Fifth Workshop on Refactoring Tools 2012 (WRT '12), June 1, 2012, Rapperswil, Switzerland.

An Approach and Tool for Synchronous Refactoring of UML Diagrams and Models Using Model-to-Model Transformations

Hafsteinn Þór Einarsson Helmut Neukirchen
University of Iceland, Reykjavík, Iceland
hafsteinn.thor.einarsson@gmail.com helmut@hi.is

Abstract

When refactorings are applied to software models that are specified using the *Unified Modeling Language* (UML), the actual model and the graphical presentation of the model using a diagram need to be distinguished. While UML refactoring tools exist, they typically perform transformations only on the model level and are not able to transform the corresponding diagram as well. Thus, the UML model and the diagram representation of the model may get out of sync. This paper presents an approach that can be used in UML tools to refactor UML models together with their diagrams. To this aim model-to-model transformations are applied to the underlying model as well as to the related diagram. To prove the applicability of this approach, a prototype plug-in for the Eclipse-based *Papyrus* UML editor has been implemented. The model transformation language *Query/View/-Transformation* (QVT) is used to specify the transformation of the UML model and the diagrams.

Categories and Subject Descriptors D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance and Enhancement—Restructuring, reverse engineering, and reengineering

Keywords Refactoring, UML, Diagrams, Models, QVT

1. Introduction

While refactoring [8] is mainly applied to improve the internal structure of source code, refactoring is also beneficial to restructure other artifacts, for example, software models represented using the *Unified Modeling Language* (UML) [14, 15]. *Model-Driven Engineering* (MDE) focuses on using high-level models instead of low-level implementation languages. When in the non-MDE approaches, refactorings are performed at the implementation language level, they need to be applied at the model level in MDE approaches. Thus, refactoring tool support for UML models in MDE is as worthwhile as for source code in programming.

When representing software models using UML, the actual *model* and the graphical presentation of a model in form of a UML *diagram* need to be distinguished: the model is completely independent from any visualisation and does therefore not contain any diagram layout information. Different (or even the same) aspects or parts of a model may be visualised by different diagrams and there may be model elements that are not visualised by any of the diagrams, but only contained in the model.

A couple of UML tools support refactorings on UML models. However, these tools do typically modify only the model level, not the existing diagrams that relate to the model that is refactored. Thus, the model and the corresponding diagrams may get out of sync. Furthermore, this prevents refactoring of the layout of the diagrams which may as well be an important target of refactoring.

This paper presents an approach to refactor diagrams and models, thus enabling to keep models and diagrams in sync and to refactor also graphical aspects of the diagrams. *Model-to-Model* (M2M) transformations are used thus avoiding transformations that are implemented using a low-level programming language. The feasibility of this approach is evaluated by implementing a prototype refactoring plug-in for the Eclipse-based *Papyrus* UML editor which is part of the open-source *Eclipse Modeling Project* [6].

This paper is structured as follows: subsequent to this introduction, foundations are presented. After that, the approach underlying the presented work is described in Section 3. Based on this, Section 4 provides an overview on the implemented prototype and preliminary results of applying it. Then, in Section 5, related work is discussed. Finally, a summary and an outlook conclude this paper.

2. Foundations

In the following, we provide an overview on UML and the *Eclipse Modeling Project* that is used for implementing our UML refactoring approach.

2.1 The MOF Family

The UML is a part of a bigger family of specifications that have been published by the *Object Management Group* (OMG). All these specifications are related via the *Meta Object Facility* (MOF) [12]. The specifications that are relevant for refactoring UML are mainly those about UML itself, *Object Constraint Language* (OCL), interchange formats, and model transformation. These are covered in the following.

2.1.1 The Unified Modeling Language UML

The *Unified Modeling Language* (UML) has been developed to represent software models. UML provides a visualisation of certain aspects of the model using different diagrams.

Ordinary programming languages are typically defined using a *Backus-Naur Form* (BNF) for the syntax. Such a programming language can then be used to write a program that might get refactored. Comparably, the abstract syntax of the *Unified Modeling Language* (UML) is defined in terms of the MOF [14]. MOF serves as an infrastructure to describe concepts (the *metamodel*) of the UML, for example allowed model elements and relationships between them. However, MOF does not convey any information about concrete syntax such as layout of graphical elements, therefore the notation of UML model elements and element relationships using diagrams is not defined using MOF but using prose language and graphical examples [15].

UML itself can be used to define an actual model (subsequently called *UML model* or just *model*) and to represent different views on a model using different *diagrams* (subsequently called *UML diagram* or just *diagram*). Both, the model and the diagram might be subject of refactoring: for example, the elements of a UML model and their relations might get restructured or just the graphical representation may get rearranged.

2.1.2 The Object Constraint Language OCL

The *Object Constraint Language* (OCL) [17] is typically used to specify application-specific constraints in UML models or to specify queries on UML models. But OCL expressions can also used in the QVT transformation languages (cf. Section 2.1.4). The abstract syntax of OCL is defined in terms of a MOF-compliant metamodel, a concrete textual syntax is defined using BNF.

2.1.3 Interchange Formats

OMG's *XML Metadata Interchange* (XMI) [16] is an *Extensible Markup Language* (XML)-based format that is typically used to exchange UML models, but in fact XMI is applicable to all models that have a MOF-compliant metamodel such as OCL. Many UML tools use the XMI format and therefore allow to exchange UML models. For example, a code generator tool reads an XMI file containing a UML model that has been created using a UML editor tool.

As the UML specifications [14, 15] define only the UML model in terms of MOF, the XMI format does not contain any layout information needed to display diagrams that visualise the model contained in an XMI description. Therefore, a UML editor tool that reads an XMI file can only import the contained model, but not any diagrams that might have been created as well. UML editors typically store the diagram-related information in some proprietary format. To overcome this, the *UML Diagram Interchange* (UMLDI) format [11] has been developed by OMG. However, the UMLDI format is in fact not supported by every tool.

Currently, UMLDI is in the process of being replaced by a specification that will be called *Diagram Definition* (DD). Both, UMLDI and DD use a MOF-based metamodel, this means the diagrams themselves are also treated as some kind of model and can thus be encoded in XMI format as well.

2.1.4 Model Transformation

As part of *Model-Driven Engineering* (MDE), model transformation approaches have been developed, for example, to generate source code from models. In addition to transforming from one abstraction level to another, *Model-to-Model* (M2M) transformations can be used to transform a model while staying at the same abstraction level or metamodel.

The OMG has developed *Query/View/Transformation* (QVT) [13] as a standard for transforming any MOF-compliant model into another MOF-compliant model. The QVT specification introduces three M2M transformation languages that are related to each other: The QVT *Core* supports mainly pattern matching and is defined using minimal extensions to OCL and MOF itself. Correspondingly, it has only an abstract syntax. QVT *Relations* is a more user-friendly declarative language that can be translated into QVT *Core* and has furthermore a concrete, textual syntax. QVT *Operational mappings* is based on QVT *Relations*, but provides a more procedural style and a concrete textual syntax that looks familiar to imperative programmers.

In addition to the built-in constructs, QVT allows to implement functions as a *black-box*. This allows calling functions that are implemented outside the scope of QVT, for example using an implementation language such as Java.

2.2 Eclipse Modeling Project

The *Eclipse Modeling Project* [6] is part of the open-source *Eclipse* platform. Those sub-projects or frameworks that are most relevant for implementing UML refactoring are presented in the following.

2.2.1 Eclipse Modeling Framework EMF

The *Eclipse Modeling Framework* (EMF) is the foundation of all other sub-projects and frameworks considered in this paper. It supports to use and manage models that have an *Ecore* compliant metamodel and to load or save it in XMI format. EMF's *Ecore* closely resembles OMG's *Essential MOF* (EMOF), a subset of basic MOF features — the *Complete MOF* (CMOF) can be built on top of EMOF.

2.2.2 Graphical Modeling Project GMP

The *Graphical Modeling Project* (GMP) was formerly called *GMF* because it combines EMF and GEF, the *Graphical Editing Framework*. GEF allows to create any kind of graphical editors. Being based on EMF and GEF, GMP supports to create editors for models that have an *Ecore* metamodel. The *GMF Notation* sub-project uses EMF as well to represent the diagram of a model as an *Ecore* model. To this aim, principles of OMG's UMLDI are applied.

2.2.3 Model Development Tools MDT

The *Model Development Tools* (MDT) provide EMF-based implementations of some industry standard metamodels. In addition, MDT contains tools, such as editors for working with the metamodel implementations.

The *UML2* sub-project contains an implementation of the UML 2.x metamodel. An implementation of OCL can be found in the *Eclipse OCL* sub-project.

The *Papyrus* sub-project provides an editor that, amongst others, supports to edit UML. As it is UML2- and GMP-based, both, the UML model and the UML diagram itself are represented using EMF. The UML model is stored in an XMI file having the extension "`.uml`" and the diagram model is stored in an UMLDI-compatible XMI file having the extension "`.notation`". A third file having the extension "`.di`" links together the model and the diagrams that refer to that model (even that file is in XMI format). The diagram model contains only the additional layout information that is not contained in the UML model — it does not contain any information that is already contained in the UML model: instead of duplicating the name of, for example, an identifier used in the UML model, the diagram model contains a reference into the UML model. When the diagram needs to be displayed, the text of the identifier is retrieved from the UML model. To enable this referencing, each model element gets a *Globally Unique Identifier* (GUID) assigned.

2.2.4 Model to Model Transformation M2M

The *Model to Model Transformation* (M2M) project provides infrastructure for M2M transformations such as implementations of the OMG QVT language family. For example, the *QVTO* (sometimes referred to as *QVTo*) sub-project implements QVT *Operational mappings*.

3. Approach

Our approach of refactoring UML models and diagrams in parallel is based on the assumption that not only the UML model, but also the corresponding UML diagrams are internally represented as MOF compliant models. Then, not only the UML model but even the diagrams may be transformed (=refactored) using M2M transformations. As UMLDI (and the forthcoming DD) are based on MOF, our observation is that many UML editors fulfill this assumption by supporting the UMLDI format for diagrams.

Although refactoring of MOF-based models can be implemented using low-level programming languages [5], using M2M languages has the advantage of being more high-level for the domain of model transformation. Furthermore, transformation engines are typically interpreters, thus hard-coded refactoring implementation can be avoided and refactorings may even be added at run-time.

While a couple of M2M languages exist, even some that are specific to describe refactoring transformations [18] and thus even more high-level for the refactoring domain, we decided to use QVT. The reason was mainly that QVT is the official OMG M2M language and therefore broad tool support exists, often even built-in into UML tool suites. From the QVT family, we chose in particular *QVT Operational mappings* because it is suitable for refactoring implementers that grew up with the imperative paradigm.

If XMI files for the UML model as well as for the diagram are available, then the QVT transformations could be applied by a standalone QVT engine outside of a UML editor. In practise it is more reasonable to apply refactorings in a more interactive style by integrating them into a UML editor.

4. Prototype Tool

To validate our approach, we created a prototype implementation as a plug-in for the Eclipse *Papyrus* editor and make in addition use of the Eclipse *QVTO* project. As only standard technologies (MOF-based UML metamodel, UMLDI diagram format, QVT transformations) are used, a comparable implementation should be also possible for other UML editors as long as they are based on these standard technologies. Note that in Papyrus, the diagram is called *notation*.

As we wanted also contribute to enlarging the catalog of UML refactorings, we developed two new refactorings for UML activity diagrams and activity-related model elements: *Merge actions* that merges two action nodes into one (including adjustment of involved edges) and the inverse refactoring *Divide actions*. (Details about these refactorings including mechanics can be found in the thesis of Einarsson [7].) A before-after example for the *Merge actions* refactoring can be found in figures 2 and 3.

4.1 Implementation

The implementation consists of just three plug-ins for Papyrus: 1) `is.hi.umlrefactoring.ui`: adds a refactoring menu to the Papyrus user-interface and invokes the QVTO refactoring transformations when clicked on the menu entry, 2) `is.hi.umlrefactoring.core.transformations`: bundles the files that contain the actual QVTO transformations, 3) `is.hi.umlrefactoring.core.libraries`: contains Java implementations of black-box functions that are called by the QVTO transformations. The workflow between these plug-ins is depicted in Figure 1: when the user selects a refactoring in Papyrus, the invocation library starts with fetching the `*.uml` file containing the UML model and the `*.notation`

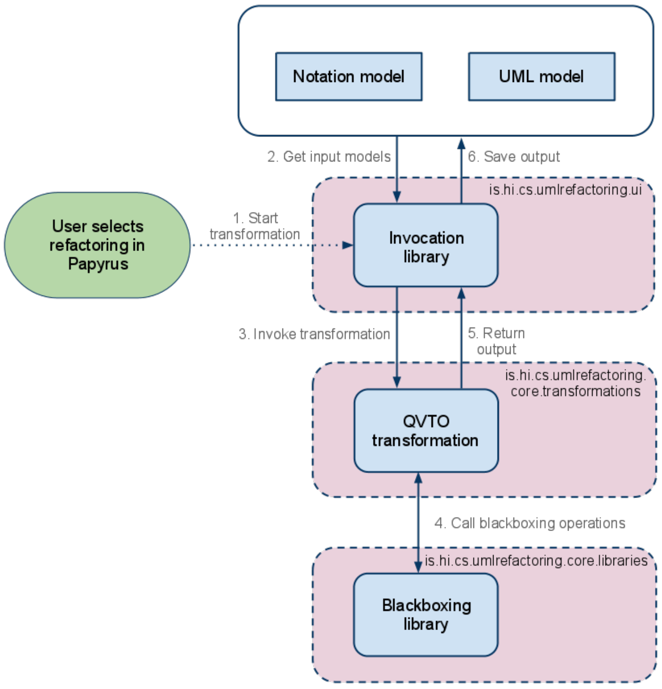

Figure 1. Workflow of refactoring transformations

file containing the diagram model. It then sends the models to the QVTO transformation which might call black-box functions if needed. After the transformation has been executed, the invocation library saves the output, overwriting the original models (Papyrus will then re-read these).

The implemented Java code is just user interface and glue code and some black-box code for QVT. The actual transformation is contained in the QVTO code.

4.1.1 Transformations

The transformation needed for the *Merge actions* refactoring is shown in Listing 1. The first line imports the QVT black-box library that contains some helper functions that needed to be implemented in Java. Lines 3–5 specify the metamodels involved in the transformation: UML for the UML model, NOTATION for the diagram model, ECORE is needed because UML and NOTATION are based on EMOF/*Ecore*. Line 7 defines the input and output metamodels of the transformation. The configuration properties in lines 8–9 hold values that are passed into the transformation by the caller (the Java code called when clicking on the refactoring menu entry): the properties toMerge1 and toMerge2 are strings that contain the GUID of the diagram elements that were in the editor selected for merging. Further properties (=global variables) that are used later are declared starting from Line 11. The entry point into the transformation is Line 18: the first two lines refactor the diagram, then the model is refactored. To this aim, Line 19 calls on every model element of the diagram model that is of type Shape (=a node

```
1  import m2m.qvt.oml.UmlUtilities;
2
3  modeltype NOTATION uses 'http://www.eclipse.org
       /gmf/runtime/1.0.2/notation';
4  modeltype UML uses 'http://www.eclipse.org/uml2
       /3.0.0/UML';
5  modeltype ECORE uses 'http://www.eclipse.org/
       emf/2002/Ecore';
6
7  transformation MergeActions(inout notation :
       NOTATION, inout uml : UML);
8  configuration property toMerge1 : String;
9  configuration property toMerge2 : String;
10
11 property objToMerge1 : notation::Shape = null;
12 property objToMerge2 : notation::Shape = null;
13 property edgeToRemove : uml::ActivityEdge =
       null;
14 property incomingEdgesToTransform : Set(
       ActivityEdge) = Set{};
15 property nodeToRemove : uml::ActivityNode =
       null;
16 property targetNode : uml::ActivityNode = null;
17
18 main() {
19   notation.objectsOfType(Shape) ->
         getSelectedObjects();
20   notation.objectsOfType(Shape) -> map merge();
21   uml.objectsOfType(ActivityEdge) -> map
         setTarget();
22   uml.objectsOfType(ActivityNode) -> map
         changeName();
23   uml.objectsOfType(ActivityEdge) -> map
         removeEdge();
24   uml.objectsOfType(ActivityNode) -> map
         removeNode();
25 }
```

Listing 1. The MergeActions.qvto transformation

in an activity diagram), the query (=function that does not transform a model) getSelectedObjects.

This query checks (lines 28 and 32 of Listing 2) for each object on which it is called whether its GUID matches the GUID stored in configuration properties toMerge1 and toMerge2. If yes, the properties objToMerge1 and objToMerge2 are set to the respective references of these Shape objects (lines 29 and 33). This is needed, because later, real reference of the objects have to be used, but for the elements selected in the Papyrus editor, only GUID strings can be passed into the QVTO code. The function hasGlobalId that is called in lines 28 and 32 is a black-box function implemented in Java. While QVT provides a function _globalId() to obtain the GUID of an element, this function is not available in Eclipse's QVTO implementation. Therefore, a black-box function was needed that accesses the URI of an element using Java (Line 2 of Listing 3).

4.1.2 Diagram Transformation

The actual refactoring on the diagram level is performed by applying in Line 20 of Listing 1 the merge() mapping (=a function that transforms an input into an output) on every model element of the diagram model that is of type Shape.

```
query Shape::getSelectedObjects() : Void
{
 if (self.hasGlobalId(toMerge1)) then {
  objToMerge1 := self;
 } endif;

 if (self.hasGlobalId(toMerge2)) then {
  objToMerge2 := self;
 } endif;
}
```

Listing 2. The getSelectedObjects() query

```
public static boolean hasGlobalId(Object
    shapeElement, String fragment) {
 String elementFragment = ((EObject)
     shapeElement).eResource().getURIFragment((
     EObject) shapeElement);
 return elementFragment.equals(fragment);
}
```

Listing 3. The Java hasGlobalId() black-box operation

The QVTO code of the merge() mapping is provided in Listing 4. The first line specifies that this mapping takes a Shape from the notation model as input and output. The next line (Line 37) guards that the transformation is only applied to the first of the two selected activity nodes to be merged (the other node and all connected edges are transformed as well when applying this transformation to the first node). In lines 39–41, local variables are declared that will be used to reference the node that comes first in the control flow of the nodes to be merged and the one that is last and the edge between these nodes, respectively. These local variables are set in lines 43–57 by checking for each incoming edge (targetEdges in the notation metamodel) if it comes from objToMerge2 (lines 43–49) or for each outgoing edge (sourceEdges in the notation metamodel) if it targets objToMerge2 (lines 51–57). If the two nodes are connected (Line 59), the actual diagram transformation is performed in lines 60–69 based on the refactoring mechanics: the first node and the connecting edge between the two nodes are removed and all edges that are connected to the first node will get connected to the remaining second node that takes the role of the resulting merged node.

In Line 60, the connecting edge that will be removed from the diagram model is used to look up the corresponding edge in the UML model (of type ActivityEdge in the uml metamodel) and the reference to it is stored in the property edgeToRemove that has been declared in Line 13 of Listing 1. (That edge needs to be removed by the later refactoring on the UML model level, so we make use of the fact that each diagram model element has a reference to the corresponding element in the UML model and memorise it in a global variable for later use by the UML model refactoring.) The same is done for the first node (Line 61) that will be removed as part of merging the two nodes and for the second node (Line 62), the node that will "survive" the merger.

Until now, only variables have been set. The first part of the actual diagram transformation is performed in lines

```
36 mapping inout notation::Shape::merge()
37  when { self.hasGlobalId(toMerge1) }
38  {
39   var firstNode : Shape = null;
40   var lastNode : Shape = null;
41   var connectingEdge : Edge = null;
42
43   self.targetEdges->forEach(incoming) {
44    if (incoming.source = objToMerge2) then {
45     firstNode := objToMerge2;
46     lastNode := objToMerge1;
47     connectingEdge := incoming;
48    } endif;
49   };
50
51   self.sourceEdges->forEach(outgoing) {
52    if (outgoing.target = objToMerge2) then {
53     firstNode := objToMerge1;
54     lastNode := objToMerge2;
55     connectingEdge := outgoing;
56    } endif;
57   };
58
59   if ((not (firstNode = null)) and (not (
         lastNode = null)) and (not (
         connectingEdge = null))) then {
60    edgeToRemove := connectingEdge.element.
         oclAsType(ActivityEdge);
61    nodeToRemove := firstNode.element.oclAsType(
         ActivityNode);
62    targetNode := lastNode.element.oclAsType(
         ActivityNode);
63    firstNode.oclAsType(Shape).targetEdges->
         forEach(incomingEdge) {
64     incomingEdge.target := lastNode;
65     incomingEdgesToTransform += incomingEdge.
          element.oclAsType(ActivityEdge);
66    };
67
68    notation.removeElement(connectingEdge);
69    notation.removeElement(firstNode);
70   }endif;
71  }
```

Listing 4. The diagram merge() mapping

63–65: Each edge that is an incoming edge of the first node (Line 63) is attached to the second node, by setting the target of the edge to the second node (Line 64). These edges are also added to a property (Line 65) that is later-on used by the UML model refactoring to apply the same change also on the model level. Once these adjustments have been made, the connecting edge and the first node can be removed from the diagram (lines 68 and 69).

4.1.3 UML Model Transformation

Based on the references to the UML model elements that have been determined and saved during the transformation of the diagram model, the transformation of the UML model is straightforward: for all edges (model element ActivityEdge, Line 21 in Listing 1) it has to be checked whether it is an edge that is contained in the determined set of edges that need to be transformed (Line 73 of Listing 5). If yes, this is an edge that was incoming to the first node and thus the

```
72 mapping inout uml::ActivityEdge::setTarget()
73   when { incomingEdgesToTransform->includes(self
         )}
74   {
75     self.target := targetNode;
76   }
77
78 mapping inout uml::ActivityNode::changeName()
79   when { self = targetNode }
80   {
81     self.name := nodeToRemove.name + " - " + self
           .name;
82   }
83
84 mapping inout uml::ActivityEdge::removeEdge()
85   when { self = edgeToRemove }
86   {
87     uml.removeElement(self);
88   }
89
90 mapping inout uml::ActivityNode::removeNode()
91   when { self = nodeToRemove }
92   {
93     uml.removeElement(self);
94   }
```

Listing 5. The UML model mappings

target of that edge has to be adjusted to the second node, the target node (Line 75). In the next step (call for every ActivityNode model element in Line 22 of Listing 1, implementation in lines 78–82 of Listing 5), the name of the second node is set to the concatenation of the name of the first node and the second node (including a separator between the names of the two merged activities). Finally, for all ActivityEdge and ActivityNode model elements, corresponding mappings are called (lines 23–24 in Listing 1) that remove a model elements from the UML model when it is an element that has been determined to be removed (lines 84–94 of Listing 5).

4.2 Application

The implemented refactorings can be used to refactor UML models and diagrams in parallel. As the typical user works on the diagram level, the refactorings are invoked by selecting the elements to be refactored in the Papyrus diagram editor window and selecting a refactoring menu entry.

Figure 2 shows an excerpt of a UML activity diagram. The two nodes Prepare order and Fill order that shall be merged are selected and the *Merge actions* refactoring entry is selected in the refactoring menu that has been added to the Papyrus context menu. Figure 3 shows the resulting activity diagram after the *Merge actions* refactoring has been applied: The two activity diagram action nodes have been merged into one action containing the text Prepare order - Fill order and the edges have been adjusted accordingly.

The XMI representations of the original and the refactored UML model not shown, but the underlying UML model has been refactored in parallel. The interested reader is referred to thesis of Einarsson [7] to see the XMI representation of UML model and diagram.

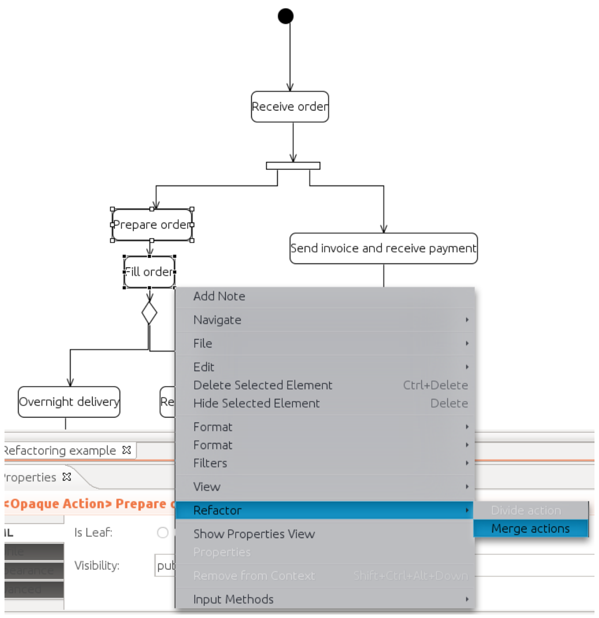

Figure 2. Selecting *Merge actions* refactoring on diagram

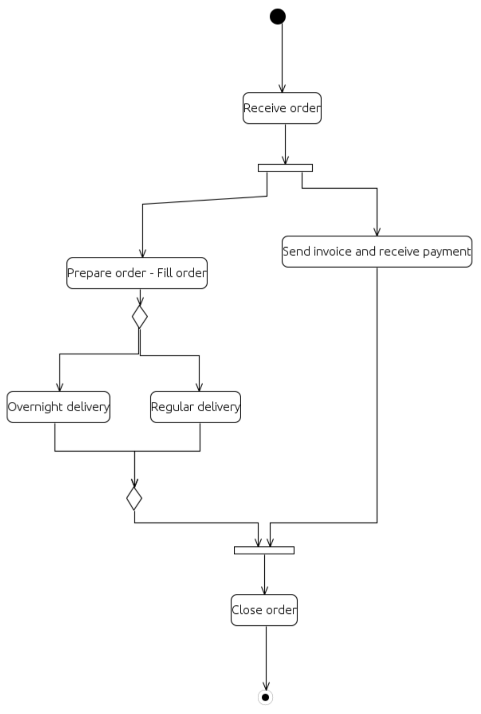

Figure 3. Activity diagram after applying *Merge actions*

4.3 Discussion

The prototype implementation demonstrates that parallel refactoring of UML models and diagrams is feasible using M2M transformations. Moreover, it has been shown that general purpose transformation languages, such as QVT, are applicable to implement refactorings. Even if the underlying metamodels or QVT do not support a needed feature, this can be easily added due to QVT's black-box approach.

The presented QVTO code looks quite close to an imperative implementation language, but in contrast to a low-level implementation language, where all pattern matching and data structure handling needed to be explicitly coded, this is here provided by QVT. As a result, the complete implementation for the two refactorings is quite short in terms of *Lines Of Code* (LOC): 205 LOC of QVTO, 57 LOC of Java for the black-box library and 284 LOC of Java for the user interface and calling the refactorings.[1] It can be expected that implementing the refactoring transformation using Java would require significantly more LOC. Furthermore, a Java implementation would be specific to the data structures internally used to represent the model, thus the transformation implementation would be tool specific.

Except for a few short black-box functions that are specific to the Eclipse Modeling Project, our transformations are completely tool independent and thus applicable using any tool that has QVT support. This allows to re-use the QVT implementations of the refactoring transformation in different tools. If the diagrams are stored in UMLDI format, then not only the model refactorings but also the diagram refactorings can be exchanged between different UML tools.[2]

In our prototype implementation, the UML model refactoring is steered by the diagram refactoring because to identify the UML model elements to be transformed we rely on the references to the UML model elements that are contained in the diagram. Accordingly only those elements that are found and transformed in the diagram model will also be transformed in the underlying UML model. Because a diagram is just a partial view on the model, it might be the case that the UML model contains more elements than the diagram. In this case, model elements do not get transformed if they are not contained in the diagram. Furthermore, it might be the case that two different diagrams refer to the same elements of the UML model: in this case, only the diagram on which the refactoring has been invoked will be transformed, any other diagrams remain untouched even though the shared underlying UML model gets changed. As the UML model does not contain any references to diagrams, the UML model cannot be used to steer the diagram refactorings. A solution that addresses these problems would apply independent, but related transformations on the UML model as well as on all diagram models. When refactoring traditional programming languages, a reference finder is used to find all locations that need to be updated and to apply thus changes consistently at all places where needed. When refactoring models using QVT, we expect that the pattern matching mechanisms of QVT simply finds all the locations that refer to a certain model element.

[1] The source code is available from `http://xp-dev.com/svn/UMLrfp/`

[2] Note that in practise due to different interpretation of the XMI specification and different implementations of the UML metamodel, XMI exchange may not work between tools.

Currently, we make no use of the Eclipse *Language Toolkit* (LTK). LTK provides wizard dialog pages for entering refactoring parameters, invoking refactoring precondition checking, and previewing changes. However, as we do not need to enter any additional refactoring parameters (the name of the actions in the refactored activity nodes can just be adjusted in the UML editor using ordinary interactive editing) for the sample refactoring implementations, we decided not to use LTK.

5. Related Work

Boger et al. [2] describe refactorings for UML state chart and activity diagrams and implemented these in a refactoring browser. As this is a plug-in for the commercial *Poseidon for UML* tool which allows diagram editing and uses an underlying UML model, it can be assumed that simultaneous refactoring of model and diagram is performed. However, they do not explicitly discuss this aspect and thus nothing about the implementation is known.

None of the other major commercial UML tools supports refactoring that goes beyond simple refactorings such as renaming elements. These do not even require to change the internal diagram representation: Consider, for example, renaming a model element. Just as described for Papyrus in Section 2.2.3, the internal diagram representation does only contain the graphical position of a model element, but not the text to be displayed. Instead, a reference to the model element is stored and that reference is used to obtain the text to be displayed. Therefore, simple refactorings do not require to keep transform model and diagram in parallel.

All further tools that support UML model refactoring are from academia: Most of the refactoring tools are not integrated into a UML editor, but standalone. These are then restricted to transform just the models, but not diagrams. This is probably due to the fact that the diagram file formats used by graphical UML editors are often proprietary and thus only the model can be accessed since only these are stored in the XMI format. An overview on older academic UML refactoring tool approaches is given by Dobrzański [4].

More recently, a couple of EMF-based refactoring approaches have been developed that are typically not specific to UML but applicable to everything that has an EMF-based metamodel. For example, the *EMF Refactor* project [5] is a sub-project of the *Eclipse Modeling Project* that provides extensible tool support for generating and applying refactorings. For specifying the model transformations, not QVT, but either low-level Java implementation or the transformation language *Henshin* [1] are suggested. This project provides however only generic infrastructure (for example an LTK-based preview of model changes using a tree-view of the EMF model) and no concrete refactorings, such as UML refactorings. Another example is *Refactory* [18] that provides generic refactorings that can then be mapped to various different EMF-based metamodels and are thus independent from the language to be refactored. A custom transformation language that is specific to the refactoring domain has been developed and is used. Although some of these generic refactorings are applicable to UML models, the concrete syntax in form of UML diagrams are not considered.

Markovic and Baar [9, 10] refactor UML and OCL expressions that refer to the UML model in parallel. Just like the approach presented in this paper, they use QVT for the transformations. However, they do not consider diagrams — only the model. But keeping UML model and OCL expressions in sync during refactoring is comparable to keeping UML model and diagram in sync as in our approach.

Diskin and Czarnecki [3] provide an algebraic framework of synchronising transformations of different models having a related metamodel. Like in our prototype implementation, they map changes applied to one model on changes applicable to another model, thus keeping both in sync.

6. Summary and Outlook

In this paper, we have presented an approach and a prototype implementation for synchronous refactoring of a UML diagram and the underlying UML model using *Model-to-Model* (M2M) transformations. In other known approaches that use M2M transformations to perform UML refactorings, only the underlying UML model is subject to transformation, thus diagram and model may get out of sync. Storing the model and the diagram using a MOF compliant metamodel enables the usage of the model transformation language QVT for applying the refactoring transformations. Using a transformation language such as QVT has the advantage that refactoring implementations are much shorter in comparison to using a low-level implementation language.

Our prototype tool was implemented as a plug-in for the Eclipse Papyrus UML editor. However, our QVT refactoring transformations are tool-independent and can thus be used without modification with any tool that uses the XMI format for the UML model and the UMLDI format for diagrams.

Future work targets at two directions: On the implementation level, it is worthwhile to include Eclipse LTK for integrating the steps of refactoring precondition checking, refactoring parameter input, and (provided by the *EMF Refactor* project [5]) preview of model changes. On the conceptual level, the goal is to rewrite the QVT transformations as described in Section 4.3 with the aim of letting them independently from each other —but still in sync— apply comparable transformations on the model as well as on all diagrams.

While developing our QVT transformations, we realised that refactoring support for QVT code would be handy. This is currently completely lacking and worthwhile to work on.

A topic that was out of scope of our work is synchronous refactoring of implementation language source code, such as Java code, and related UML models. The problem is here that programming languages typically have no metamodel, thus M2M transformations are not applicable.

References

[1] T. Arendt, E. Bierman, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MoDELS'10: Proc. 13th international conference on Model Driven Engineering Languages and Systems, Part I*, volume 6394 of *LNCS*. Springer, 2010.

[2] M. Boger, T. Sturm, and P. Fragemann. Refactoring Browser for UML. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*. Springer, 2003.

[3] Z. Diskin, Y. Xiong, and K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, 10:6:1–25, 2011.

[4] Ł. Dobrzański. UML Model Refactoring-Support for Maintenance of Executable UML Models. Master's thesis, Blekinge Institute of Technology, School of Engineering, Ronneby, Sweden, 2005.

[5] Eclipse Foundation. EMF Refactor Project. `http://www.eclipse.org/modeling/emft/refactor/`, 2012.

[6] Eclipse Foundation. Eclipse Modeling Project. `http://www.eclipse.org/modeling/`, 2012.

[7] H. Þ. Einarsson. Refactoring UML Diagrams and Models with Model-to-Model Transformations. Master's thesis, Faculty of Industrial Engineering, Mechinal Engineering and Computer Science, University of Iceland, Reykjavík, Iceland, 2011. URL `http://hdl.handle.net/1946/8624`.

[8] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[9] S. Markovic and T. Baar. Synchronizing Refactored UML Class Diagrams and OCL Constraints. In *1st Workshop on Refactoring Tools (WRT 2007), Berlin, Proceedings*, 2007.

[10] S. Markovic and T. Baar. Refactoring OCL annotated UML class diagrams. *Software and System Modeling*, 7(1):25–47, 2008.

[11] OMG. UML Diagram Interchange Version 1.0.0 (formal/2006-04-04), 2006.

[12] OMG. Meta Object Facility (MOF) Core Specification Version 2.4.1 (formal/2011-08-07), 2011.

[13] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1 (formal/2011-01-01), 2011.

[14] OMG. Unified Modeling Language (OMG UML) Infrastructure Version 2.4.1 (formal/2011-08-05), 2011.

[15] OMG. Unified Modeling Language (OMG UML) Superstructure Version 2.4.1 (formal/2011-08-06), 2011.

[16] OMG. MOF 2 XMI Mapping Specification Version 2.4.1 (formal/2011-08-09), 2011.

[17] OMG. Object Constraint Language (OCL) Version 2.3.1 (formal/2012-01-01), 2012.

[18] J. Reimann, M. Seifert, and U. Aßmann. Role-based generic model refactoring. In *MoDELS'10: Proc. 13th international conference on Model Driven Engineering Languages and Systems, Part II*, volume 6395 of *LNCS*. Springer, 2010.