elements and behavior and generating Java source code from them. Within an integrated user interface, the structural elements are specified by diagrams in a graphical editor. For operations and derived structural features, the behavior may be specified by using ALF in a textual parser-based editor that is visible together with the diagram editor. In the background, a bidirectional transformation between UML and ALF is performed.

The tool *Papyrus* (Guermazi et al., 2015) poses another application that refers to ALF. Papyrus allows for creating UML, SysML, and MARTE models using various diagram editors. Papyrus comes along with a code generation engine that facilitates generating source code from class models; besides Java source code, also C++ is supported. Using a textual parser-based ALF editor, UML model elements may be described by ALF notation. In addition, the behavior of activities can be specified in the ALF editor. The primary goal of the Papyrus ALF integration is providing alternative representations – graphical and textual syntax – of the same UML model and not generating source code to execute the behavioral specifications.

As a commercial tool, *MagicDraw UML* (Seidewitz, 2017) also provides behavioral modeling with ALF which integrates the *ALF Reference Implementation*[9]. Using ALF syntax, bodies of activities may be modeled. The *Cameo Simulation Toolkit*[10] is required to execute UML activity, state machine, and interaction models in MagicDraw. Similar to Papyrus, the ALF fragments are compiled to activity models that are integrated within the wider UML modeling context.

The language *QVTo* (*Operational QVT*) as a part of the *QVT* (*Query/View/Transformation*) standard (OMG, 2016) allows for specifying unidirectional model transformations in an imperative way. To this end, mappings between model elements can be specified by using an imperative language based on OCL called *Imperative OCL*. In contrast to SOIL, Imperative OCL uses imperative expressions as a subtype of OCL expressions. This circumstance comes along with problems which are discussed in (Büttner and Kuhlmann, 2008). While SOIL and ALF aim at an integration of the imperative elements into the model to decribe its behavior, QVTo employs imperative constructs to define the transformation between models.

In the context of Ecore models, *Xcore*[11] provides a textual concrete syntax that captures both structure and behavior. The textual syntax of Xcore does not base upon an official standard. From Xcore models, executable Java source code is generated. The behavioral model elements are described in a strongly procedural way. As a consequence, an object-oriented way of modeling – as known from OCL or ALF – is only possible to a limited extent. For instance, no concept of object creation by means of constructors is supported by Xcore.

As a graphical modeling language, *Fujaba* (The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth, 2005) bases upon graph transformations that allows for expressing both the structural and the behavioral part of a software system on the modeling level. A code generation engine facilitates the transformation of Fujaba specifications into executable Java source code. Fujaba employs *story diagrams* to specify the behavior of a software system. Similar to UML, story diagrams contain activities for which story patterns are used. A *story pattern* constitutes a graph transformation rule where a single graphical notation comprises both the left-hand as well as the right-hand side of the transformation rule. Therefore, manipulations of the runtime object graph may be described by story patterns in a declarative way on a high level of abstraction. Nevertheless, the control flow in activity diagrams in on the same level as in control flow diagrams which leads to a rather basic level of abstraction in respect to the control flow of methods. As a case study (Buchmann et al., 2011) revealed, software systems only contain a low number of problems which require complex story patterns. The resulting story diagrams nevertheless

[9]http://alf.modeldriven.org

[10]https://docs.nomagic.com/display/CST190/Cameo+Simulation+Toolkit+Documentation

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

are big and look complex because of the limited capabilities to express the control flow.

Future work will cover the mapping of invariants that are also elements of the SOIL model. The EMF OCL implementation offers the tool *Complete OCL*[12]. It allows for running invariants to check an EMF-based model. Our goal is just reusing this mechanism: From the invariants within one SOIL model, one text file is generated comprising the respective OCL invariants that are directly runnable by Complete OCL. Thus, invariants can be checked on demand. In (Heidenreich et al., 2008), the authors present an approach that maps invariants in the context of data queries. The proposed framework allows for mapping UML models to arbitrary data schemas. The referring OCL invariants result in sentences in corresponding declarative query languages where semantical data integrity on implementation level is forced.

6 CONCLUSION

This paper presented an approach of a tool chain that allows for generating fully executable Java source code from class models. In a projectional editor, class models can be specified textually comprising structural elements (classes, associations, attributes), behavior (operations including body statements), and invariants (constraints between model elements). To represent the statements in the operations of the class model, the textual language SOIL is used. The resulting Java code comprises code artifacts corresponding to all class model elements, in particular fully implemented method bodies. Therefore, no additional code fragments inserted by the user are required.

This work is still ongoing. We already implemented a primary version of the projectional text editor and large parts of the model-to-text transformation that maps SOIL statements to Java code. Currently, the model-to-model transformation which maps structural elements specified in the textual editor to Ecore model elements is built. Future work will deal with the integration of the target code of the model-to-text transformation in the Ecore model which is generated as the target of the model-to-model transformation; to this end, annotations are going to play a major role. Also the question whether these annotations can be directly included by the model-to-text transformation will be discussed. Regarding the model-to-model transformation, supporting the backward direction may pose a wise extension; that would allow for propagating changes of the Ecore model to the SOIL model presented by the projectional editor. Furthermore, the question of processing invariants in the class model is going to be considered.

Eventually, after completing the implementation, an evaluation of the approach will follow. For a range of class models from various contexts, the size and complexity of the respective SOIL models should be compared with the corresponding Ecore models (only containing structural information) and the Java source code that has to be written manually after applying the EMF code generator to the (partial) Ecore model. In this context, an additional model-to-model transformation may be useful that transforms (already existing) UML models to corresponding SOIL models; this extension could allow for reusing models from different existing projects and tools within the context of the EMF environment.

[12]https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FCompleteOCLTutorial.html

ACKNOWLEDGEMENTS

The author wants to thank Jonas Adler for implementing major parts of the model-to-text transformation which maps the behavioral SOIL model elements to Java code fragments in the scope of his bachelor's thesis.

REFERENCES

Bank, M., Buchmann, T., and Westfechtel, B. (2021). Combining a declarative language and an imperative language for bidirectional incremental model transformations. In Hammoudi, S., Pires, L. F., Seidewitz, E., and Soley, R., editors, *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 15–27. INSTICC, SciTePress.

Bettini, L. (2014). Developing user interfaces with EMF parsley. In Holzinger, A., Cardoso, J., Cordeiro, J., van Sinderen, M., and Mellor, S. J., editors, *Proceedings of the 9th International Conference on Software Paradigm Trends (ICSOFT 2014)*, pages 58–66. SciTePress.

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd., Birmingham B3 2PB, UK, second edition.

Buchmann, T. and Schwägerl, F. (2015). On a-posteriori integration of ecore models and hand-written java code. In Lorenz, P., van Sinderen, M., and Cardoso, J., editors, *Proceedings of the 10th International Conference on Software Paradigm Trends (ICSOFT 2015)*, pages 95–102. INSTICC, SciTePress.

Buchmann, T., Westfechtel, B., and Winetzhammer, S. (2011). The added value of programmed graph transformations – A case study from software configuration management. In Schürr, A., Varró, D., and Varró, G., editors, *Applications of Graph Transformations with Industrial Relevance – 4th International Symposium (AGTIVE 2011), Revised Selected and Invited Papers*, volume 7233 of *Lecture Notes in Computer Science*, pages 198–209. Springer.

Büttner, F. and Gogolla, M. (2014). On ocl-based imperative languages. *Sci. Comput. Program.*, 92:162–178.

Büttner, F. and Kuhlmann, M. (2008). Problems and enhancements of the embedding of OCL into QVT imperativeocl. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 15.

Campagne, F. (2015). *The MPS Language Workbench*, volume I. Fabien Campagne, second edition.

Gogolla, M., Büttner, F., and Richters, M. (2007). USE: A uml-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34.

Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., and Gérard, S. (2015). Executable modeling with fuml and alf in papyrus: Tooling and experiments. In Mayerhofer, T., Langer, P., Seidewitz, E., and Gray, J., editors, *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, volume 1560 of *CEUR Workshop Proceedings*, pages 3–8. CEUR-WS.org.

Heidenreich, F., Wende, C., and Demuth, B. (2008). A framework for generating query language code from OCL invariants. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 9.

Madiot, F. and Paganelli, M. (2015). Eclipse sirius demonstration. In Kulkarni, V. and Badreddin, O., editors, *Proceedings of the MoDELS 2015 Demo and Poster Session co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015)*, volume 1554 of *CEUR Workshop Proceedings*, pages 9–11. CEUR-WS.org.

OMG (2008). *MOF Model to Text Transformation Language, v1.0*. Object Management Group, Needham, MA, formal/2008-01-16 edition.

OMG (2014). *Object Constraint Language*. Object Management Group, Needham, MA, formal/2014-02-03 edition.

OMG (2016). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, Needham, MA, formal/2016-06-03 edition.

OMG (2017). *Action Language for Foundational UML (Alf)*. Object Management Group, Needham, MA, formal/2017-07-04 edition.

OMG (2021). *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Object Management Group, Needham, MA, formal/2021-03-01 edition.

Schröpfer, J. and Buchmann, T. (2019). Integrating UML and ALF: an approach to overcome the code generation dilemma in model-driven software engineering. In Hammoudi, S., Pires, L. F., and Selic, B., editors, *Model-Driven Engineering and Software Development – 7th International Conference (MODELSWARD 2019), Revised Selected Papers*, volume 1161 of *Communications in Computer and Information Science*, pages 1–26. INSTICC, Springer.

Schröpfer, J., Buchmann, T., and Westfechtel, B. (2020). A generic projectional editor for EMF models. In Hammoudi, S., Pires, L. F., and Selic, B., editors, *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 381–392. INSTICC, SciTePress.

Seidewitz, E. (2017). A development environment for the alf language within the magicdraw UML tool (tool demo). In Combemale, B., Mernik, M., and Rumpe, B., editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*, pages 217–220. ACM.

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

The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth (2005). The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. In Giese, H. and Zündorf, A., editors, *Proceedings of the 3rd international Fujaba Days*, pages 1–13.