Integrating UML and ALF: An Approach to Overcome the Code Generation Dilemma in Model-Driven Software Engineering

Johannes Schröpfer(✉) and Thomas Buchmann

Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany
{johannes.schroepfer,thomas.buchmann}@uni-bayreuth.de

Abstract. The state of the art in model-driven software engineering is a combination of structural modeling and conventional programming to supply the operational behavior of the system. This fact leads to the so-called code generation dilemma when model and hand-written code evolve independently during the software development process. In this paper we present an approach of integrating two OMG standards to overcome this problem: A tight integration of UML used for structural modeling and the Action Language for Foundational UML (ALF) for behavioral modeling using a textual surface notation leads to a full-blown model-driven process which allows for the generation of fully executable source code. Supplying hand-written code fragments in the target language is no longer necessary.

Keywords: Model-driven development · Code generation · Executable models · ALF · UML · BXtend · fUML · Xtext

1 Introduction

This paper is an extended version of [20] and provides besides an extended example use case some more technical details. *Model-driven software engineering* (MDSE) [24] aims at reducing effort for developing software by specifying higher-level (executable) models, instead of lower-level hand-written source code. An initial model capturing the requirements is often the starting point from which a number of models over multiple levels of abstraction is derived, until the system is eventually implemented. In order to support model-driven software engineering in a full-fledged way, key enabling technologies are mandatory for defining modeling languages and specifying and executing model transformations.

Usually, *modeling languages* are defined with the help of *metamodels* in the context of object-oriented modeling. To this end, the *Object Management Group (OMG)* provides the *Meta Object Facility (MOF)* standard [18]. Throughout the last two decades, *UML* [19] has been established as the de-facto standard modeling language for model-driven development. In its current version, UML comprises seven kinds of diagrams dedicated to structural modeling and seven different diagrams addressing behavioral aspects of a software system. In order to support model-driven software engineering in a full-fledged way, having models which allow for a generation of fully executable code is crucial.

However, generating executable code requires a precise and well-defined execution semantics of behavioral models. Unfortunately, not all behavioral diagrams provided by UML are equipped with such a well-defined semantics. Furthermore, some diagrams with a well-defined execution semantics, e.g., activity diagrams, are on a lower level of abstraction in terms of specifying control flow. As a consequence, the state of the art in model-driven software engineering nowadays is specifying the static structure of the software system using models from which source code is generated. This generated source code is then augmented with behavioral elements using regular programming languages.

This fact which we call the "*code generation dilemma*" [6] is problematic as the different fragments of the software system tend to evolve separately which quickly leads to inconsistencies between the model and the (generated) source code. Round-trip engineering [7] may help to keep the structural parts consistent but unfortunately there is still no adequate representation of the manually supplied behavioral fragments.

The *Action Language for Foundational UML (ALF)* [15] is also an OMG standard addressing a textual surface representation for a major part of UML model elements. Furthermore, it provides an execution semantics via a mapping of the ALF concrete syntax to the abstract syntax of the OMG standard of a *Foundational Subset for Executable UML Models*, also known as *Foundational UML* or just *fUML* [16]. The primary goal is to provide a concrete textual syntax allowing software engineers to specify executable behavior within a wider model which is represented using the usual graphical notations of UML. A simple use case is the specification of method bodies for operations contained in class diagrams. To this end, it provides a language with a procedural character whose underlying data model is UML. However, ALF also provides a concrete syntax for structural modeling within the limits of the fUML subset.

In the academic world, the *Eclipse Modeling Framework (EMF)* [22] constitutes the platform for research dedicated to model-driven software engineering. Its metamodel *Ecore* is based on a subset of MOF called *Essential MOF (EMOF)*. Following a pragmatic approach, EMF strictly focuses on principles from object-oriented modeling only providing core concepts for defining classes, attributes, and relationships between classes. Furthermore, it allows for Java code generation from these structural model definitions. EMF provides an extensible platform for the development of MDSE applications.

In this paper, we present a tight integration of the OMG standards UML and ALF to realize an integrated modeling environment which allows for structural as well as behavioral modeling. Fully executable Java source code is generated from the resulting models, allowing for "real" MDSE approaches.

The paper is structured as follows: Sect. 2 provides a conceptual overview of our solution. An example use case is presented in Sect. 3. Technical details are given in Sect. 4, while our approach is discussed in Sect. 5. Related work is discussed in Sect. 6 before Sect. 7 concludes the paper.

2 Overview

As stated above, the current state of the art in model-driven software engineering is modeling the structure of a software system which is used as a basis for generating code. In a subsequent step, the generated code is augmented with manually programmed method bodies to supply behavior as shown in Fig. 1. In a strict waterfall-like development process, this approach is feasible. But in iterative processes, the danger of model and code evolving separately is immanent. In the past, approaches addressing this issue using round-trip engineering between model and source code have been published [5,7,9]. However, a much better solution would clearly separate the primary development artifacts – i.e., the development models – from generated ones – i.e., the generated platform-specific source code – in order to avoid a concurrent evolution of artifacts located on different levels of abstraction as depicted in Fig. 2.

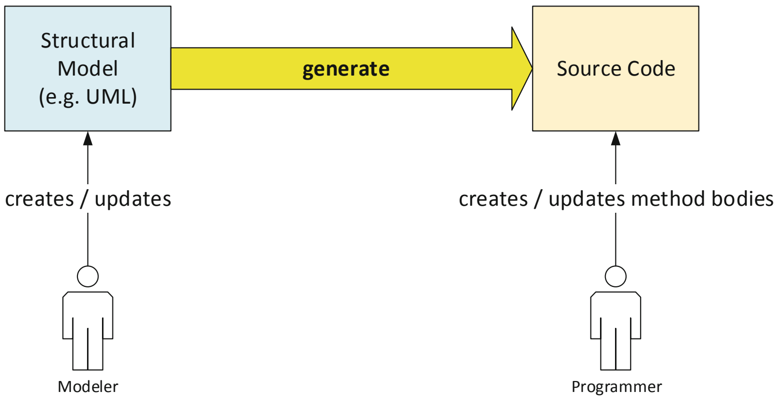


Fig. 1. A state-of-the-art MDSD process.

To this end, our work presented in this paper integrates two OMG standards in a single tool-chain. (1) UML [19] models are used for structural modeling and (2) ALF [15] is used to model the behavior in terms of method bodies. While UML provides a wide range of diagrams supporting structural modeling, we limited ourselves to package diagrams for the task of modeling-in-the-large [4] and class diagrams for modeling-in-the-small. The ALF standard primarily addresses a textual surface representation for a subset of UML model elements. Its main benefit is an execution semantics which allows for the generation of *executable code* out of ALF specifications. Executable code in this context means that the resulting source code also comprises method bodies and extending generated source code with hand-written code fragments is no longer necessary.

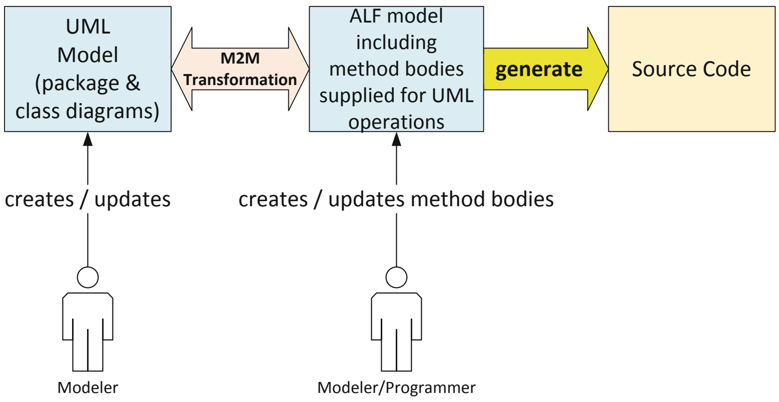

UML Model (package & class diagrams)
M2M Transformation
ALF model including method bodies supplied for UML operations
generate
Source Code
creates / updates
creates / updates method bodies
Modeler
Modeler/Programmer

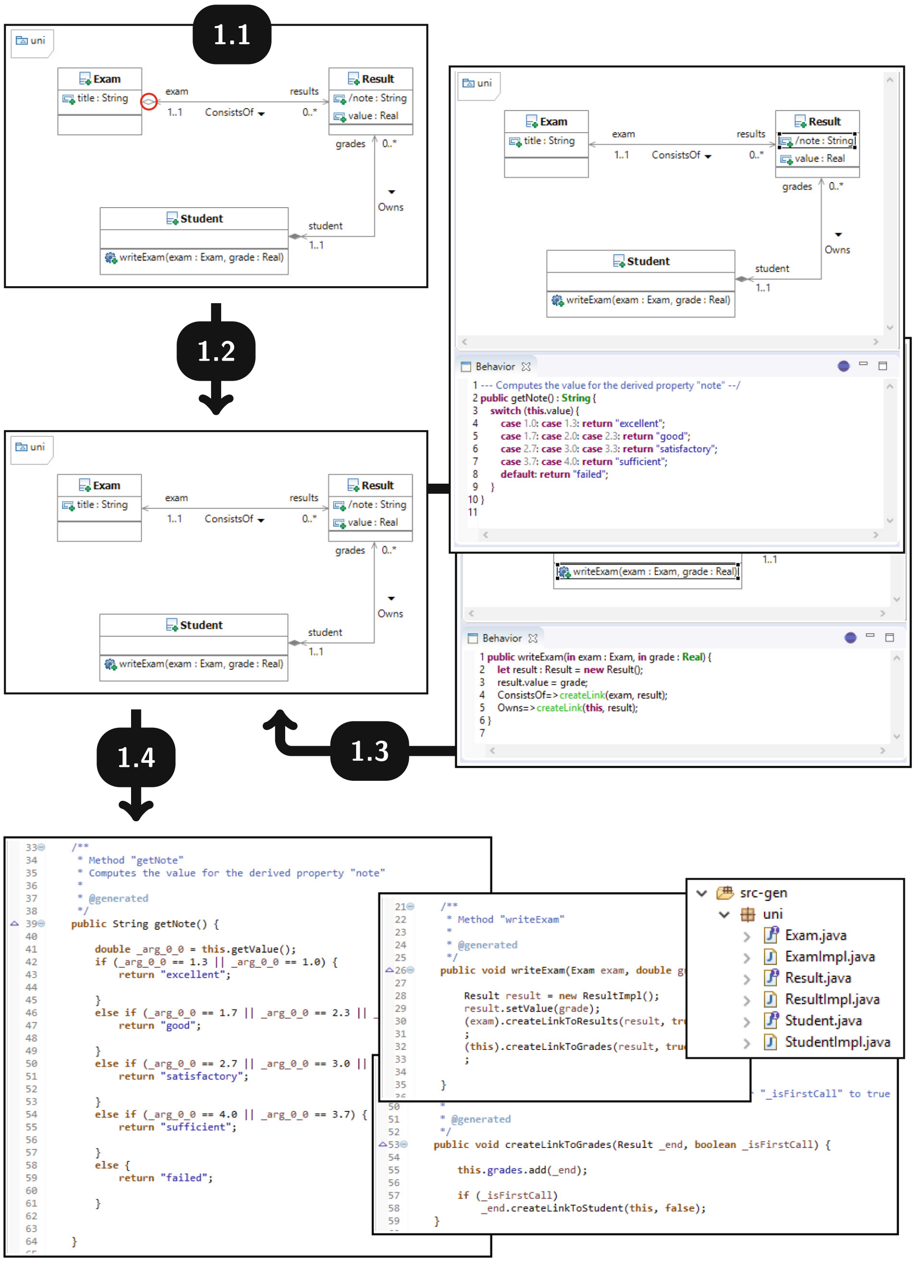
Fig. 2. MDSD process realized with our approach.

The following section introduces an example workflow which may occur in a real-life iterative development process as an interaction between design and implementation phase.

3 Example Workflow

This section describes an example development process of a software system in the context of a university management system using our tool. The complete workflow consists of four iterations each of which considering modeling parts of the structure, augmenting the model with behavioral elements, and eventually generating fully executable Java source code. At the end of this section some JUnit test fragments are provided that use the generated code and test its functionality. This example is an extended version of that shown in [20, Sect. 5].

Figure 3 depicts the first iteration. We start with a UML class diagram (step 1.1) that contains the three classes `Exam`, `Result`, and `Student`. As semantic relationships between them, a composition and an aggregation is used. The classes contain one operation and several properties where the property `Result::note` is derived; for an instance of the class `Result`, the value `note` constitutes the string representation of the numeric literal stored by `Result::value`, e.g., the note `"excellent"` for the values `1.0` and `1.3`. By invoking the corresponding command, the background transformation creates the ALF model system from the UML model (step 1.2). The ALF model system consists of the main model for the structural model elements and two branch models for the created ALF operations storing the behavioral elements; one ALF operation corresponds with the UML operation and one with the derived UML property. Since derived UML properties are not contained in the fUML subset, they are approximated by ALF operations that are supposed to define the computation semantics. In addition, fUML does not support aggregations. In this case, the user is informed of this incidence and can decide whether the transformation is executed or aborted; if the transformation is not aborted, its execution results



```
--- Computes the value for the derived property "note" --/
public getNote() : String {
  switch (this.value) {
    case 1.0: case 1.3: return "excellent";
    case 1.7: case 2.0: case 2.3: return "good";
    case 2.7: case 3.0: case 3.3: return "satisfactory";
    case 3.7: case 4.0: return "sufficient";
    default: return "failed";
  }
}
```

```
public writeExam(in exam : Exam, in grade : Real) {
  let result : Result = new Result();
  result.value = grade;
  ConsistsOf=>createLink(exam, result);
  Owns=>createLink(this, result);
}
```

```
    /**
     * Method "getNote"
     * Computes the value for the derived property "note"
     *
     * @generated
     */
    public String getNote() {

        double _arg_0_0 = this.getValue();
        if (_arg_0_0 == 1.3 || _arg_0_0 == 1.0) {
            return "excellent";

        }
        else if (_arg_0_0 == 1.7 || _arg_0_0 == 2.3 || _
            return "good";

        }
        else if (_arg_0_0 == 2.7 || _arg_0_0 == 3.0 ||
            return "satisfactory";

        }
        else if (_arg_0_0 == 4.0 || _arg_0_0 == 3.7) {
            return "sufficient";

        }
        else {
            return "failed";

        }


    }
```

```
    /**
     * Method "writeExam"
     *
     * @generated
     */
    public void writeExam(Exam exam, double g

        Result result = new ResultImpl();
        result.setValue(grade);
        (exam).createLinkToResults(result, tr
        ;
        (this).createLinkToGrades(result, tru
        ;

    }
```

```
                                                 "_isFirstCall" to true
     *
     * @generated
     */
    public void createLinkToGrades(Result _end, boolean _isFirstCall) {

        this.grades.add(_end);

        if (_isFirstCall)
            _end.createLinkToStudent(this, false);
    }
```

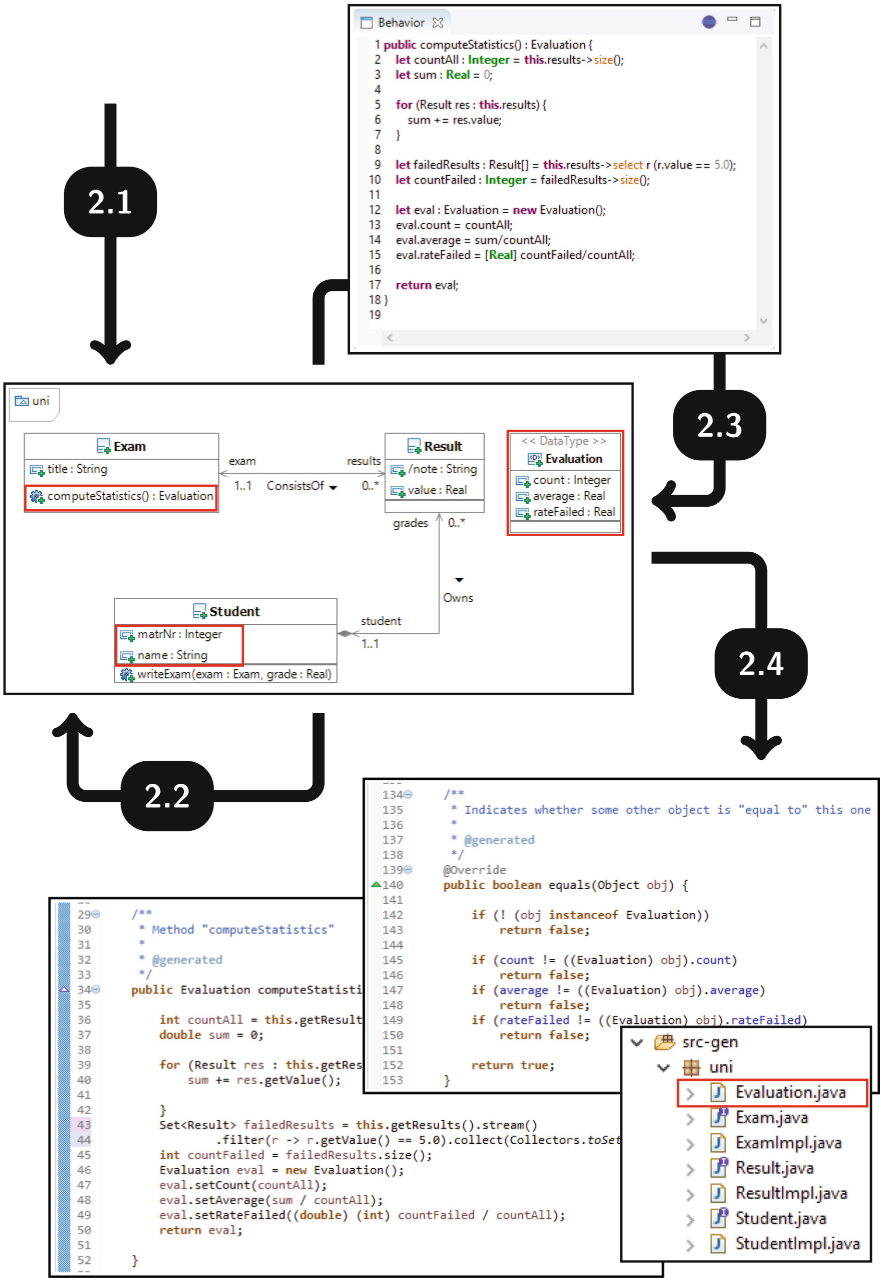
Fig. 3. The first iteration of the example workflow. Class diagram elements that are colored red are non-fUML elements that are adapted during the process. (Color figure online)

in converting the aggregation into an ordinary, fUML-compatible association. In this scenario, the user decides to accept this simplification and executes the transformation.

After the transformation has been executed, the semantics of the ALF operations is modeled (step 1.3). By clicking the related UML element – the operation or the derived property, respectively – the user gets access to the textual representation of the related branch model within an extra view providing an embedded text editor. At this moment, the operations only consist of the structural elements, i.e., the operation head and its parameters. By completing the respective operation textually, its behavior is defined. First, the ALF operation `Result::getNote()` for the derived UML property `Result::note` is defined by means of a switch statement for mapping the real values to string representations. Furthermore, the ALF operation `Student::writeExam(...)` for the related UML operation creates a new instance of the class `Result` and inserts it into the model system by means of link operation expressions for creating links.

Now, the structure as well as the behavior of our model is defined, i.e., it is fully executable. By invoking the related command, Java source code is generated (step 1.4). For the complete class diagram, one package is created that contains several Java interfaces and classes. For each UML class, there is one Java interface and one Java class that implements the respective interface; by means of generating interfaces and implementing classes, the obstacle of expressing multi-inheritance in UML with Java is overcome. Since in this case, the ALF switch statements cannot be expressed by completely analogous Java switch statements (underlying version: Java 1.8), Java if statements are used to express this semantics. The link operation expressions are mapped to invocation expressions of special Java methods that are generated in the Java classes that correspond to UML member classes of the related association.

Figure 4 depicts the second iteration. At the beginning, the structural model is extended (step 2.1). The resulting UML class diagram has a new data type `Evaluation` for storing statistical information about exams. It is used by the new operation `Exam::computeStatistics()` that returns an instance of this data type. Furthermore, the class `Student` contains two properties where the property `Student::matrNr` is set to be read-only. By executing the background transformation again, the modifications of the UML model are propagated to the ALF model system (step 2.2), i.e., it is not built up from scratch again but it is augmented by new elements such that the behavioral ALF elements that have been defined within the preceding iteration are still contained. The model system exhibits two additional ALF operations – one corresponds with the new UML operation and one with the read-only property `Student::matrNr`. Properties in fUML cannot be set to read-only and are therefore not supported by ALF. The semantics is approximated by mapping UML read-only properties to private ALF properties and ALF access operations to read the values; i.e., while the values can be modified within their respective classes by setting the property, from outside only the getter operations can be accessed since the visibility of



```
public computeStatistics() : Evaluation {
  let countAll : Integer = this.results->size();
  let sum : Real = 0;

  for (Result res : this.results) {
    sum += res.value;
  }

  let failedResults : Result[] = this.results->select r (r.value == 5.0);
  let countFailed : Integer = failedResults->size();

  let eval : Evaluation = new Evaluation();
  eval.count = countAll;
  eval.average = sum/countAll;
  eval.rateFailed = [Real] countFailed/countAll;

  return eval;
}
```

```
/**
 * Indicates whether some other object is "equal to" this one
 *
 * @generated
 */
@Override
public boolean equals(Object obj) {

    if (! (obj instanceof Evaluation))
        return false;

    if (count != ((Evaluation) obj).count)
        return false;
    if (average != ((Evaluation) obj).average)
        return false;
    if (rateFailed != ((Evaluation) obj).rateFailed)
        return false;

    return true;
}
```

```
/**
 * Method "computeStatistics"
 *
 * @generated
 */
public Evaluation computeStatisti

    int countAll = this.getResult
    double sum = 0;

    for (Result res : this.getRes
        sum += res.getValue();

    }
    Set<Result> failedResults = this.getResults().stream()
            .filter(r -> r.getValue() == 5.0).collect(Collectors.toSet
    int countFailed = failedResults.size();
    Evaluation eval = new Evaluation();
    eval.setCount(countAll);
    eval.setAverage(sum / countAll);
    eval.setRateFailed((double) (int) countFailed / countAll);
    return eval;

}
```

Fig. 4. The second iteration of the example workflow. Elements that are colored red are additional class diagram artifacts or source code files which are created during the second iteration. (Color figure online)

the UML property – in this case `public` – corresponds with the visibility of the ALF getter operation.

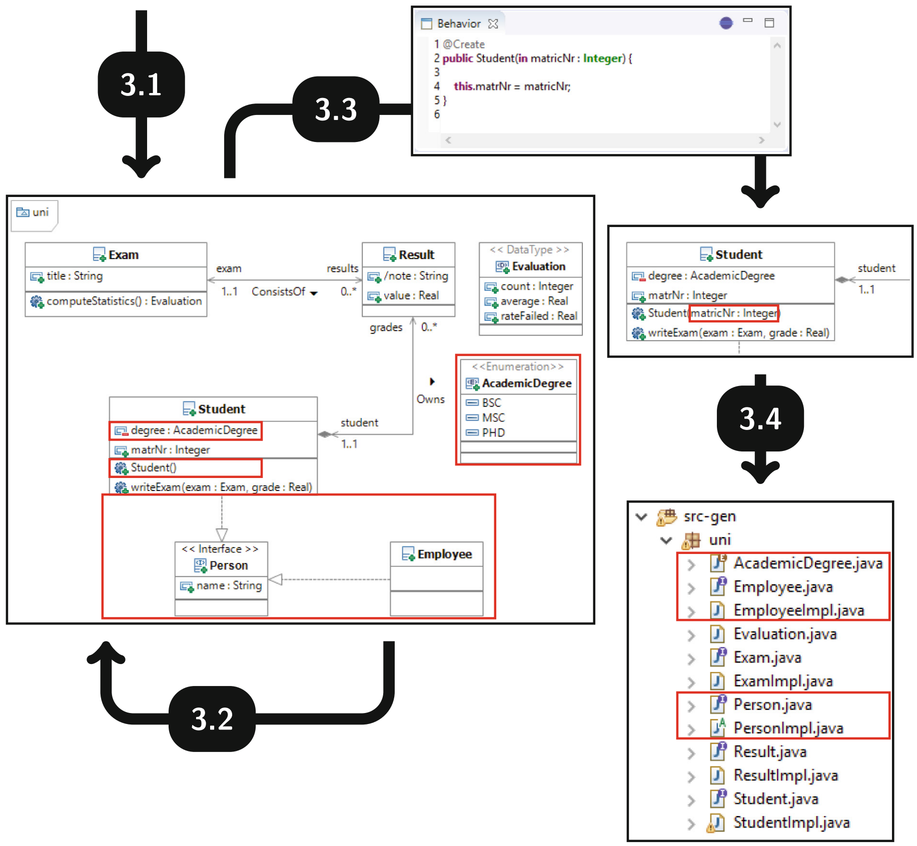
After the ALF model has been adapted, the user defines the behavior for the new ALF operation that has emerged from the UML operation (step 2.3). Sequence operation expressions are used for the computation of the `count` value, a for statement during the process for the `average` value, and a sequence expansion expression to find the correct subset of the collection `results` that is important for computing the `rateFailed` value. Within the integrated view, the user does not have any possibility to modify the ALF operation for the UML read-only property; the access operation is supposed to purely return the base value without any further effects.

Next, source code is generated for the current ALF model (step 2.4). The Java files are regenerated from scratch since the user has not modified them after the generation process at the end of the first iteration. While the generated package of the first iteration contains six files – one Java interface and one Java class for each UML class – the package currently has an additional file – one Java class for the UML data type `Evaluation`. As the behavior of the ALF operations defined during the first iteration is still contained in the ALF model system after the second iteration, the generated method bodies are also still part of the generated code. In addition, a method for the ALF operation emerged from the UML operation `Exam::computeStatistics()` is generated. ALF expressions working on sequences are mapped to corresponding Java operation calls that work on Java collections; for functional operations emerged from ALF sequence expansion expressions – e.g., `select` –, in the generated code the collections are converted into Java streams that provide analogous operations – e.g., `filter`.

The third iteration (cf. Fig. 5) starts with modifying the UML class diagram again (step 3.1). A new class `Employee` is introduced. Obviously, the property `Student::name` is not specific to students but is general for persons – also for employees, for instance. Due to this observation an interface `Person` is added that comprises the commonalities of students and employees; the property `name` is moved from the class `Student` to the interface `Person` and adequate interface realizations are inserted in the classes `Student` and `Employee`. The enumeration `AcademicDegree` is created and the class `Student` has the new private property `degree` of this type. Furthermore, the class `Student` has the new operation `Student(...)` which is supposed to initialize a student object by setting the property `Student::matrNr`.

During the propagation of the new modifications to the ALF model system (step 3.2), another approximation of non-fUML elements is performed. Within this iteration, the class diagram has been augmented with an interface. Interfaces are not contained in the fUML subset and can therefore not be mapped to directly corresponding ALF elements; instead, UML interfaces are mapped to abstract ALF classes and their realizations to corresponding generalizations.

Now, the new operation can be implemented (step 3.3); in its textual syntax, the operation can be identified to be a constructor by means of the annotation `@Create`. The constructor is supposed to assign an initial value to the



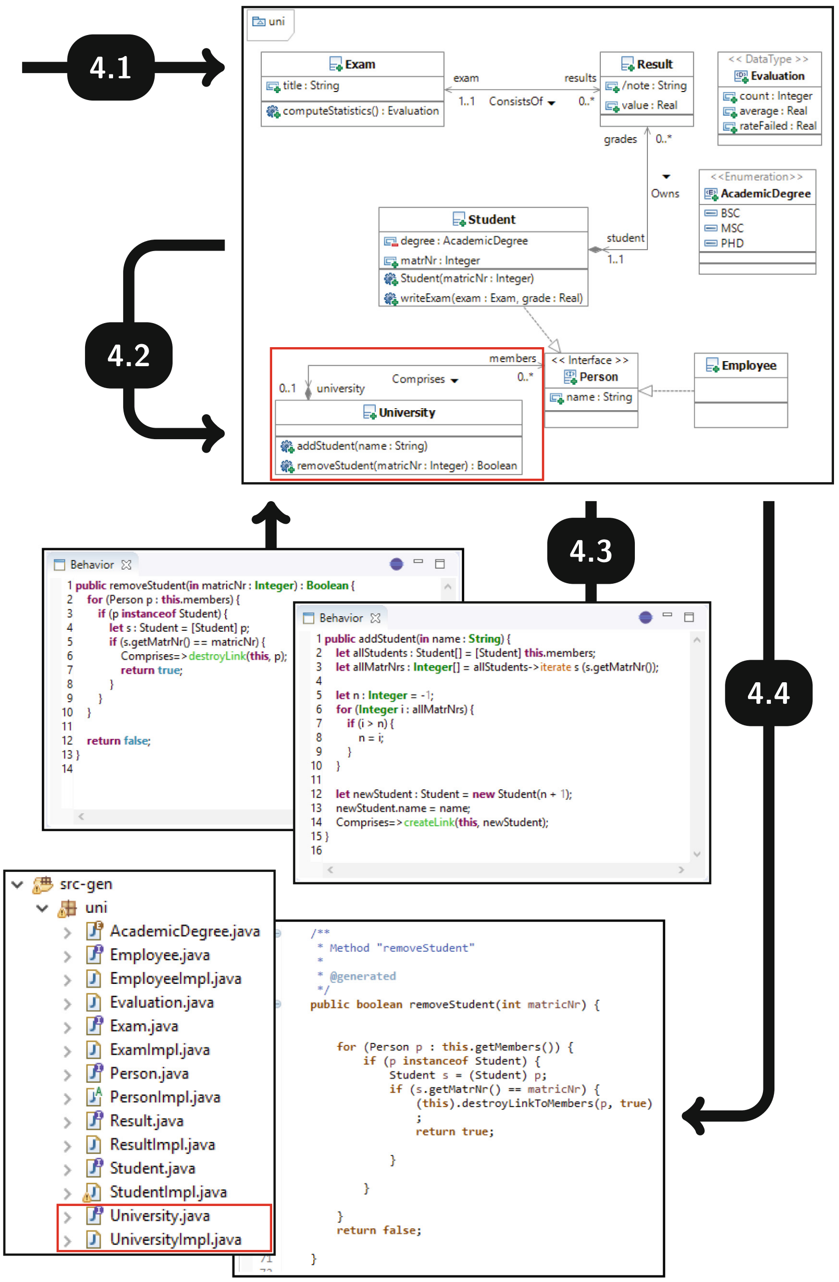
```
@Create
public Student(in matricNr : Integer) {

  this.matrNr = matricNr;
}
```

Fig. 5. The third iteration of the example workflow. Elements that are colored red are additional class diagram artifacts or source code files which are created during the third iteration. (Color figure online)

property `Student::matrNr`. At this point, the user recognizes that the constructor needs an input parameter and the signature modeled in the class diagram is not complete as there is no input parameter defined at all. Since the text editor represents the complete ALF operation – including head and in particular its parameter –, this modification can also be performed now. In the operation body, the value of the parameter is assigned to the property. While completing and saving the ALF operation modification process, the edit within the parameter list is incrementally propagated to the UML model and as a result, it is also visible in the UML diagram.

At the end of the third iteration, source code is generated (step 3.4). The source package contains five new files: one Java interface and one Java class for each of both the new UML class – that was transformed to an ALF class – and the new UML interface – that was also transformed to an ALF class – and one Java enumeration for the UML enumeration. The Java interface `Person` for the UML class `Person` is a super-interface of both the Java interfaces `Student` and



```
public removeStudent(in matricNr : Integer) : Boolean {
  for (Person p : this.members) {
    if (p instanceof Student) {
      let s : Student = [Student] p;
      if (s.getMatrNr() == matricNr) {
        Comprises=>destroyLink(this, p);
        return true;
      }
    }
  }

  return false;
}
```

```
public addStudent(in name : String) {
  let allStudents : Student[] = [Student] this.members;
  let allMatrNrs : Integer[] = allStudents->iterate s (s.getMatrNr());

  let n : Integer = -1;
  for (Integer i : allMatrNrs) {
    if (i > n) {
      n = i;
    }
  }

  let newStudent : Student = new Student(n + 1);
  newStudent.name = name;
  Comprises=>createLink(this, newStudent);
}
```

```
    /**
     * Method "removeStudent"
     *
     * @generated
     */
    public boolean removeStudent(int matricNr) {


        for (Person p : this.getMembers()) {
            if (p instanceof Student) {
                Student s = (Student) p;
                if (s.getMatrNr() == matricNr) {
                    (this).destroyLinkToMembers(p, true)
                    ;
                    return true;

                }

            }

        }
        return false;

    }
```

Fig. 6. The fourth iteration of the example workflow. Elements that are colored red are additional class diagram artifacts or source code files which are created during the fourth iteration. (Color figure online)

Employee. The Java class PersonImpl for the UML class Person is a superclass for both the UML classes StudentImpl and EmployeeImpl. From the ALF constructor, a Java constructor is generated that can be called to create new instances of this class.

Finally, a fourth iteration follows (cf. Fig. 6) which starts again with adding UML elements to the class diagram (step 4.1). The university itself is modeled by means of a UML class University that is connected to the interface Person by means of a composition. For the purpose of managing students, two UML properties are inserted in the new class University. The UML operation addStudent(...) adds a student with the name value specified as parameter. The UML operation removeStudent(...) removes the student with the matrNr value specified as parameter and returns true if and only if there exists an appropriate object; otherwise, false is returned.

During the transformation process (step 4.2), two new branch models for the two UML operations are created. The behavior is implemented (step 4.3). Both operations addStudent(...) and removeStudent(...) use for statements in order to iterate over collections. While the operation removeStudent(...) uses the reference University::persons itself as its underlying collection, the operation addStudent(...) converts the collection of persons to a collection of integers representing the students' matriculation numbers; for that purpose, a cast expression is used to filter the Student objects within the collection of Person objects and a sequence expansion expression (iterate) is used to map the collection of students to the collection of their matrNr values. For adding and removing the links, both operations use adequate link operation expressions. Since the matrNr value is supposed to be unique among all Student objects contained in some University object, the Student object created by the operation addStudent(...) gets the currently maximum value incremented by one as its matrNr value.

Listing 1. JUnit test cases for the generated Java code.

```
@Test public void testStudentsCreation() {
    University university = new UniversityImpl();
    assertTrue(university.getStudents().isEmpty());
    university.addStudents("Alice");
    university.addStudents("Bob");
    assertEquals(2, university.getMembers().size());
    assertTrue(university.removeStudent(1));
    assertFalse(university.removeStudent(1));
    assertTrue(university.removeStudent(0));
    assertTrue(university.getStudents().isEmpty());
}

@Test public void testStudentsCreation() {
    University university = new UniversityImpl();
    university.addStudent("Alice");
    Student s = (Student) university.getMembers().iterator().next();
    Exam exam = new ExamImpl();
    assertTrue(exam.getResults().isEmpty());
    assertTrue(s.getGrades().isEmpty());
    s.writeExam(exam, 3);
    assertTrue(!exam.getResults().isEmpty());
    assertTrue(!s.getGrades().isEmpty());
}
```

At the end of the fourth iteration, Java code is generated for the last time (step 4.4). For the new UML class, two Java files are generated. The ALF link operation expressions for creating and destroying links are mapped to invocations of special Java methods that are generated in the Java classes corresponding to the UML members classes of the new composition.

During four iterations a model has emerged that contains structural as well as behavioral elements from which fully-executable source code has been generated that does not require any further user editing in order to use it by further programs. As an application for the generated Java source code, several test cases are provided. Listing 1 depicts two JUnit test methods that check the correct semantics of the generated Java source code, in particular of the generated Java methods that have emerged from the specified ALF operations.

4 Integration of UML and ALF

This section describes the implementation background concerning the technical as well as the visual integration of the textual ALF editor and the graphical UML-based modeling tool *Valkyrie* [1]. The diagram editor within the Valkyrie environment was created using GMF; *GMF* (*Graphical Modeling Framework*)[1] generates projectional editors as this is a typical architecture for graphical editors: The underlying model – in case of Valkyrie the UML model – and the diagram that constitutes a view onto the model – in this case the class diagram – are separated into two files. When the user edits the model using editor commands – e.g., the name of a model element is modified –, the underlying model is modified and afterwards, the changes are propagated to the diagram.

In contrast to the projectional GMF-based editor, the textual ALF editor was built using *Xtext*[2] and therefore constitutes a parser-based editor: The text files are persisted and a parser creates an in-memory model representation for each text file; the resulting model is a temporary artifact and is not persisted within an additional file. The visual integration combines the projectional graphical UML class diagram editor for modeling structure and the parser-based textual ALF editor for modeling behavior as well as a bidirectional and incremental synchronization between them.

4.1 Overview of the Tool Chain

Figure 7 shows an overview of models involved in the tool chain of our approach. The graphical editor shows the UML class diagram (1) that corresponds to an underlying UML model (2). Since the graphical editor is a projectional editor, the user modifies the underlying model directly and the changes are propagated to the diagram.

The UML model is involved in the background model transformation. A bidirectional and incremental transformation (3, cf. Sect. 4.4) converts it into

[1] https://www.eclipse.org/modeling/gmp/.
[2] https://www.eclipse.org/Xtext/.

an ALF model system (4, cf. Sect. 4.2) – corresponding to the UML model, but containing several models – and vice versa. The ALF model system is augmented with behavioral elements by means of the textual ALF editor (5). For each UML element within the class diagram, a counterpart within the ALF model system is present. As a result, the for ALF editor functions – e.g., content assist proposals – as well as the code generation process (6, cf. Sect. 4.3) can be limited to the ALF system since no information from the UML model is required.

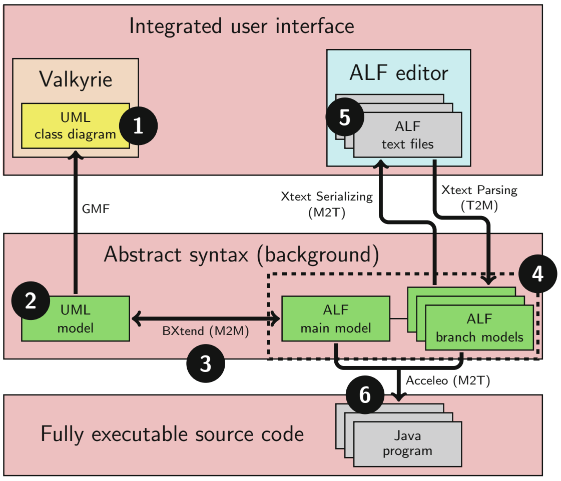


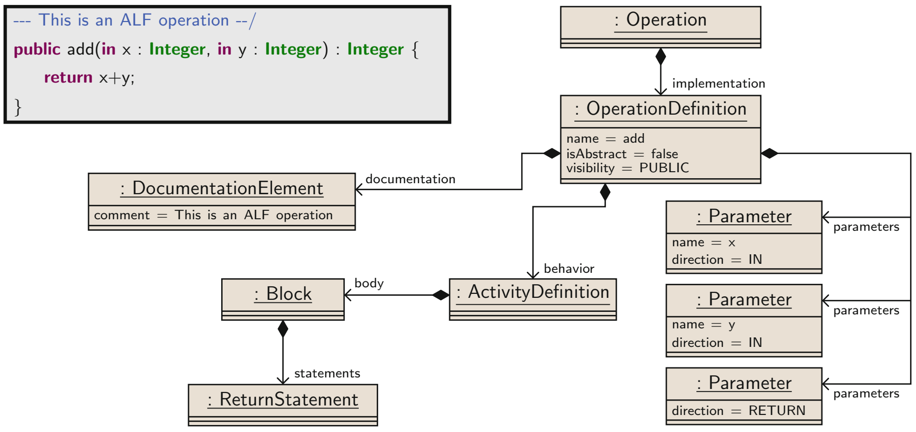
Fig. 7. An overview of the tool chain. Based on [20, Fig. 3].

The basic idea of the visual integration within an integrated user interface provides a textual editor that only shows an ALF operation and not the complete ALF model. The ALF model system comprises several models: one main model – which contains most of the structural elements corresponding to the UML model aside from operation contents as parameters – and several branch models – which comprise apart from structural artifacts of the ALF operations all the behavioral model elements which are not stored in the UML model.

From the branch models, the textual representations of the ALF operations – that are visible in the text editor – are created by means of reusing the Xtext serialization process. The serialization process works incrementally, i.e., if a textual representation of the ALF operation already exists, the body is not affected; thus, custom layout and comments are preserved. When the text files are modified, a parsing process propagates the changes to the respective branch model. Within this process, the abstract syntax tree that is build temporarily by the Xtext parser is used to store the mentioned changes permanently within the ALF model system; when this process is performed, the contents of the branch models are changed but no new resource is created such that the references between the main model and the branch models are not affected. Finally, the code generator that is implemented using Acceleo[3] creates Java files for ALF model elements. The code generator retrieves information from the main model as well as the branch models; as a result, fully executable source code is generated that contains complete method bodies without requiring any manual code extensions by a user.

4.2 The ALF Model System

According to the standard [15], an ALF model consists of structural elements – packages, classes, data types, associations, properties, and operations – as well as behavioral elements – different kinds of statements within the activity definitions of the respective operations – and is self-contained. Thus, a model resource with exactly one root model element contains all children model elements. The basic idea of this approach is the division of the complete ALF model into several models each of which is stored within its own resource and constitutes a certain portion of the complete model; the single models are connected by means of inter-model cross references resulting in an ALF model system. In order to achieve the goal of an integrated user interface, each ALF operation is stored within its own model. Hence, an ALF model system consists of several branch models – each of which contains the structural and behavioral elements of one ALF operation – and one main model – that contains all the structural model elements that are not contained in the branch models, i.e., packages, classes, data types, associations, and properties – with inter-model references from the main model to the branch models in order to access the operations from their owning classes.



```
--- This is an ALF operation --/
public add(in x : Integer, in y : Integer) : Integer {
    return x+y;
}
```

Fig. 8. A textually represented example ALF operation and its abstract syntax tree (simplified).

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

To this end, the underlying ALF metamodel resulting from the official standard was slightly modified in order to conform to our approach. The standard provides one metaclass `Operation` for ALF operations. The access to the surrounding context – represented by the metaclass `Class` – is provided by means of a container reference – called `ownerClass` here. The adapted metamodel that is used for this approach comprises three metaclasses for representing ALF operations:

– The metaclass `OperationDefinition` contains all the information about operations that is necessary in order to serialize them within a text file. This includes attribute values – e.g., the name and the visibility –, documentation, contained formal parameters – represented by the metaclass `Parameter` –, and the contained behavior – represented by the metaclass `ActivityDefinition`.
– The metaclass `Operation` serves as the root class for the branch models. Instances of `OperationDefinition` are contained in respective instances of `Operation`. For technical reasons, not the metaclass `OperationDefinition` serves as the root class; instead, an additional metaclass was introduced. The textual representation of an `Operation` instance only comprises an initially empty text file; if the textual representation contains an ALF operation representation, this corresponds to an `OperationDefinition` instance that is contained in the `Operation` instance.
– Finally, the adapted metamodel contains the metaclass `OperationNode` that provides the container reference `ownerClass`. Its instances are contained in the main model and serve as placeholders for the operations that are contained in the branch models and accessed by means of the inter-model cross reference `operation` with the metaclass `OperationNode` as its source and `Operation` as its target.

Figure 8 depicts the textual representation of an example ALF operation and the underlying model. The root `Operation` contains the `OperationDefinition` object – represented by the whole text fragment – that contains three parameters, some documentation, and a `ReturnStatement` object as a behavioral element.

Figure 9 shows a complete ALF model system consisting of a main model – with a `Model` instance as its root – and two branch models. The main model contains the classes, enumerations, properties, and `OperationNode` instances that provide links to the root objects of the branch models – `Operation` instances. The branch models contain the operations – `OperationDefinition` instances – as well as their parameters and the respective body – represented by an `ActivityDefinition` instance.

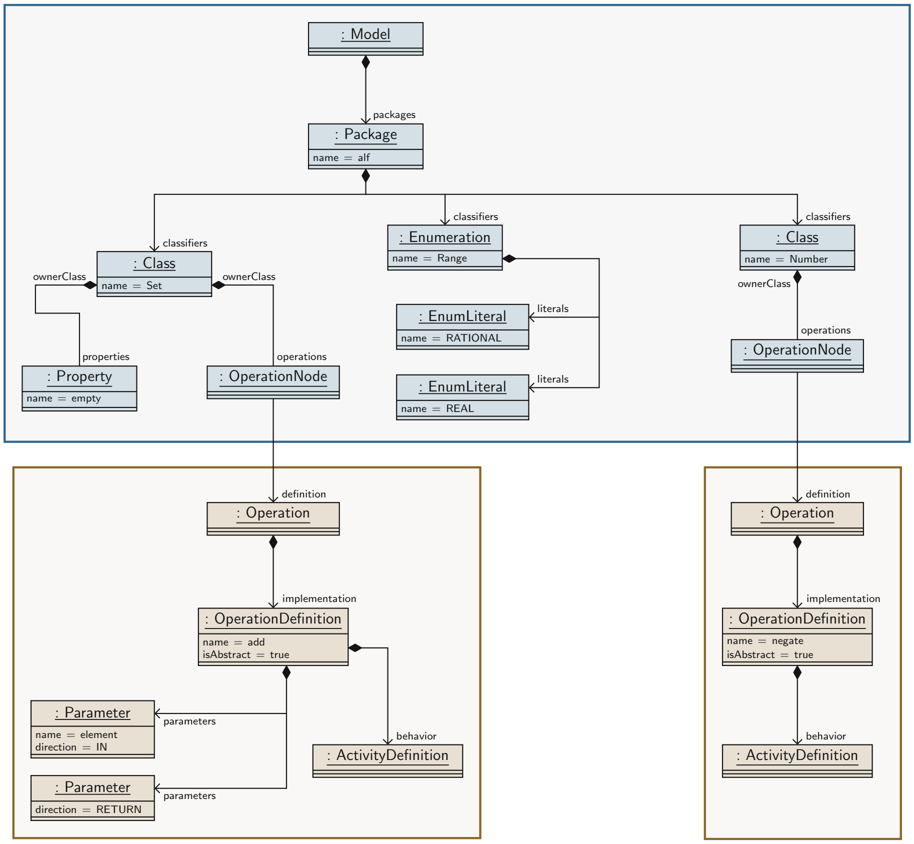


Fig. 9. An example ALF model system (simplified). Each model persisted within its own resource is surrounded by a rectangle; elements of the main model are colored blue, elements of the branch models brown. (Color figure online)

4.3 Generation of Java Source Code

In order to execute the modeled software system, Java source code is generated using a model-to-text transformation. For this purpose, we use *Acceleo*[4] that allows to express the transformation by templates and queries pretty intuitively using the *Object Constraint Language* (*OCL*) [17]. It constitutes a pragmatic implementation of the *MOF Model to Text Transformation Language* (*MOFM2T*) [14] standard. Acceleo allows for an integration of Java code: Queries can be defined by invoking Java methods besides expressing the queries exhaustively with OCL expressions. For this approach, "Java services" are used to access other parts of the implementation, e.g., the ALF type system used for the text editor.

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

For a given ALF model system, the Acceleo templates use the main model as well as all the branch models for generating fully executable source code. The contained ALF classifiers are successively transformed to Java files – ALF classes to pairs of Java interfaces and classes, ALF data types to Java classes, and ALF enumerations to Java enumerations. By accessing the branch models, the body implementations of the ALF operations are transformed to corresponding Java method bodies. Functional ALF expressions that cannot be expressed by completely analogous Java 1.8 constructs – e.g., operations for filtering collections – are mapped to Java operation calls that work on streams. Java streams are sequences of elements supporting sequential and parallel aggregation operations as filtering and mapping methods.

4.4 The Kernel Model-to-Model Transformation

The kernel of the tool chain constitutes the model transformation between UML models and ALF model systems. It was implemented using the *BXtend* approach [3]; a physically persisted correspondence model contains objects which represent the correspondence links between UML and ALF model elements. The transformation is bidirectional: An arbitrary UML model is transformed to an ALF model system that contains ALF elements that approximately express the semantics given by the UML model and vice versa.

Since ALF only supports fUML which is a proper subset of the complete UML metamodel, some UML elements cannot be transformed into completely corresponding ALF element; using alternate mappings an approximation of the UML elements in question by several ALF elements is performed such that significant non-fUML elements often used in practice are also supported by this approach (cf. Table 1).

Table 1. Mapping significant non-fUML elements to appropriate ALF elements [20].

UML model elements	Alternate ALF model elements
Derived property	Getter operation
Read-only property	Property + getter operation
Interface	Abstract class
Interface realization	Generalization

Furthermore, the transformation works incrementally, i.e., in case a UML model and a corresponding ALF model system already exist, model changes are propagated to the respective opposite model rather than creating those models from scratch again. This is an important feature for software development processes in practice which consist of several incremental iterations until the software system reaches its final state. On the one hand, user-supplied method bodies have to be retained when the UML model is transformed. On the other hand, in general UML model elements are referenced by other models such that if they were replaced, links from other models would get lost; in case of the UML tool presented here, those links exist from the diagram file to the UML model which would become invalid if underlying UML model elements were replaced.

The transformation definition comprises a sequence of bidirectional transformation rule calls each of which is applied to all appropriate model elements. By means of a boolean expression that at least contains a check for the correct type, the considered subset of model elements for a certain transformation rule and direction is filtered; for each model element at most one transformation rule exists for which the related boolean expression is satisfied. The execution order of the transformation rules ensures that if a model element is transformed, its container element (if any) has already been transformed resulting in a top-down traversation. The rules are applied to main model elements as well as branch model elements. For ALF operations, the respective `OperationNode` instance is called from which the respective branch model is accessed. The single rules are applied to the model elements in the following order:

1. The model roots are considered. Each UML model root instance corresponds to an ALF model root instance with a contained root package.
2. The rules **UmlClass2AlfClass** and **UmlInterface2AlfClass** are applied. UML interfaces are approximated by means of abstract ALF classes.
3. The transformation rules for associations, structured data types, enumerations, and enumeration literals are applied.
4. For UML properties and operations, three rules are provided: The rule **Uml-Property2AlfProperty** is applied to UML properties that are not derived – they correspond to ALF properties and (if they are read-only properties) to an accessing getter operation –, the rule **UmlProperty2AlfOperation** to derived UML properties – they correspond to ALF getter operations – and finally the rule **UmlOperation2AlfOperation** for UML operations – corresponding to ordinary ALF operations.
5. Parameters are considered. For technical reasons, two different rules are provided depending on the direction of the parameter.
6. UML comments are transformed to appropriate ALF documentation.
7. The rules **UmlGeneralization2AlfGeneralization** and **UmlInterface-Realization 2AlfGeneralization** are applied. Due to the approximation of UML interfaces by means of abstract ALF classes, UML interface realizations lead to ALF generalizations.

Listing 2. Xtend class for the bidirectional transformation rule **UmlAssociation2AlfAssociation**.

```
class UmlAssociation2AlfAssociation extends UmlElem2AlfElem {
    // constructors omitted ...

    override sourceToTarget(String desc) { // UML2ALF
        sourceModel.eAllContents.filter(typeof(UmlAssociation)).forEach[a |
            val target = a.getOrCreateCorrModelElement(desc)
                    .getOrCreateTargetElem(ALF_ASSOCIATION) as AlfAssociation

            target.isAbstract = a.isAbstract
            target.name = a.name
            target.visibility = a.visibility.umlVisibilityToAlfVisibility;

            (a.eContainer.corrModelElem.targetElem as AlfPackage)
                    .classifiers += target
        ]
        super.sourceToTarget(desc)
    }

    override targetToSource(String desc) { // ALF2UML
        // completely symmetrical to sourceToTarget(...)
    }
}
```

The Xtend classes for the transformation rules have the same structure. They specialize the abstract super-class `UmlElem2AlfElem` and redefine the methods `sourceToTarget(...)` – transforming UML elements to corresponding ALF elements – and `targetToSource(...)` – transforming ALF elements to corresponding UML elements. The Xtend class for the transformation rule **UmlAssociation2 AlfAssociation** is depicted in Listing 2. Since both methods have the same structure, we only consider the method `sourceToTarget(...)`, i.e., the direction UML to ALF. The body starts with filtering the correct elements; in this case, all UML model elements that are associations and therefore have the correct type are considered. Each selected element `a` is transformed now: If it does not already exist a corresponding `target` element, a new object is created; otherwise, it exists a correspondence element for `a` and `target` such that the `target` object can be accessed. Next, the attribute values are transformed. Finally, the `target` element is added to its container (if it is not already contained) – the corresponding element for the container of `a` – that has already been transformed due to the execution order of the rules. After this iteration, the super-method is called that performs general clean-up operations as deleting model elements that have no correspondence elements any more.

Listing 3. Xtend class for the bidirectional transformation rule **UmlOperation2AlfOperation**.

```
class UmlOperation2AlfOperation extends UmlElem2AlfElem {
    // constructors omitted ...

    override sourceToTarget(String desc) { // UML2ALF
        sourceModel.eAllContents.filter(typeof(UmlOperation)).forEach[o |
            val target = a.getOrCreateCorrModelElement(desc)
                    .getOrCreateTargetElem(ALF_OPERATION_NODE)
                    as AlfOperationNode
            val opDef = target.getOrCreateOperationDefinition(o)

            opDef.isAbstract =
                    o.eContainer instanceof UmlInterface || o.isAbstract
            // some more assignments omitted  ...

            // setting container omitted ...
        ]
        super.sourceToTarget(desc)
    }

    override targetToSource(String desc) { // ALF2UML
        targetModel.eAllContents.filter(typeof(AlfOperationNode)).forEach[o |
            val opDef = o.operation.implementation

            if (opDef.isGetter) {
                val source = o.getOrCreateCorrModelElement(desc)
                        .getOrCreateTargetElem(UML_OPERATION)
                        as UmlOperation

                source.isAbstract = opDef.isAbstract
                // some more assignments omitted  ...

                // setting container omitted ...
            }
        ]
        super.targetToSource(desc)
    }
}
```

Since the ALF model system consists of a collection of models each of which represents a certain portion of the context, executing transformation rules may come along with creating and deleting models. If a UML operation is transformed and a corresponding ALF `OperationNode` instance does not exist yet, the rule does not only have to create a new `OperationNode` object but also the branch model for this operation. Analogously, if a UML operation is removed, the respective branch model has to be deleted. Listing 3 shows the Xtend class for the transformation rule **UmlOperation2AlfOperation**. During the iteration over the `UmlOperation` within the method `sourceToTarget(...)`, for each object `o` not only the corresponding `AlfOperationNode` object but also its `AlfOperationDefinition` object is considered. This is performed by calling the operation `getOrCreateOperationDefinition(...)` returning the linked `AlfOperationDefinition` instance; if the branch model does not exist yet, it is created and the links are set correctly. The assignments – apart from setting the container – use the `AlfOperationDefinition` object directly. Within the method `targetToSource(...)` the branch model is directly accessed; since an `OperationNode` instance requires a transitively contained `OperationDefinition` child object, it is ensured that it exists.

4.5 The Integrated User Interface

One significant goal for the integrated modeling tool was that the integration is not reduced to a technical combination of both languages UML and ALF but also comprises the user interface constituting a visual integration of different editors such that an easy and fluent usage is feasible. Although a pretty wide range of models are involved in the background processes, the user should get the feeling of editing one model instead of a collection of models where each of them represents a certain portion of the context. This section describes the foundations of the implementation with respect to the user interface.

In order to facilitate an integrated user interface, an Eclipse view was created that provides the textual modifications of the ALF operations. While the class diagram is visible within the graphical editor which constitutes the main editor where the user edits the structure, the behavior is modified textually within the additional view.

Xtext provides tool support to embed generated editors within SWT composites which is used for our tool to embed the text editor within the Eclipse view. The view contents depend on the user's actions in the main editor. If the user clicks an operation or a derived property within the class diagram, the view is notified about the respective edit part and shows the embedded editor with the textual representation of the corresponding ALF operation. The complete ALF operation is now visible and can be edited as in case of a usual text editor; apart from behavioral modeling concerning the operation body, also the structural model elements related to the operation – i.e., the name, the visibility, the parameter list, and documentation – can be modified textually. The view provides a button for finishing and persisting the current modification of the respective ALF operation; when the button is clicked, the respective text file is saved, the parsing process of the ALF model is performed and eventually the ALF-to-UML transformation is induced such that the structural changes of the respective operation get visible within the class diagram. Thus, the integrated tool provides round-trip engineering with respect to structural elements of operations; all other structural model elements are edited within the diagram editor while behavior can only be edited textually.

5 Discussion

In this section, the approach presented in this paper is discussed. First, the resulting benefits are given:

Fully Executable Models. This approach overcomes the code generation dilemma that occurs when model and hand-written code evolve independently. Specifying both structure and behavior leads to a generation of fully executable source code that can be used by further programs and does not require any user interaction or code modifications afterwards; any information contained in the model is mapped adequately to the resulting source code automatically.

Convenient Notation. Our approach combines modeling structure and behavior using two different modeling languages and two different paradigms of editing models. On the one hand, the projectional diagram editor provides convenient graphical notation for modeling structural elements. On the other hand, behavior is added textually by means of the parser-based editor instead of using another graphical editor; while graphical notation for behavioral model elements can result in very large and confusing diagrams, using textual syntax results in concise model representations that are easy to read and understand.

Visual Integration. Modeling structure using UML diagrams and behavior using ALF text is not only combined technically but also visually: The different editors are combined in the modeling environment by means of appropriate Eclipse concepts. The user gets the feeling of editing one model instead of dealing with a collection of models that are involved in the background.

Interlinked Model System. Another conceivable approach to overcome the code generation dilemma could provide for modeling the structure and adding the behavior by means of code snippets in terms of plain text comments in the UML model. By contrast, our approach contains all the information of the modeled system in several models within an interlinked system; hence, all the artifacts used for the code generation – in particular the ALF operations – are persisted in terms of models within the ALF model system. In contrast to plain code snippets, cross links can be exploited to find model elements and text editor mechanisms as a content assist can be used.

Flexible Workflow. The kernel transformation converting UML models to ALF model systems and vice versa is bidirectional and incremental. Thus, a very flexible workflow is supported that allows for development processes consisting of several iterations of editing structural and behavioral model elements.

These aspects emphasize in particular the benefits of using ALF as the underlying language for expressing behavioral elements. However, using ALF comes along with a significant drawback with respect to expressiveness: Although by means of ALF a quite large range of model elements can be expressed, only a proper subset of UML is supported; thus, some elements – e.g., interfaces – cannot be expressed exactly. Nevertheless, the semantics of non-fUML elements often can be approximated pretty well using alternate components – e.g., abstract classes instead of interfaces – such that in practice, the limited expressiveness resulting from using ALF does not restrict the modeling process too hard.

6 Related Work

In the past, several tools relying on textual or graphical syntax, or even a combination thereof have been published aiming for addressing model-driven development with special emphasis on modeling behavior. While some of them are equipped with code generation capabilities, others only allow for creating models and thus only serve as a visualization tool.

Fujaba [23] is a graphical modeling language based on graph transformations which allows to express both the structural and the behavioral part of a software system on the modeling level. Furthermore, Fujaba provides a code generation engine that is able to transform the Fujaba specifications into executable Java code. Behavior is specified using *story diagrams*. A story diagram resembles UML activity diagrams where the activities are described using *story patterns*. A story pattern specifies a graph transformation rule where both the left hand side and the right hand side of the rule are displayed in a single graphical notation. While story patterns provide a declarative way to describe manipulations of the run time object graph on a high level of abstraction, the control flow of a method is on a rather basic level as the control flow in activity diagrams is on the same level as control flow diagrams. As a case study [8] revealed, software systems only contain a low number of problems which require complex story patterns. The resulting story diagrams nevertheless are big and look complex because of the limited capabilities to express the control flow.

Another textual modeling language, designed for *model-oriented programming*, is provided by **Umple**[5]. The language has been developed independently from the EMF context and may be used as an Eclipse plug-in or via an online service. In its current state, Umple allows for structural modeling with UML class diagrams and describing behavior using state machines. A code generation engine allows to translate Umple specifications into Java, Ruby, or PHP code. Umple scripts may also be visualized using a graphical notation. Unfortunately, the Eclipse-based editor only offers basic functions like syntax highlighting and a simple validation of the parsed Umple model. Umple offers an interesting approach which aims at assisting developers in rasing the level of abstraction ("umplification") in their programs [12]. Using this approach, a Java program may be stepwise translated into an Umple script. The level of abstraction is raised by using Umple syntax for associations.

Xcore[6] recently gained more and more attention in the modeling community. It provides a textual concrete syntax for Ecore models allowing to express the structure as well as the behavior of the system. In contrast to ALF, the textual concrete syntax is not based on an official standard. Xcore relies on Xbase – a statically typed expression language built on Java – to model behavior. Executable Java code may be generated from Xcore models. Just as the realization of ALF presented in this paper, Xcore blurs the gap between Ecore modeling and Java programming. In contrast to ALF, the behavioral modeling part of Xcore has a strongly procedural character. As a consequence an object-oriented way of modeling is only possible to a limited extent. For instance, there is no way to define object constructors to describe the instantiation of objects of a class. Since Xcore reuses the EMF code generation mechanism [22], the factory pattern is used for object creation. Furthermore, ALF provides more expressive power since it is based on fUML while Xcore only addresses Ecore.

The graphical UML modeling tool **Papyrus** [10] allows for creating UML, SysML, and MARTE models using various diagram editors. Additionally, Papyrus offers dedicated support for UML profiles which includes customizing the Papyrus UI to get a DSL-like look and feel. Papyrus is equipped with a

[5] http://cruise.site.uottawa.ca/umple.

[6] http://wiki.eclipse.org/Xcore.

code generation engine allowing for producing source code from class diagrams (currently Java and C++ is supported). Future versions of Papyrus will also come with an ALF editor. A preliminary version of the editor is available and allows a glimpse on its provided features [11]. The textual ALF editor is integrated as a property view and may be used to textually describe elements of package or class diagrams. Furthermore, it allows to describe the behavior of activities. The primary goal of the Papyrus ALF integration is using graphical and textual syntax as alternative representations of the same view on the model and not executing behavioral specifications by generating source code. While Papyrus strictly focuses on a forward engineering process (from UML to ALF), the approach presented in this paper explicitly addresses round-trip engineering.

The commercial tool **MagicDraw UML**[7] recently also provides a plug-in allowing behavioral modeling with ALF [21]. Modelers may express bodies of activities using ALF statements which are then executed either sequentially or with considerable concurrency (depending on the underlying computation platform). The plug-in integrates the **ALF Reference Implementation**[8] into the commercial tool MagicDraw UML. In order to execute UML activity, state machine, and interaction models in MagicDraw, the **Cameo Simulation Toolkit**[9] is required. The ALF plug-in allows to define executable behavior in this context. The ALF implementation for MagicDraw compiles ALF text into activity models in the background and integrates the resulting activity models within the wider UML modeling context. Furthermore, these models may be executed as parts of full system simulation scenarios.

Compared with our own solution presented in [2], the approach discussed in this paper provides a much tighter integration of UML and ALF modeling by means of a single integrated user interface. The motivation behind our approach presented in this paper is the combination of graphical and textual modeling in an integrated tool in a way such that the most appropriate formalism is used depending on the considered model elements; while structure is represented pretty intuitively using graphical elements, behavioral model elements can be expressed very precisely by a textual language. For this purpose, only ALF operations are persisted, presented, and edited textually, i.e., all other aspects of the ALF model are hidden. The modeler may focus on the current task which results in a lower cognitive complexity exposed to the user. Furthermore, instead of providing two different editors which are not connected to each other, the ALF editor is now integrated visually in the graphical editing process. When the user clicks an operation within the UML model, a specific view shows the corresponding ALF operation containing the method body; UML model and ALF text are displayed at the same time. Additionally, some actions of the tool chain are bundled such that the user does not have to take care about the technical details running in the background.

[7] https://www.nomagic.com/products/magicdraw.
[8] http://alf.modeldriven.org.
[9] https://www.nomagic.com/product-addons/magicdraw-addons/cameo-simulation-toolkit.

7 Conclusion

In this paper, we presented an approach which allows for modeling structure as well as behavior of a software system using an integrated tool combining two OMG standards: UML and ALF. While UML package diagrams and class diagrams are used for modeling the structure, method bodies – i.e., the behavior of a method – are specified using ALF. Fully executable Java source code may be generated from the resulting model system which can be integrated seamlessly in existing software ecosystems. The integrated user interface abstracts from the underlying set of models and provides a unified look and feel for the end user allowing for graphical as well as textual modeling. The feasibility of the approach has been demonstrated using an example workflow which may occur in real-life iterative software development processes.

References

1. Buchmann, T.: Valkyrie: a UML-based model-driven environment for model-driven software engineering. In: Proceedings of the 7th International Conference on Software Paradigm Trends, ICSOFT 2012, pp. 147–157. SciTePress, Rome (2012)
2. Buchmann, T.: Prodeling with the action language for foundational UML. In: Damiani, E., Spanoudakis, G., Maciaszek, L.A. (eds.) ENASE 2017 - Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, Porto, Portugal, 28–29 April 2017, pp. 263–270. SciTePress (2017). https://doi.org/10.5220/0006353602630270
3. Buchmann, T.: BXtend - a framework for (bidirectional) incremental model transformations. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, 22–24 January 2018, pp. 336–345. SciTePress (2018). https://doi.org/10.5220/0006563503360345
4. Buchmann, T., Dotor, A., Westfechtel, B.: Model-driven software engineering: concepts and tools for modeling-in-the-large with package diagrams. Comput. Sci. - Res. Dev. 1–21. https://doi.org/10.1007/s00450-011-0201-1
5. Buchmann, T., Greiner, S.: Handcrafting a triple graph transformation system to realize round-trip engineering between UML class models and Java source code. In: Maciaszek, L.A., Cardoso, J.S., Ludwig, A., van Sinderen, M., Cabello, E. (eds.) Proceedings of the 11th International Joint Conference on Software Technologies, ICSOFT 2016 - Volume 2: ICSOFT-PT, Lisbon, Portugal, 24–26 July 2016, pp. 27–38. SciTePress (2016). https://doi.org/10.5220/0005957100270038
6. Buchmann, T., Schwägerl, F.: On a-posteriori integration of Ecore models and hand-written Java code. In: Lorenz, P., Van Sinderen, M., Cardoso, J. (eds.) Proceedings of the 10th International Conference on Software Paradigm Trends, pp. 95–102. SciTePress, July 2015. https://doi.org/10.5220/0005552200950102
7. Buchmann, T., Westfechtel, B.: Using Triple Graph Grammars to Realize Incremental Round-Trip Engineering. IET Software (July 2016). https://doi.org/10.1049/iet-sen.2015.0125
8. Buchmann, T., Westfechtel, B., Winetzhammer, S.: The added value of programmed graph transformations – a case study from software configuration management. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 198–209. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34176-2_17

9. Greiner, S., Buchmann, T., Westfechtel, B.: Bidirectional transformations with QVT-R: a case study in round-trip engineering UML class models and Java source code. In: MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19–21 February, 2016, pp. 15–27 (2016). https://doi.org/10.5220/0005644700150027
10. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., Gérard, S.: Executable modeling with fUML and Alf in Papyrus: tooling and experiments. In: Mayerhofer et al. [13], pp. 3–8
11. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., Gérard, S.: Executable modeling with fUML and Alf in papyrus: Tooling and experiments. In: Mayerhofer et al. [13], pp. 3–8. http://ceur-ws.org/Vol-1560/paper1.pdf
12. Lethbridge, T.C., Forward, A., Badreddin, O.: Umplification: refactoring to incrementally add abstraction to a program. In: 2010 17th Working Conference on Reverse Engineering (WCRE), pp. 220–224. IEEE (2010)
13. Mayerhofer, T., Langer, P., Seidewitz, E., Gray, J. (eds.): 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, Ottawa, Canada, 27 September 2015, CEUR Workshop Proceedings, vol. 1560. CEUR-WS.org (2016)
14. OMG: MOF Model to Text Transformation Language, v1.0. OMG, Needham, MA, formal/2008-01-16 edn., January 2008
15. OMG: Action Language for Foundational UML (ALF). Object Management Group, Needham, MA, formal/2013-09-01 edn., September 2013
16. OMG: Semantics of a Foundational Subset for Executable UML Models (fUML). Object Management Group, Needham, MA, formal/2013-08-06 edn., August 2013
17. OMG: Object Constraint Language. OMG, Needham, MA, formal/2014-02-03 edn., February 2014
18. OMG: Meta Object Facility (MOF) Version 2.5. OMG, Needham, MA, formal/2015-06-05 edn. (2015)
19. OMG: Unified Modeling Language (UML). Object Management Group, Needham, MA, formal/15-03-01 edn., March 2015
20. Schröpfer, J., Buchmann, T.: Unifying Modeling and Programming with Valkyrie. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, 20–22 February, pp. 27–38. SciTePress (2019). https://doi.org/10.5220/0007259600270038
21. Seidewitz, E.: A development environment for the Alf language within the MagicDraw UML tool (tool demo). In: Combemale, B., Mernik, M., Rumpe, B. (eds.) Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, 23–24 October 2017, pp. 217–220. ACM (2017). https://doi.org/10.1145/3136014.3136028
22. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Boston (2009)
23. The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth: The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. In: Giese, H., Zündorf, A. (eds.) Proceedings of the 3rd international Fujaba Days, pp. 1–13, September 2005
24. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Hoboken (2006)