Refactoring UML models: Co-refactoring fUML conform class diagrams and activity diagrams

Sebastian Geiger (1127054) and Kristof Meixner (9725208)

Business Informatics Group
Vienna Technical University

Abstract. In this work we will present ideas and concepts for the refactoring of fUML conform UML models. The main contribution of this work is the extension of existing UML refactorings. Our refactorings cover not only the static aspect of UML such as class diagrams but consider the co-refactoring of the dynamic parts such as activity diagrams as well. In this work we will present basic concepts for the refactoring of models with EMF and show how model semantics can be preserved through the use of OCL constraints. Furthermore we present our toolchain and the technologies of EMF (in particular Ecore and OCL) and how we used them for the co-refactoring of fUML conform UML models. We also present a discussion of EMF Refactor, which shows how such refactorings can be made available in the Eclipse GUIs such as the UML tree editor or the Papyrus UML editor.

Table of Contents

1 Introduction

Model-based software development or model-driven software development is not only an extensive field of research but also receives increased attention from the industry. Nowadays models are not only used as visual explanations of software concepts but as source for the development process itself. Thus models need to provide an abstraction of the represented domains in a high quality. Mohagheghi *et al.* [10] discussed quality attributes like *Correctness*, *Completeness* and *Consistency* of models in their work.

To represent models in a more formal way the *Object Management Group* (OMG) developed the *Unified Modeling Language* (UML) [12] which by now advanced to an industry standard for modeling. UML provides an abstract syntax of the modeling concepts and rules how to combine them, an explanation of the semantics of the concepts as well as a specification of the notation elements in human-readable form. Build on these common concepts models can be preserved over time and even be reused due to their formalized semantics. Nevertheless models might have to be revised over the lifecycle of the software. With todays trend to more agile software development such as eXtreme Programming [3] or Scrum [15] changes on models have to be even more efficient which brings refactoring of models into the focus of research.

Refactoring is a technique that originates from source code development but can also be applied to model engineering. The goal is to introduce behavior preserving changes [14] that increase the quality and understandability of the models.

While refactoring source code and textual code respectively applies to a single type of representation, in UML different types of diagrams exist to represent various aspects of the models. This makes behavior preserving refactoring even harder as it needs to span over those different types of diagrams and semantics.

Different approaches exist to prove the semantic preservation of refactorings. One is the static analysis of UML models via the *Object Constraint Language* (OCL) [11] to assert different properties of the UML model, both before and after the refactoring. Another way is to verify that models can be executed and to compare different attributes of the execution like input and output values, execution traces or states of the original and refactored models. The OMG introduced a foundational subset of UML metamodel concepts abbreviated fUML [13] and precisely defined the semantics for their execution. Thanks to this standard compliant UML models can be transformed to an executable form. Furthermore in [8] Mayerhofer *et al.* proposed a framework based on fUML that is able to execute and debug models that conform to fUML.

The goal of this work is to introduce refactorings for fUML models, examine the requirements for co-refactoring of the corresponding diagram types and define which co-changes have to be performed to preserve the behavior. Our approach to verify the semantic preservation and the correctness of the models is twofold. On the one hand we use OCL pre- and postconditions [16] to determine whether a refactoring can be applied and whether the models are semantically correct afterwards. On the other hand the models have to stay executable after the changes, if they are executed with the same input data.

The rest of the work is structured as follows. Section 2 gives an introduction to fUML and shows how the preservation of semantics can be achieved during refactoring. Section 3 describes a selection of useful refactorings inspired by Fowler [5] and Markovic and Baar [7] and their effects on class diagrams as well as activity diagrams. We also provide a motivating example of a model that is used throughout the paper. In Section 4 we show which pre- and postcondition are needed for the refactorings and how we refactor the models. In Section 5 we describe the toolchain that we use to define the models, implement a set of refactorings and test them. Section 6 describes which limitations our work has. Related work is covered in Section 7 and a conclusion is drawn in Section 8 to summarize the paper.

2 Motivation

fUML is a subset of UML for which additional semantics are defined such that models can be executed. The subset contains concepts of the packages *Classes*, *CommonBehaviors*, *Activities* and *Actions*, which basically means that it is possible to create *class diagrams* and *activity diagrams* with fUML. At the same time it means that UML models which conform to fUML are also executable.

If a refactoring is performed on a UML model such as a class diagram, then any activity diagram which is releated to the class diagram, has to be checked and possibly changed as well. In Section 4 we will present some examples of fUML activity diagrams and present the implications that result from changing class diagrams (a procedure called *co-refactoring*).

Since a refactoring changes the structure of a model it is important to ensure that all changes maintain the original semantics of the model. Violating this requirement can result in models with either a different behavior, or in models which can no longer be executed. To ensure semantic preservation, two main techniques can be used. First, the refactoring can be broken down into smaller steps, each of which either guarantees to preserve the semantics of the model or makes it easier to verify that this is the case. Second, logical constraints can be used to limit refactorings on models to only those cases where semantic preservation can be ensured. For this purpose pre- and postconditions are specified with OCL constraints. A refactoring is then only applied if the original model satisfies the precondition before the refactoring is applied and the postcondition after the refactoring has been completed. Such constraints must be individually specified for each kind of refactoring that is to be performed. In this paper we introduce and discuss different OCL constraints for the refactorings that we introduce.

Refactorings are only useful if they can be easily applied to the models through an easy to use process such as a graphical front end. EMF Refactor[1] allows the integration of refactorings into editors such as the UML tree editor

[1] `http://www.eclipse.org/emf-refactor/`

or Papyrus[2]. EMF Refactor provides a Java API as well as a module concept to integrate refactorings. We will discuss the use of EMF Refactor as part of our tool chain presentation.

3 Refactoring examples

This section covers some refactorings of fUML models as well as a fUML model which we use as an example for this work. Markovic [7] presented a list of refactorings for class diagrams and co-refactoring of OCL constraints. However they do not consider other diagram types and possible co-refactorings between them. Nevertheless we use this catalog as an input and adapt it for our use case.

Class diagram refactoring	Abstract syntax changes	Activity diagram co-refactoring
Encapsulate property	Yes	Yes
Extract class	Yes	Yes
Extract superclass	Yes	No
Pull up association end	Yes	Yes
Pull up property	Yes	Yes
Pull up operation	Yes	Yes
Remove unused class	Yes	No
Rename class	No	No
Rename operation	No	No
Rename property	No	No

Fig. 1. Considered refactorings

The resulting list of refactorings which we plan to evaluate and implement is shown in Figure 1. The first column (*Class diagram refactoring*) displays the name of the refactoring as used in [7], but adapted to the concept names used in UML[3]. The second column (*Abstract syntax change*) indicates whether the abstract syntax changes due to the refactoring. Finally the third column (*Activity diagram co-refactoring*) shows whether we need to co-refactor the corresponding activity diagrams in parallel to the changes in the class diagram.

If we apply for example a rename refactoring the abstract syntax does not change because the concepts are linked by reference and the new name will automatically "propagated" to the other concepts. Thus there is no need to co-refactor the activity dagrams. An other example would be the *encapsulate property* refactoring which changes the abstract syntax as new operations are introduced and the visibility of the property is changed. This has also effects on the activity diagram where several action accessing the property might have to be changed. A more detailed explanation of co-refactorings is given in Section 4 together with the introduction of each refactoring.

[2] `http://www.eclipse.org/papyrus/`

[3] E.g. instead of *attribute* we use the *property* in the name of the refactoring

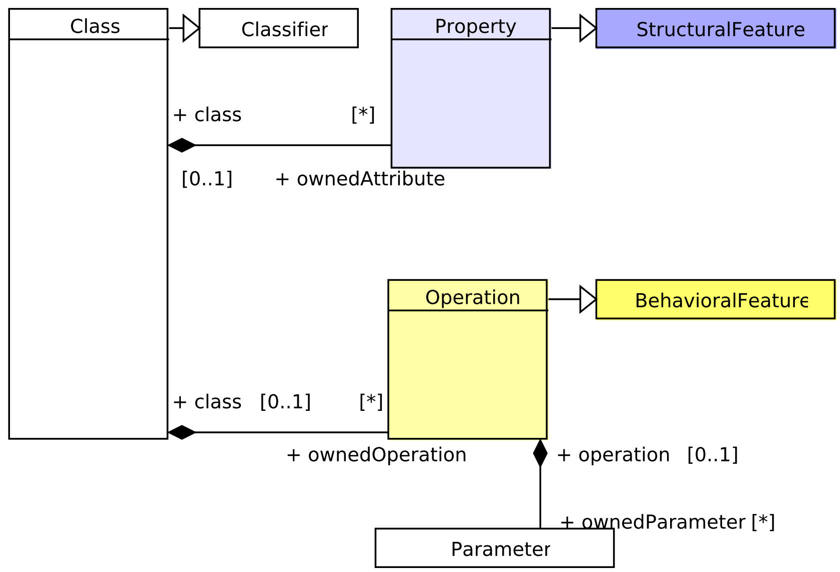

Fig. 2. Abstract syntax for used class concepts in *fUML*

In order to demonstrate how the presented refactorings affect both class and activity diagrams we will present an example model from the insurance domain. For a better understanding of the abstract syntax we first show the relevant parts of the fUML subset for our example. Figure 2 explain how the meta-model concepts `Property`, which is a subclass of `StructuralFeature`, and `Operation`, which is a `BehavioralFeature`, are linked to the `Class`. Furthermore is shows that `Operation` owns zero or more `Parameter`. Figure 3 shows the hierarchy of the most relevant actions. Important to mention is the `CallOperation-Action` which holds a single `Operation`, and the `StructuralFeatureAction`, which holds a single `StructuralFeature`.

Figure 4 shows our example class diagram of an insurance company with relevant properties and operations. In our case there are different domain objects. A company administers different customers and employs some employees. Further the company sells insurance policies where trucks and cars can be added and insured for a certain period. Customers can hold such insurance policies that are signed by an employee to verify the correctness.

This class diagram would benefit from several possible refactorings such as an *extract superclass* which can be applied to both `Car` and `Truck` to extract a `Vehicle` class. An *extract class* refactoring can be used on `InsurancePolicy` to extract the `from` and `until` dates into an own `InsurancePeriod` class. As part of the *extract superclass* refactoring two additional refactorings namely *pull up property* and *pull up operation* are used to move the identical properties and operations of both classes to the new superclass. As the properties `weight`

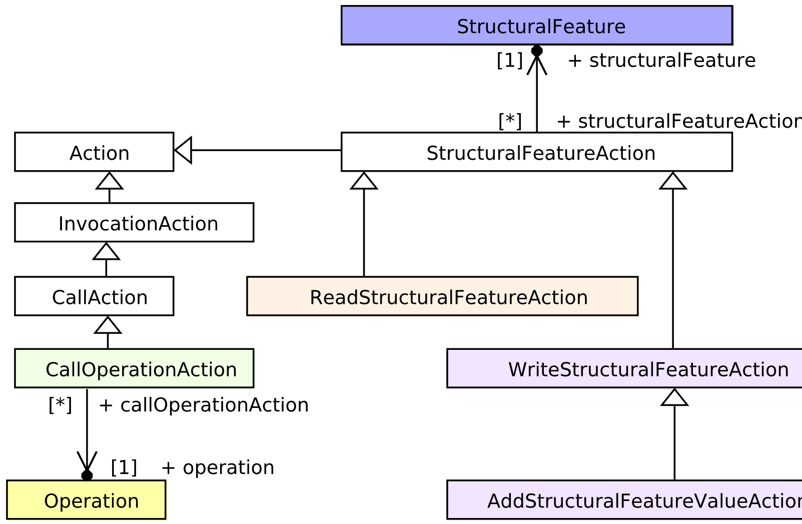

Fig. 3. Abstract syntax for used actions in *fUML*

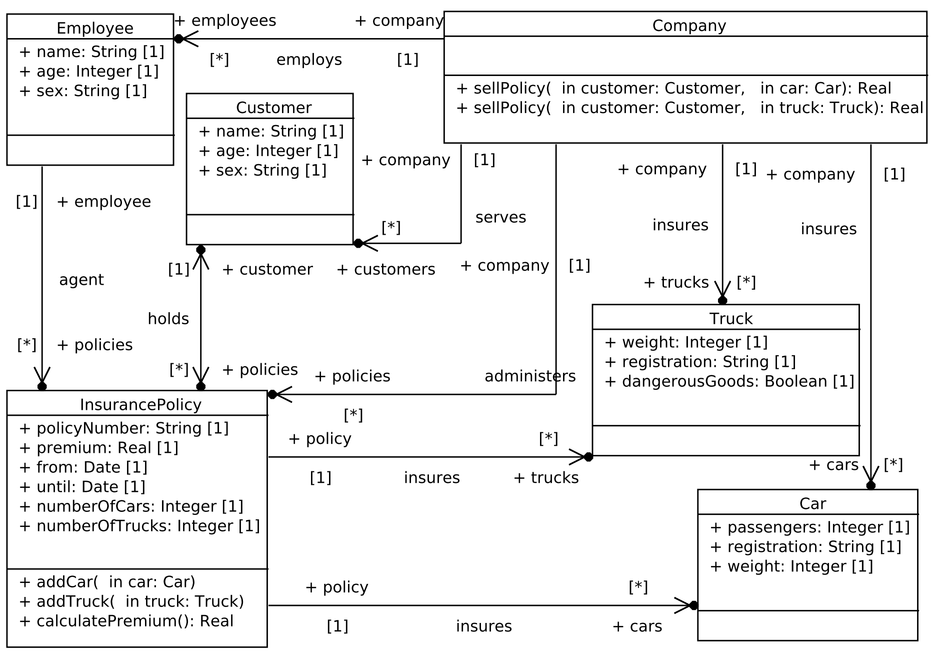

Fig. 4. Insurance class diagram before refactoring

and `registration` are public, we can use *encapsulate field* to set their visibility to private and provide getter and setter operations. Finally a new operation `addVehicle` can be introduced and the `addTruck` as well as the `addCar` operation can be removed with *remove operation*[4]. Figure 5 shows the class diagram resulting from the refactoring.

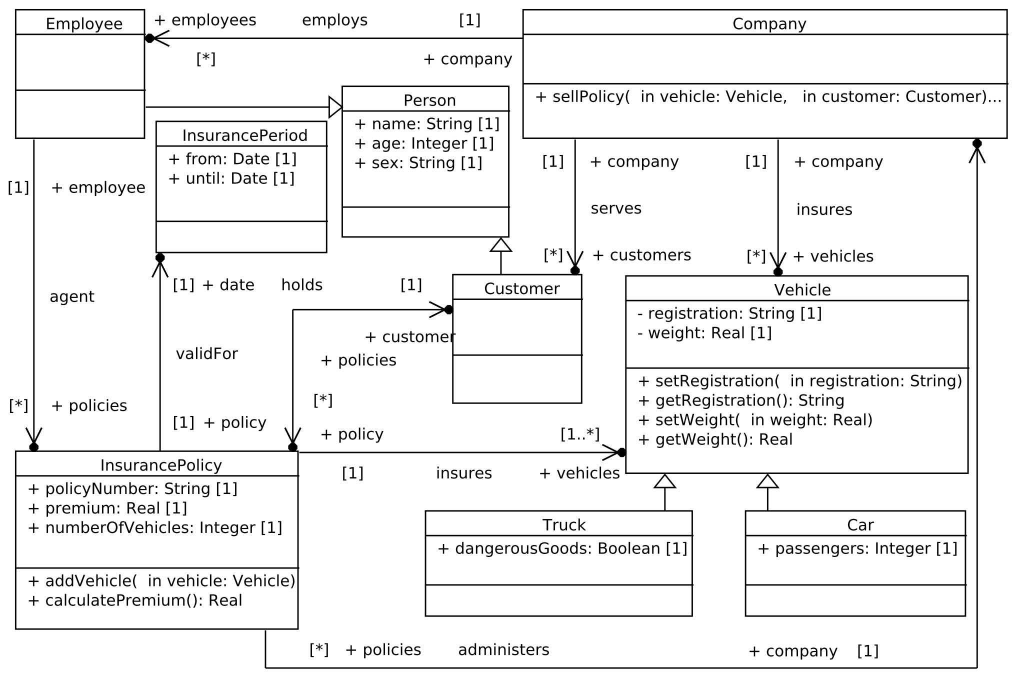

Fig. 5. Insurance class diagram after refactoring

For each of the operations in the `InsurancePolicy` class a separate activity diagram exists which defines the behavior of the respective operations. As some of them are quite similar we will present only the diagrams for `addCar` in Figure 6 and `calculatePremium` in Figure 7.

The activity for the `addCar` operation works as follows. It reads the policy with a `ReadSelfAction`, takes the car as a parameter and uses the `AddStructuralFeatureValueAction` to add the car to the insurance policy. In parallel it reads the number of cars in the policy with `ReadStructuralFeatureValueAction`, specifies an integer with a value of '1' with a `ValueSpecificationAction`, adds the numbers with a `CallBehaviorAction` and writes it back to

[4] *add operation* and *remove operation* are not covered by our paper

the `numberOfCars` variable with another `AddStructuralFeatureValueAction` that replaces the old value.

The `calculatePremium` activity is more complex, it calculates the insurance premium as described below:

– Read the policy with a `ReadSelfAction`.
– Read the number of cars and trucks with `ReadStructuralFeatureValueAction`.
– Add the two numbers up with a `CallBehaviorAction`, which invokes the opaque behavior 'add'.
– Multiply the result in a `CallBehaviorAction` with a base premium value specified in a `ValueSpecificationAction`.
– Read the age of customer of the policy with two actions of type `ReadStructuralFeatureValueAction` in a row.
– Decide whether the age of a customer is below a certain age and define a base value or supplement with a `ValueSpecificationAction`.
– Multiply the value and the result of the calculation above and return it.

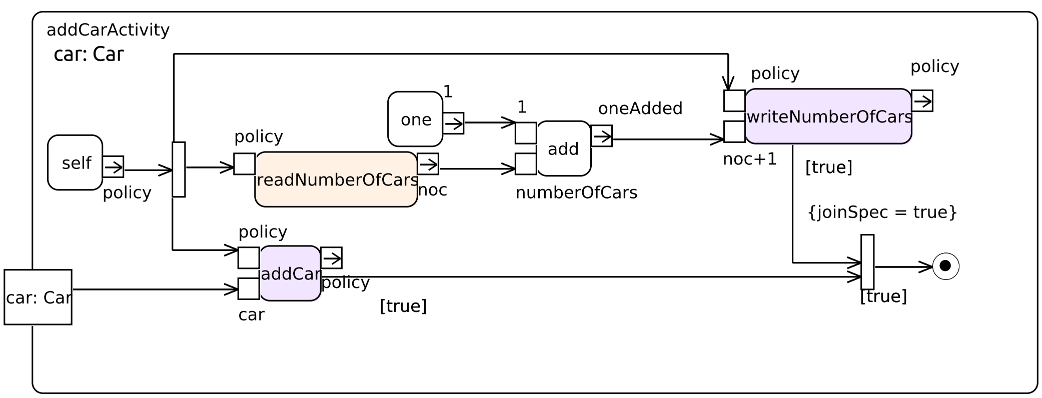

Fig. 6. Activity diagram of operation `addCar` before refactoring

Figure 8 shows how the activity diagram of the operation `calculatePremium` benefits from a refactoring of the corresponding class diagram. Additionally a new activity diagram for the `addVehicle` operation instead of the old `addCar` and `addTruck` diagrams was created shown in Figure 9. It is rather obvious that the complexity of both diagrams decreased on the visual level which helps interested parties to better understand the model.

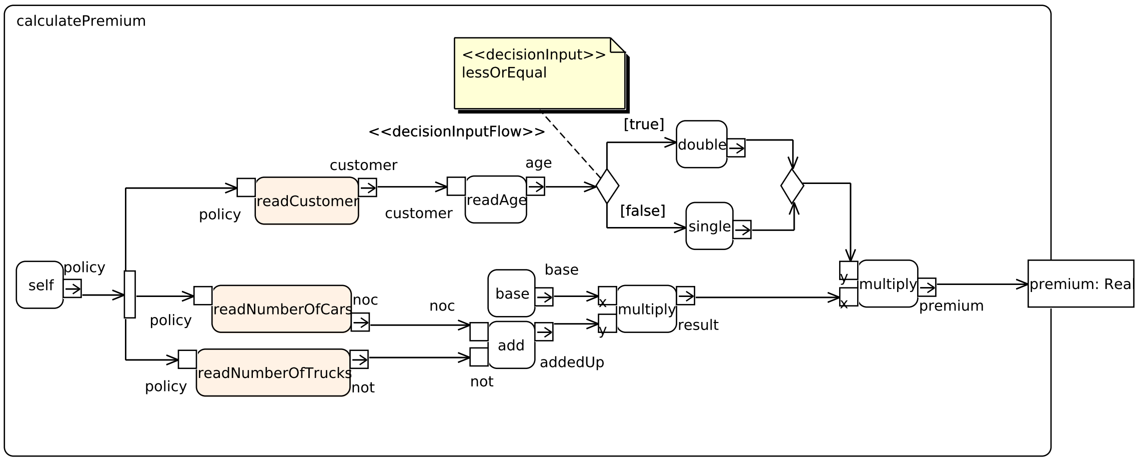

Fig. 7. Activity diagram of the operation `calculatePremium` before co-refactoring

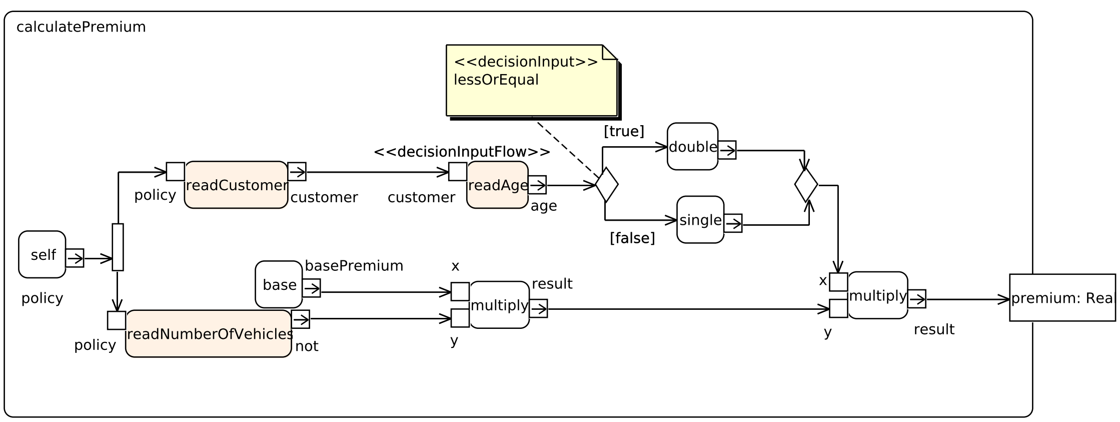

Fig. 8. Activity diagram of the operation `calculatePremium` after co-refactoring

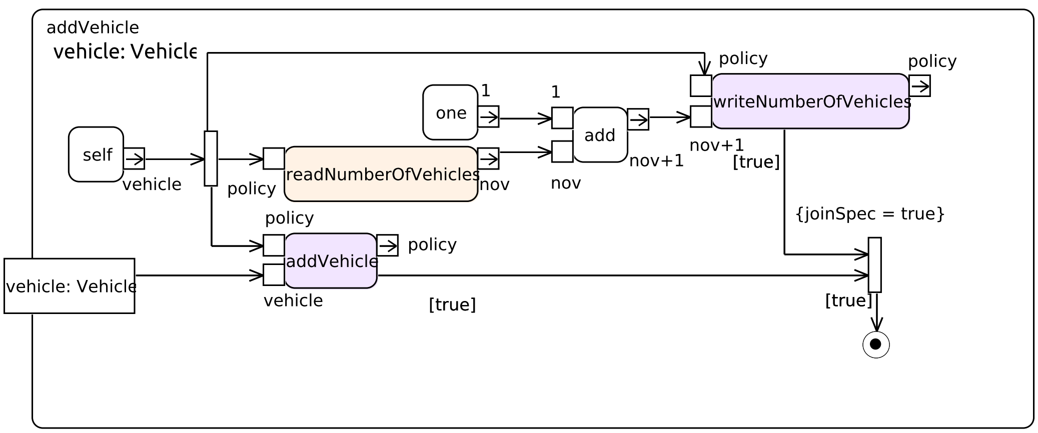

Fig. 9. Activity diagram of operation `addVehicle` resulting from the refactoring

4 Refactoring of fUML models

In the last section we presented an example fUML model comprising a class diagrams and several activity diagrams. We also presented some of the possible refactorings such as *extract superclass* on the class diagram and their impact on the activity diagrams. In this section we discuss how the syntax of the models actually need to be transformed in order to realize the refactoring. Furthermore we describe the refactorings from the prior section in detail.

Every model has two kinds of syntaxes. The concrete syntax (or notation), which defines how the model is visualized, and as described the abstract syntax, which defines the UML language elements and the grammar of the model. In order to refactor a model, we need to look at its abstract syntax and transform it according to the refactoring rules. Adopting the concrete syntax of co-refactored activity diagrams is out of the scope of this work.

4.1 Rename class, rename property & rename operation

A refactoring takes several parameters to configure the parts of the model that change. Rename refactorings need the model element that shall be refactored as well as the new name as parameter. As the abstract syntax is not changed the respective classes, operations and properties can then be adapted and the changes are automatically propagated. As refactoring is trivial in our example there is no such case explicitly included. However in order to guarantee that the refactoring is possible, we have to ensure that there are no occurrences of the same type with the same name as the new name. In this work we did not take techniques such as operation overloading into account.

Listing 1 shows the OCL constraint that has to be checked prior to the *rename property* refactoring. All properties of the class and its parents have to be check such that any property name does not match with the indicted new name.

```
context Property:
pre:  self.class.attribute.name->
        forAll(n | n <> 'newAttributeName')
      and
      self.class.inheritedMember->selectByType(Property).
        name->forAll(n | n <> 'newAttributeName')
```

Listing 1. OCL for *rename property* refactoring

Listing 2 shows the OCL precondition for the *rename operation* refactoring. As operations might have same name but different parameter declarations we use the OCL function `isDistinguishableFrom` on the united operations of the class and its parents. This precondition is not a pure OCL constraint as the operation `newOperation` is constructed in Java before and checked against the existing ones in the model. We will show an example of this in Section 5.

```
context Operation:
pre:  self.class.inheritedMember->selectByType(Operation)
        ->forAll(o | (o.parameterableElements() <>
          self.parameterableElements())
          and (o.name <> 'newOperationName'))
      and
      self.class.ownedOperation->excluding(self)
        ->forAll(o | (o.parameterableElements() <>
          self.parameterableElements()
          and (o.name <> 'newOperationName')))
```

Listing 2. OCL for *rename operation* refactoring

As the OCL constraint for the *rename class* refactoring is very similar to Listing 1 we put it into Appendix A under Listing 10.

As we already mentioned the changes are automatically propagated to the respective diagrams. Because of this we do not need to formulate postconditions for the rename refactorings.

4.2 Extract superclass

The *extract superclass* refactoring takes two parameters. The name of the superclass to extract as well as the class that participates in the refactoring and which will receive the created class as its superclass.

Figure 4.2 shows the changes of the refactoring on the abstrax syntax level. A new `Class` is introduced and linked via a `Generalization` to the already existing `Class`. In our example the two classes `Car` and `Truck` will be connected to the new superclass `Vehicle` via the attribute *superClass* of the meta-model concept `Class`. Listing 3 shows the OCL pre- and postcondition for this refactoring. The preconditions checks that there are no classes in the namespace with the same name. The postcondition whether the newly created class is visible and the existing class is a subclass. Parts of the model[5] might also benefit from changing references to the new class.

```
context Class:
pre:  self.namespace.member
        ->selectByType(Class).name
        ->forAll(o | o <> 'nameOfClassToExtract')
post: newSuperClass.visibility = uml::VisibilityKind::public
      and
      self.general->includes(newSuperClass)
```

Listing 3. OCL for *extract superclass* refactoring

[5] E.g. the insurance policy can hold a single variable referencing all vehicles

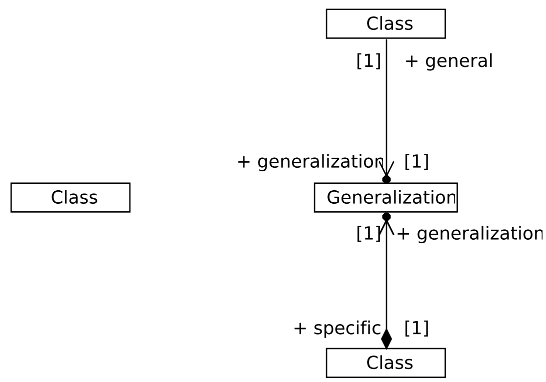

Fig. 10. Abstract syntax before & after *extract superclass* refactoring

4.3 Pull up property & pull up operation

Both refactorings are quite similar which is why we describe the first and explain the specifics for the latter one afterwards. There are two cases of the *pull up property* refactoring. It can either be executed for a property that occurs in a single class or in multiple classes where the properties should also be combined and pulled up. In the class diagram the property has to be pulled up and the occurrences of the properties in the subclasses have to be deleted. The activity diagrams needs to change occurrences of the `StructuralFeatureValueAction` and use the property of the super class.

For the *pull up operation* refactoring all `CallOperationAction` actions have to be adapted like described above. A precondition for both of these refactorings is that there does not already exist an property or a operation with the same name in the target class and that they are not private.

4.4 Encapsulate property

The *encapsulate property* refactoring is a rather complex one. The goal is to change the visibility of a property to *private* and introduce getter and setter operations. Thus at first setter and getter operations for the property are created and added to the class. For both operations the corresponding activities are also created. Afterwards each `StructuralFeatureAction` in the model (which accesses the property) must be replaced by a `CallOperationAction`. If the action is a `ReadStructuralFeatureAction` then the call operation invokes the getter, otherwise the setter. Finally the property's visibility is set to private.

Figure 11 shows the abstract syntax of a part of the model before the refactoring and Figure 12 shows the abstract syntax of the same model after the refactoring. In Figure 12 we have omitted the activity objects to make the diagram simpler. For each operation there is actually a corresponding `Activity` object, which is also contained by the `Class` object.

Creating the two activities for the getter and the setter is not trivial. It requires the creation of a `ReadSelfAction` to retrieve the properties owner class and the creation of the correct `StructuralFeatureAction`. Additionally object and control flow edges, input and output pins and activity parameter nodes need to be created and connected with the actions. For each of these elements there are several properties in the abstract syntax that have to be set correctly, such as the *type* property of pins or the *source* and *target* properties of edges. If only one element is not correctly generated by the refactoring, then the refactoring results in an incorrect model. Once the activities are added to the model, the refactoring searches for structural feature actions to replace. Besides creating the call operation and removing the structural feature action, care must be taken to reassign the corresponding input pins and control flow edges to the call operation action.

As UML does not explicitly distinguish between scalars and list types and the differences is only visible in the multiplicity of the property, the refactoring needs to take this into account. Our current implementation only handles scalars properties. The precondition for this refactoring is that the property is not already private and there are not already getters and setters for it. Additionally the multiplicity of the property must be '1' to prevent the refactoring for list types. We also reject the refactoring if `ClearStructuralFeatureAction`s or `RemoveStructuralFeatureValueAction`s are present for the property as these would also require additional consideration in the refactoring code. Listing 4 show the complete precondition.

The postcondition is more complex and thus we have split it into five separate conditions. We verify that the amount of input and output pins from the removed structural feature actions matches the number of respective pins of the added call operation actions. Additionally we require that the type of each pin is correctly set. The resulting postconditions can be found in appendix A.

```
context Property:
pre:  self.visibility <> uml::VisibilityKind::private and
      self.class.ownedOperation
      ->forAll(o | o.isDistinguishableFrom(setOperation,
               self.namespace) and
      o.isDistinguishableFrom(getOperation, self.namespace))
```

Listing 4. OCL for encapsulate property

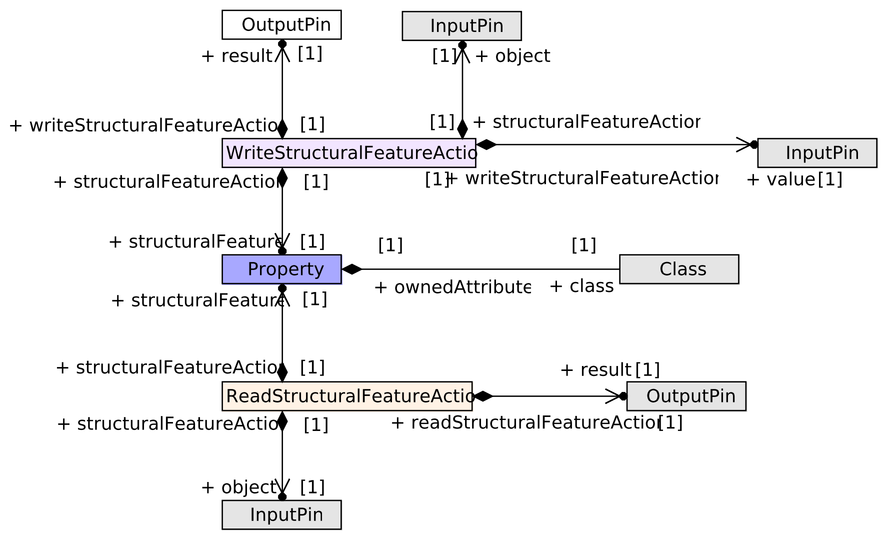

Fig. 11. Abstract syntax before *encapsulate property* refactoring

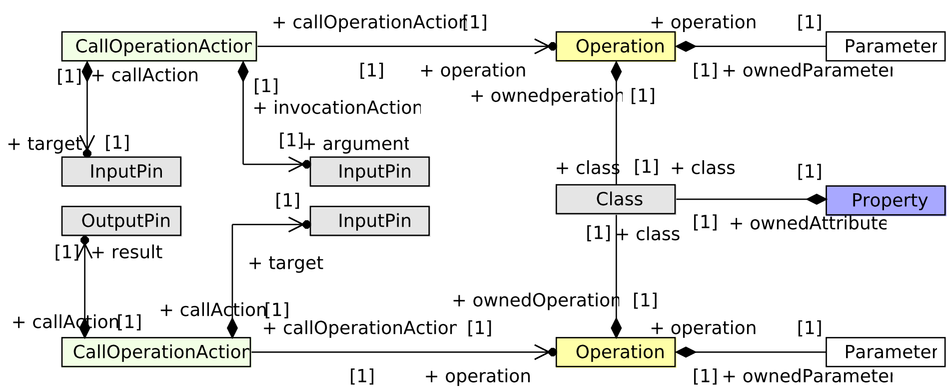

Fig. 12. Abstract syntax after *encapsulate property* refactoring

5 Toolchain and implementation

For our toolchain we have relied on the moliz[6] repository, mainly for the ability to execute the fUML models in a virtual machine. The models are stored in XMI format and loaded with an EMF ResourceSet. The ResourceSet can be used to retrieve a Resource object through an URI, which is then used to access the different objects in the model (see Listing 5). All elements contained in the model are instances of Ecore (the implementation of the MOF meta-meta-model) and they represent classes of the UML meta-model, which is implemented with Ecore.

```
resourceSet = new ResourceSetImpl();
resourceSet.getPackageRegistry().put(UMLPackage.eNS_URI,
                                     UMLPackage.eINSTANCE);
resourceSet.getResourceFactoryRegistry()
    .getExtensionToFactoryMap()
    .put(UMLResource.FILE_EXTENSION,
         UMLResource.Factory.INSTANCE);
File file = new File(umlModelFile);
URI uri = URI.createFileURI(file.getAbsolutePath());
    resource = resourceSet.getResource(uri, true);
```

Listing 5. Getting the resourceset

5.1 Model refactoring

Our refactorings are implemented with a strategy pattern, where each refactoring implements a `Refactorable` interface that has three operations `checkPrecondition`, `performRefactoring`, and `checkPostcondition`. Each refactoring identifies the model objects that participate in the refactoring and then creates and evaluates the *OCL* constraints on these objects. An object to type `OCL` from the org.eclipse.ocl package is used to create and evaluate the queries as can be seen in Listing 6.

[6] http://www.modelexecution.org

```java
variable = ExpressionsFactory.eINSTANCE.createVariable();
variable.setName("newSuperClass");
variable.setType(UMLPackage.Literals.CLASSIFIER);
ocl.getEnvironment().addElement(variable.getName(),
                                variable, true);
query = helper.createQuery(
    "self.general->includes(newSuperClass)");
eval = ocl.createQuery(query);
eval.getEvaluationEnvironment()
    .add("newSuperClass", superClass);
if (!eval.check(clazz))
    return false;
```

Listing 6. OCL validation in Java

We have used *OCL* notation to specify all constraints, but in the actual implementation of the refactorings it is also possible to verify some of these constraints directly on the model by checking properties of the abstract syntax and not use the *OCL* validation facilities. One example is the constraint that verifies that the new super class does not yet exist in the model.

If the preconditions are satisfied the actual refactoring is performed through a direct modification of the model object. A new `Class` instance with the name of the new superclass is created and added to the model.

```java
Class superClass = UMLFactory.eINSTANCE.createClass();
superClass.setName(newSuperClassName);
```

Listing 7. UML element creation

Then the existing classes of the model are filtered to select the ones that receive the new generalization, and the newly create superclass is added.

5.2 Model execution

In the prior section we discussed how the models are statically examined with *OCL* constraints if a refactoring is applicable. In this section we will show how the models can be tested dynamically by executing them. We use the reference implementation as described in [9] to execute the activity diagrams of our models.

```java
NamedElement namedElement = obtainFirstNamedElement();
IConverter converter = getConverter(namedElement);
IConversionResult conversionResult = converter
  .convert(namedElement);
Activity activity = conversionResult.getActivity(name);
```

Listing 8. Converting the UML diagram to fUML

Our models are formally created in *UML* and are converted to *fUML* with the `UML2fUML` converter provided by the implementation as shown in Listing 8. The resulting *fUML* diagram is executed in the virtual machine for model execution.

We test our models by examining if a complete execution trace is possible for the chosen activity. Listing 9 shows how the activity is executed step by step. The `EventListener` catches the event which is created for each activity step and builds up a trace. If a trace is created and the process of execution is not interrupted we consider the test as successful. Testing more than one trace as some kind of branching is not part of this work and may be included in future studies.

```
getExecutionContext().addEventListener(
  new ExecutionEventListener() {
  @Override
  public void notify(Event event) {
    System.out.println(event);
    if (event instanceof ActivityEntryEvent
      && executionID == -1) {
      executionID = ((ActivityEntryEvent) event)
          .getActivityExecutionID();
    }
    if (event instanceof SuspendEvent) {
      SuspendEvent suspendEvent = (SuspendEvent) event;
      getExecutionContext().resume(
          suspendEvent.getActivityExecutionID());
    }
  }
});
getExecutionContext().executeStepwise(activity, null,
  new ParameterValueList());
getExecutionContext().getTrace(executionID);
```

Listing 9. Executing the activity stepwise and getting the execution trace

In the process of refactoring before and after each model change the corresponding activities are executed to prove that the refactoring did not corrupt the model. Further testing can be implemented by going through the trace and verifying that every node of the trace has the expected values.

5.3 Eclipse Integration

In this section we discuss how we can integrate the defined refactorings to the Eclipse framework. For our further research we plan to implement some the refactorings in EMF Refactor. EMF Refactor is an Eclipse incubation project which focuses on static model analysis and refactoring.

EMF Refactor analyses a project for so called code smells and calculates common model metrics. Those two project quality indicators reflect in a very convenient way which parts of a model could be improved. From a report view different refactorings already implemented in EMF Refactor can be applied to change the model. The refactorings work in the UML tree editor as well as in several graphical editors based on EMF.

However EMF Refactor mainly supports the refactoring of class diagrams. The effects that can be seen in activity diagrams when refactoring class diagrams are error prone and seem to happen more or less by accident. Nevertheless we intend to implement our above mentioned catalog in EMF Refactor because of the good integration with the Eclipse framework, the easy generation of simple refactorings[7] and the various implementation possibilities[8].

6 Current limitations

Our work currently has several limitations. So far we have designed a set of related class and activity diagrams, which are the basis for our refactorings. We have discussed the *extract superclass* refactoring including its pre- and post conditions and shown how it can be refactored with our toolchain. Until the final version of this paper we will formulate the pre- and postconditions for some of the refactorings that have not yet been discussed. Furthermore we will implement several more refactorings from the list shown in Figure 1, such as the *extract class*, the *rename operation*, the *pullup operation*, the *pullup field* and *rename variable* refactorings.

When the paper was written the refactorings were not yet implemented in EMF Refactor. We plan to create the whole refactoring chain for *extractSuperclass* in EMF Refactor which includes extracting a super class and pulling up the properties and operations programmatically. We also take the co-refactoring of the activity diagrams into account. If this works as intended we plan to rewrite the creation wizard which allows only refactorings on a single class. The code should ideally be carried to the EMF Refactor repository after a quality review.

7 Related work

In this section we give an overview on the related work. Refactoring in a general way with preconditions was described by Opdyke [14] in his master thesis and stated more precisely by Roberts [16] who also introduced postconditions for refactorings. Fowler [5] generated an extensive yet simple to understand catalog of refactorings for Java and Ruby which can be be adapted to model refactorings.

Sunyé et. al. [17] described how several refactorings can be applied to *UML* diagrams and introduced OCL as a possiblity to specify pre- and postconditions. Gorp *et al.* [6] extends the discussion with the usage of *OCL* for additional analysis such as code smells of models.

Despite of the further discussion of model refactoring ([4], [2], [1]) most authors concentrate on static analysis and class diagram representations of models.

[7] The framework supports the generation of new smells, metrics and refactorings with a module wizard in Eclipse.

[8] EMF Refactor supports the creation of refactorings in *Java*, *OCL* and *Henshin* a transformation language.

Dynamic analysis of models by execution and debugging *fUML* models is discussed by Mayerhofer [8] and provides the basis for the approach discussed here. Mayerhofer *et al.* [9] furthermore introduce a runtime model and an implementation[9] that is capable to test the models and directly show impacts of refactorings.

Arendt and Taentzer [1] present a framework that is based on *Eclipse* and the *Eclipse Modelling Framework* which allows static model analysis and refactorings that are implemented in different languages (Java, OCL & Henshin) and can be directly extended in *Eclipse*.

8 Conclusion

Refactoring models is a rather difficult task and it requires a concise knowledge of the involved technologies, with the list of involved technologies being rather long. For one a reasonably well understanding of fUML is required, in particular how class diagrams and activity diagrams are constructed and how they are related. It is not enough to simply be able to draw both class and activity diagrams, as one also needs a good understanding of the meta-meta-model (MOF) and the fUML and UML meta-model implementations in Ecore in order to understand and manipulate the model on the level of its abstract syntax. Finally an understanding of EMF (in particular Ecore and OCL) is required. While MOF and OCL are both modeling concepts and languages respectively, EMF contains implementations for them in Java with a complex API. Understanding these APIs of Ecore and OCL is required to perform the refactorings and was a prerequisite to build our toolchain.

While it took a considerable amount of time to get familiar with all these technologies, we were eventually able to create a tool chain for model refactoring. With the toolchain in place another challenge was to identify the required steps to perform the actual refactoring work. In particular this meant to identify how the activity diagrams need to change if a change is made in the class diagram.

Most of our efforts in the development of this work were focused on gaining a comprehensive understanding of the technologies described above, to develop the tool chain and to draw the various diagrams presented in this paper and make them executable. We have presented a comprehensive set of diagrams as the basis for our refactoring work and demonstrated the feasibility of our tool chain. However a more comprehensive evaluation of the different refactorings including the specification of pre- and postconditions OCL in is still required and will be the focus of our ongoing efforts.

[9] http://www.modelexecution.org

References

1. Arendt, T., and Taentzer, G. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Autom. Softw. Eng. 20*, 2 (2013), 141–184.
2. Baar, T., and Markovic, S. A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. In *Ershov Memorial Conference* (2006), pp. 70–83.
3. Beck, K. Embracing Change with Extreme Programming. *IEEE Computer 32*, 10 (1999), 70–77.
4. Correa, A. L., and Werner, C. M. L. Applying Refactoring Techniques to UML/OCL Models. In *UML* (2004), pp. 173–187.
5. Fowler, M. *Refactoring - Improving the Design of Existing Code*. AddisonWesley, July 1999.
6. Gorp, P., Stenten, H., Mens, T., and Demeyer, S. Towards Automating Source-Consistent UML Refactorings. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 144–158.
7. Markovic, S., and Baar, T. Refactoring OCL annotated UML class diagrams. *Software and System Modeling 7*, 1 (2008), 25–47.
8. Mayerhofer, T. Testing and debugging UML models based on fUML. In *ICSE* (2012), pp. 1579–1582.
9. Mayerhofer, T., Langer, P., and Kappel, G. A runtime model for fUML. In *Models@run.time* (2012), pp. 53–58.
10. Mohagheghi, P., Dehlen, V., and Neple, T. Definitions and approaches to model quality in model-based software development - A review of literature. *Information & Software Technology 51*, 12 (2009), 1646–1669.
11. OMG. *OMG Object Constraint Language*, 2.3.1 ed. OMG, `http://www.omg.org/spec/OCL/2.3.1`, 01 2011.
12. OMG. *OMG Unified Modeling Language*, 2.4.1 ed. OMG, `http://www.omg.org/spec/UML/2.4.1/`, 05 2011.
13. OMG. *Semantics of a Foundational Subset for Executable UML Models*, 1.1 ed. OMG, `http://www.omg.org/spec/FUML/1.1`, 08 2013.
14. Opdyke, W. F. Refactoring object-oriented Frameworks. Master's thesis, University of Illinois, 1992.
15. Rising, L., and Janoff, N. S. The Scrum Software Development Process for Small Teams. *IEEE Software 17*, 4 (2000), 26–32.
16. Roberts, D. Practical analysis for refactoring. Master's thesis, University of Illinois, 1999.
17. Sunyé, G., Pollet, D., Traon, Y. L., and Jézéquel, J.-M. Refactoring UML Models. In *UML* (2001), pp. 134–148.

A Appendix

```
context Class:
pre:  self.namespace.member
      ->selectByType(Class)
      ->forAll(c | c.name <> 'newClassName')
```

Listing 10. OCL for *rename class* refactoring

```
context Property:
pre:  self.visibility <> uml::VisibilityKind::private
      and
      self.class.ownedOperation
        ->forAll(o | o.isDistinguishableFrom(setOperation,
            self.namespace)
          and
          o.isDistinguishableFrom(getOperation,
            self.namespace))
      and
      uml::ClearStructuralFeatureAction.allInstances().
        structuralFeature->forAll(s|s<>self)
      and
      uml::RemoveStructuralFeatureValueAction.allInstances().
        structuralFeature->forAll(s|s<>self)
      and
      self.upper <= 1"
post: uml::CallOperationAction.allInstances()
        ->select(action | action.operation = operation)
        ->collect(input)->select(pin | pin <> null)
        ->size() = inputPinCounter
      and
      uml::CallOperationAction.allInstances()
        ->select(action | action.operation = operation)
        ->collect(result)->select(pin | pin <> null)
        ->size() = outputPinCounter
      and
      uml::CallOperationAction.allInstances()
        ->select(action | action.operation = operation)
        ->collect(target)->forAll(target | target.type
        = self.class)
      and
      uml::CallOperationAction.allInstances()
        ->select(action | action.operation = operation)
        ->collect(argument)->forAll(argument | argument.type
        = self.type)
      and
      uml::CallOperationAction.allInstances()
        ->select(action | action.operation = operation)
        ->collect(result)->forAll(result | result.type
        = self.type)
```

Listing 11. OCL for *encapsulate property* refactoring