fUML Refactoring with EMF

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 models. The main contribution of this work is the extension of existing UML refactorings to cover not only the static aspect of UML such as class diagrams but to also include refactorings for dynamic parts such as activity diagrams. In this work we will present basic concepts for refactoring of models with EMF and show how model semantics can be preserved through the use of OCL constraints. In the remainder of the paper we then present our toolchain and the used technologies of EMF (in particular Ecore and OCL) and how we used them for refactoring of 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 Papyrus.

Table of Contents

1 Introduction

Model-based software development or model-driven software development is not only an extensive field of research but receives also more and more 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. [11] discussed possible quality attributes of models in their work.

To represent formal models the OMG developed UML [13] which by now advanced to an industry standard for modeling. With these common semantics enriched models can be preserved over time and even reused. 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 [4] or Scrum [16] 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 [15] 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.

To prove the correctness of models before and after refactoring different approaches exist. One is the static analysis of UML models via metrics and the attempt to find model smells [2] which can be done via OCL [12]. Another way is to verify that models can be executed and to compare the execution traces of the original and refactored models. The OMG introduced a subset of UML named fUML [14] which precisely defines semantics for class and activity diagrams to make them executable. Furthermore in [9] Mayerhofer proposed a framework based on fUML that is able to execute and debug those models.

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 correctness of the models after refactoring is twofold. On the one hand we use pre- and postconditions [17] to define if the specific changes can be applied and that the models are semantically correct afterwards. On the other hand the models have to stay executable as well as traceable after the changes.

The rest of the work is structured as follows. In section 3 we give an overview over fUML and provide a motivating example of a model that is used throughout the paper. Section Y describes a selection of useful refactorings inspired by Fowler [6] and Markovic and Baar [8] and their effects on class diagrams as well as activity diagrams. 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. Related work is covered in section 7 and a conclusion is drawn in section 8 to summarize the paper.

2 Motivation

fUML is an extension of UML which builds on a subset of UML classes with the purpose of adding semantics to UML models such that they can be executed on the model level. The dominant concept for this is the activity diagram. 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 will 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 allows the integration of refactorings into editors such as the UML tree editor or papyrus. It 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 UML models as well as a model which we use as an example for this work. Markovic [8] presented a list of refactorings which is relevant for changes in combination with class diagrams and OCL constraints. We use this catalog as an input and adapt it for our use case. The resulting list of refactorings which we plan to evaluate and implement is shown in Figure 1.

As the basis for these refactorings an example from the insurance domain is used. In the insurance business there are domain objects such as an insurance

	abstract syntax changes
Encapsulate Field	Yes
Extract class	Yes
Extract superclass	Yes
Pull up association end	Yes
Pull up attribute	Yes
Pull up operation	Yes
Remove unused class	No
Rename Class	No
Rename Method	No
Rename Variable	No

Fig. 1. Refactoring examples

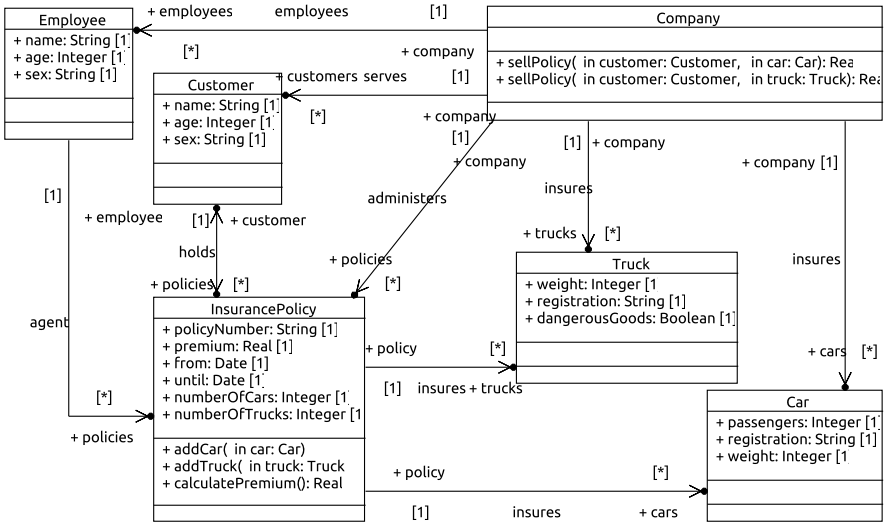

Fig. 2. Insurance class diagram without refactorings

policy. Cars and trucks can be insured by adding them to the policy. There is an insurance company which has customers and employees where customers may purchase an insurance policy for their cars or trucks. Figure 2 shows a class diagram of such an insurance company. 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. A simple *extract class* can be used on `InsurancePolicy` to extract the `from` and `until` dates into their own `InsurancePeriod` class. As part of the *extract superclass* refactoring two additional refactorings namely *pull up attribute* and *pull up method* are used to move the identical attributes and methods of both classes to the new superclass. As the attributes `weight` and `registration` are public, we can use *encapsulate field*

to make them private and provide getter and setter implementations. Finally the method `addCar` can be renamed into `addVehicle` with *rename method* and the `addTruck` method can be removed with *remove method*. Finally figure 3 shows the refactored class diagram.

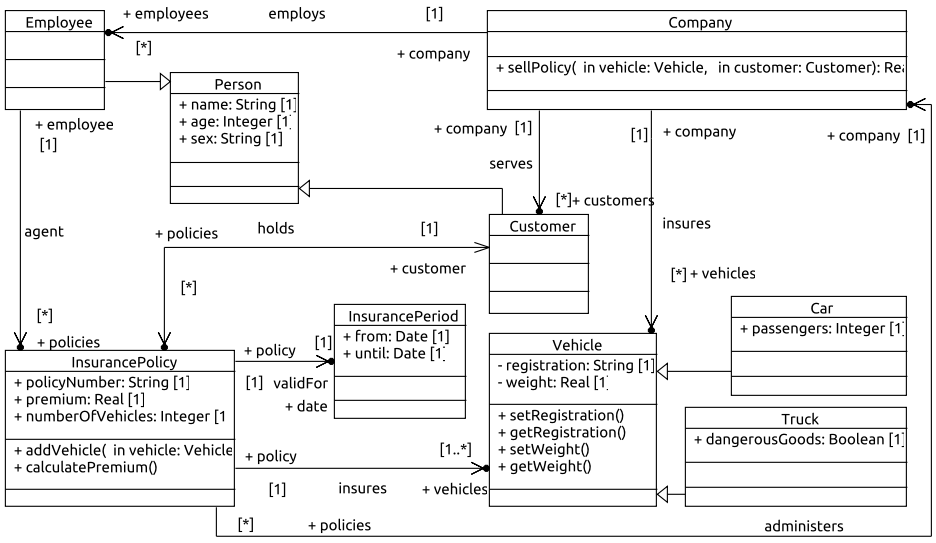

Fig. 3. Insurance class diagram with refactorings

For each of the methods in the `InsurancePolicy` class a separate activity diagram exists which defines the semantics of these methods. As some of them are quite similar we will present only the diagrams for `addCar` in figure 5 and `calculatePremium` in figure 4.

The activity for the `addCar` operation works as follows. Read the policy with a `ReadSelfAction`, take the car as a parameter and use 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 the value of one with a `ValueSpecificationAction`, adds the numbers with a `CallBehaviorAction` and writes it back to 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`.
- 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 or a base value or supplement with a `ValueSpecificationAction`.
– Multiply the value and the result of the calculation above and return it.

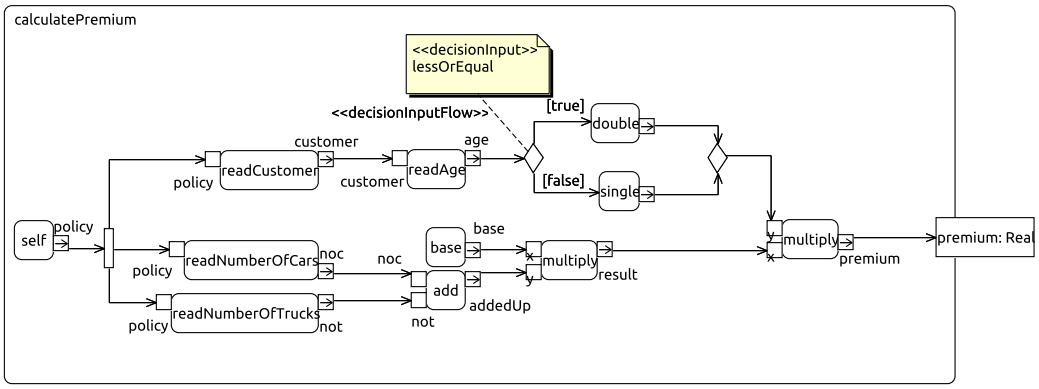

Fig. 4. Calculate premium activity diagram without refactorings

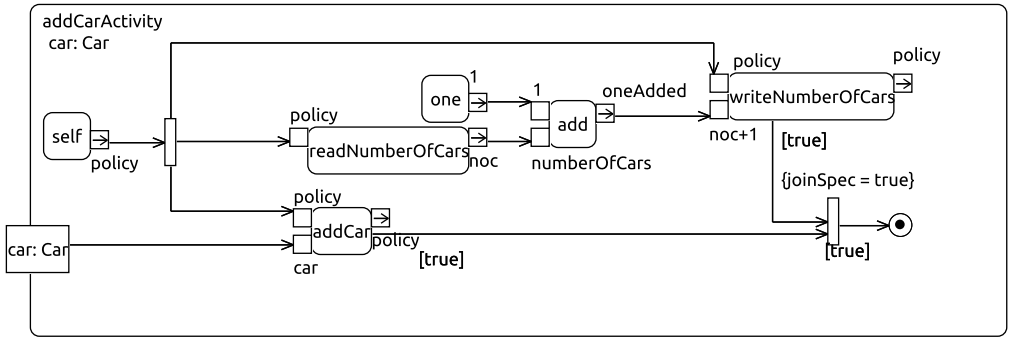

Fig. 5. Add car activity diagram without refactorings

Figure 6 shows how the activity diagram benefits from a refactoring of the corresponding class diagram. Additionally a new activity diagram for the

`addVehicle` method instead of the old `addCar` and `addTruck` diagrams was created shown in 7. It is rather obvious that the complexity of both diagrams decreased on the visual level as well as on the level of execution.

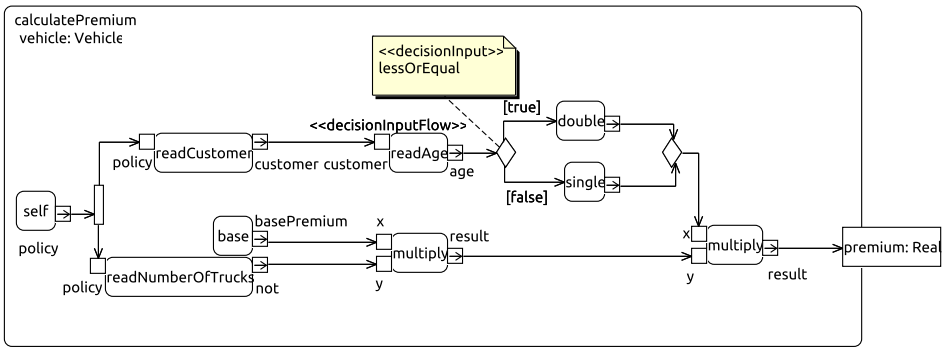

Fig. 6. Calculate premium activity diagram with refactorings

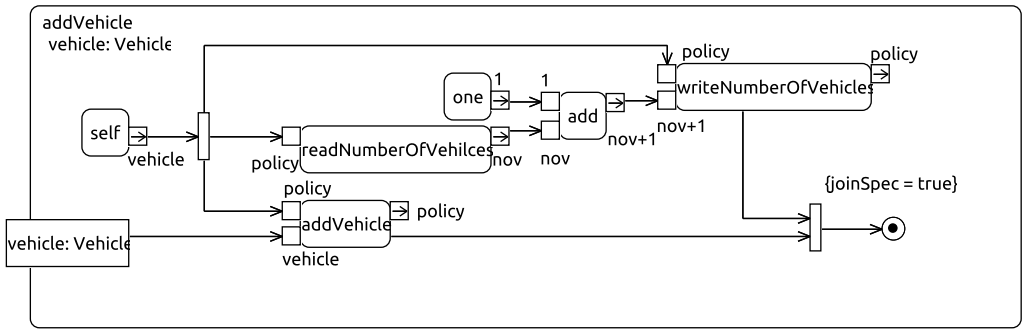

Fig. 7. Add vehicle activity diagram resulting from the refactorings

4 Refactoring of fUML models

In the last section we presented an example model and its related 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. This

section discusses how the models actually need to be transformed in order to realize the refactoring.

Every model has two kinds of syntaxes. The concrete syntax, which defines how the model looks like, and the abstract syntax, which defines the 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. In Figure 1 the refactorings are classified in order if the refactoring also implies changes in the meta-model representation. In the sections below we describe some examples that range from small to large impacts in the models.

4.1 Rename refacorings

A refactoring takes several parameters to configure the parts of the model that change and how they change. Rename refacorings just need the new name as parameter. The respective classes, methods and attributes can then be adapted and the direct references be changed as the abstract syntax representation does not change. 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 method overloading into account. In our example there is no such case explicitly included.

OCL for rename attribute:

```
pre: self.allParents().attribute
     ->union(self.attribute)
     ->forAll(a | a.name <> 'newAttributeName')
post: self.attribute->exists(a | a.name = 'passengers')
activity-diagram-post:
      self.structuralFeature.name = 'newAttributeName'
```

OCL for rename operation:

```
pre: self.getAllOperations()
     ->forAll(operation | operation.name
               <> 'newOperationName')
post: self.ownedOperation
      ->exists(op  | op.name = 'newOperationName')
activity-diagram-post:
     self.name = 'newOperationName'
```

OCL for rename class:

```
pre: self.owner.ownedElement
     ->select(o | o.oclIsTypeOf(Class))
     .oclAsType(Class).name
     ->forAll(name | name <> 'newSuperClass')
post: self.owner.ownedElement
     ->select(o | o.oclIsTypeOf(Class))
     .oclAsType(Class).name
     ->exists(name | name = 'newSuperClass')
```

The post conditions for rename attribute apply to all `StructuralFeatureAction` objects which reference the renamed attribute. For rename operation it applies to all `CallOperationAction` objects which call the renamed operation. For rename class all elements which reference objects of the renamed class must reference the new class name, that includes objects such as `Parameter` or `Pins` (InputPin, OutputPin).

4.2 Extract superclass

The *extract superclass* refactoring takes two parameters, the new name of the extracted superclass as well as the list of classes that participate in the refactoring and which will receive the new class as their superclass.

In terms of the abstract syntax of these models, the two classes `Car` and `Truck` are instances of type *Class* of the meta-model a superclass named `Vehicle` would be an instance of type *Class* as well, and can be associated with these two classes through the recursive attribute *general* of the class *Class* to create the inheritance hierarchy. However the abstract syntax representation of the diagram does not change. In order to achieve a correct refactoring each class that participates in the refactoring must not already have a superclass. The model would also benefit from changing all possible references to the new class.

OCL for extract superclass:

```
Single Inheritance:
    pre: self.superClass->isEmpty() and
         self.owner.ownedElement
            ->select(class|class.oclIsTypeOf(Class))
            .oclAsType(Class).name
            ->forAll(o | o<>newSuperClass.name)
    post: self.superClass->size() = 1

Multiple Inheritance:
   pre: self.owner.ownedElement
            ->select(class|class.oclIsTypeOf(Class))
            .oclAsType(Class).name
            ->forAll(o | o<>newSuperClass.name)
   post: self.general->contains(superClass)
```

4.3 Pull up attribute & pull up method

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 attribute* refactoring. It can either be executed for an attribute that occurs in a single class or in multiple classes where the attributes should also be combined and pulled up. In the class diagram the attribute has to be pulled up and the occurrences of the attributes in the subclasses have to be deleted. The activity diagrams needs to change occurrences of the `StructuralFeatureValueAction` and use the attribute of the super class.

For the *pull up method* 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 attribute or a method with the same name in the target class and that they are not private.

4.4 Encapsulate attribute

The *encapsulate attribute* refactoring is a rather complex one. At first setter and getter methods for the attribute have to be created. Afterwards the occurrences of the attribute have to be changed either to the setter for value assignments and to the getter for value usage. For the activity diagram the `WriteStructuralFeatureValueAction` has to be changed to a `CallOperationAction` which is a `BehavioralFeature` class instead of a `StructuralFeature` class in the meta-model. Finally the affected attribute has to be set to private. The precondition for this refactoring is that there are no methods with the same name in the classes that use the attribute.

5 Toolchain and implementation

For our toolchain we have relied on the moliz[1] 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 figure 8). 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.

```
1  resourceSet = new ResourceSetImpl();
2  resourceSet.getPackageRegistry().put(UMLPackage.eNS_URI,
3                                       UMLPackage.eINSTANCE);
4  resourceSet.getResourceFactoryRegistry()
5      .getExtensionToFactoryMap()
6      .put(UMLResource.FILE_EXTENSION,
7           UMLResource.Factory.INSTANCE);
8  File file = new File(umlModelFile);
9  URI uri = URI.createFileURI(file.getAbsolutePath());
10     resource = resourceSet.getResource(uri, true);
```

Fig. 8. Loading a UML model into a resource set

.

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

5.1 Model refactoring

Our refactorings are implemented with a strategy pattern, where each refactoring implements a `Refactorable` interface that has three methods `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 figure 9.

```
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;
```

Fig. 9. 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. 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 refacoring 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 [10] to execute the activity diagrams of our models.

Our models are formally created in *UML* and are converted to *fUML* with the `UML2fUML` converter provided by the implementation as shown in Figure 10. The resulting *fUML* diagram is executed in the virtual machine for model execution.

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

Fig. 10. Converting the UML diagram to fUML

We test our models by examining if a complete execution trace is possible for the chosen activity. Figure 11 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);
```

Fig. 11. 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 GUI 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 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 could be improved in a model. 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[2] and the various implementation possibilities[3].

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 method*, the *pullup method*, 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 attributes and methods 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 [15] in his master thesis and

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

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

stated more precisely by Roberts [17] who also introduced postconditions for refactorings. Fowler [6] 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. [18] described how several refactorings can be applied to *UML* diagrams and introduced OCL as a possiblity to specify pre- and postconditions. Gorp et al. [7] 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 ([5], [3], [1]) most authors concentrate on static analysis and class diagram representations of models. Dynamic analysis of models by execution and debugging *fUML* models is discussed by Mayerhofer [9] and provides the basis for the approach discussed here. Mayerhofer et al. [10] furthermore introduce a runtime model and an implementation[4] 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.

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

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

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.

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. Arendt, T., Taentzer, G., and Weber, A. Quality assurance of textual models within eclipse using ocl and model transformations. In *OCL@MoDELS* (2013), pp. 1–12.
3. 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.
4. Beck, K. Embracing change with extreme programming. *IEEE Computer 32*, 10 (1999), 70–77.
5. Correa, A. L., and Werner, C. M. L. Applying refactoring techniques to uml/ocl models. In *UML* (2004), pp. 173–187.
6. Fowler, M. *Refactoring - Improving the Design of Existing Code*. AddisonWesley, July 1999.
7. 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.
8. Markovic, S., and Baar, T. Refactoring ocl annotated uml class diagrams. *Software and System Modeling 7*, 1 (2008), 25–47.
9. Mayerhofer, T. Testing and debugging uml models based on fuml. In *ICSE* (2012), pp. 1579–1582.
10. Mayerhofer, T., Langer, P., and Kappel, G. A runtime model for fuml. In *Models@run.time* (2012), pp. 53–58.
11. 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.
12. OMG. *OMG Object Constraint Language*, 2.3.1 ed. OMG, http://www.omg.org/spec/OCL/2.3.1, 01 2011.
13. OMG. *OMG Unified Modeling Language*, 2.4.1 ed. OMG, http://www.omg.org/spec/UML/2.4.1/, 05 2011.
14. OMG. *Semantics of a Foundational Subset for Executable UML Models*, 1.1 ed. OMG, http://www.omg.org/spec/FUML/1.1, 08 2013.
15. Opdyke, W. F. Refactoring object-oriented frameworks. Master's thesis, University of Illinois, 1992.
16. Rising, L., and Janoff, N. S. The scrum software development process for small teams. *IEEE Software 17*, 4 (2000), 26–32.
17. Roberts, D. Practical analysis for refactoring. Master's thesis, University of Illinois, 1999.
18. Sunyé, G., Pollet, D., Traon, Y. L., and Jézéquel, J.-M. Refactoring uml models. In *UML* (2001), pp. 134–148.