# Realizing UML Model Transformations with USE

Fabian Büttner[1] and Hanna Bauerdick[2]

[1] University of Bremen, Computer Science Department, Database Systems Group
[2] University of Bremen, Center of Computing Technologies (TZI), Artificial Intelligence Group

**Abstract.** The USE (UML-based Specification Environment) tool has been successfully applied for model validation in the past. In our current work, we are enriching the USE specification language with imperative elements. We employ this extension as an assembler to realize UML model (class diagram) transformations with USE in a flexible way: UML transformations are described using a custom abstract language based on object diagram-like patterns. These descriptions are automatically translated into the imperative USE extensions. Our approach aims to provide a flexible instrument to experiment with different transformations and transformation formalisms.

## 1 Introduction

In the last years, model transformation has become an increasingly important field within software development. At the same time, the notion of a *model* has become more or less a synonym for models in the object-oriented paradigm. Today, the OMG is about to finalize the Query, Views, Transformations (QVT) [OMG06] specification, which aims to provide a set of standardized formalisms to transform one object-oriented model into another one.

We have successfully employed our USE tool [GR02] for validation of static structure models in the past. In our current work, we are employing USE to develop, apply, and validate model transformations. Although QVT is about to be finalized soon, we feel that we still need more evidence on how well it fits for different kinds of transformations. Our approach gives us a flexible instrument to experiment with different transformations and transformation formalisms in a common and accessible environment.

In earlier work, we utilized several ways to describe transformations of structural models [Büt05,BG06]. As a central part, we have been working on a catalog of transformations of OCL annotated UML class diagrams. In our current work, we are now implementing this catalog based on our extension to USE. We started with a very simple transformation language based on object diagram-like patterns and regular expressions. Later, we realized a need for certain extensions to this language to describe our transformation catalog with reasonable effort. Due to the flexibility of our approach, this extensions could be realized quite easily. We motivate the extensions here, too, because we feel that the underlying problems may also occur in other model transformation scenarios.

On the technical level, we did two things: We first added a small imperative OCL-based language to USE which can be used to define operations. We then created a simple transformation language, based on UML diagrams and a few elements of graph transformation [Roz97]. We then implemented a UNIX filter like program which translates our transformations into the simple imperative language. This way, we can employ USE as a "virtual machine" to execute and validate transformations, under (potentially) various transformation formalisms.

Several other approaches to object-oriented model transformation exists. [CH03] provides a general classification. Existing model transformation frameworks include ATL [JK05], the TopMODL initiative [MDFH04], Modelware [Mod], and the graph transformation-based Fujaba [FUJ]. Our work also resembles [MFJ05] and Kermeta [FDVF06] as it adds executability to meta-models.

This paper is structured as follows: In Sect. 2, we introduce our extension to USE and our transformation language. We then take an excerpt (a "Many2One" transformation) of our transformation catalog to illustrate our approach in Sect. 3. After describing Many2One in general, we show how we implemented the transformation in USE. Section 4 concludes this paper.

## 2 Realizing UML Model Transformations with USE

In a nutshell, we are employing USE to apply transformations to UML models – or more specifically, to UML class diagrams. Although USE directly supports class diagrams (USE specifications are mainly class diagrams with constraints), we represent them as object diagrams of the UML meta-model here. The USE specification is provided by the UML 2.0 meta-model in our current work.

The two basic ideas of our approach are as follows: *1) The USE specification language is extended to support imperative descriptions of operations.* We achieve this by enriching USE with a minimal object-oriented programming language that can be used to modify a system state (i.e., an object diagram). This language resembles the "ImperativeOCL" language of the upcoming QVT specification. We employ this new feature of USE to formulate additional operations with side-effects for the UML meta-model.

*2) Transformations are applied to UML models by adding additional transformations objects to their meta-level representation.* These transformation objects provide operations that modify the model in the intended way. The operational transformation semantics is located in explicit transformation meta-classes.

The remaining section explains this sketch in more detail.

### 2.1 Models Everywhere

Figure 1 shows an overall structural picture of our approach. It shows the different involved modeling artifacts, which creates them, and how they are realized using USE.

Starting in the lower left-hand corner, we locate the role of the *modeler*. That is the origin of our initial model, an exemplary PersonCompany class diagram (which we will modify by means of model transformation in Sect. 3).

This class diagram is created as an instance of the UML meta-model (UMLOCL2 in Fig. 1) in USE. As the name suggests, UMLOCL2 actually combines the UML2 and OCL2 meta-models into one meta-model. It is defined as a USE specification (UMLOCL2.use). So far, this specification does not require any of the extensions mentioned above. In addition, an augmented version UMLOCL2withTransformations.use exists which contains further meta-classes which we describe below.

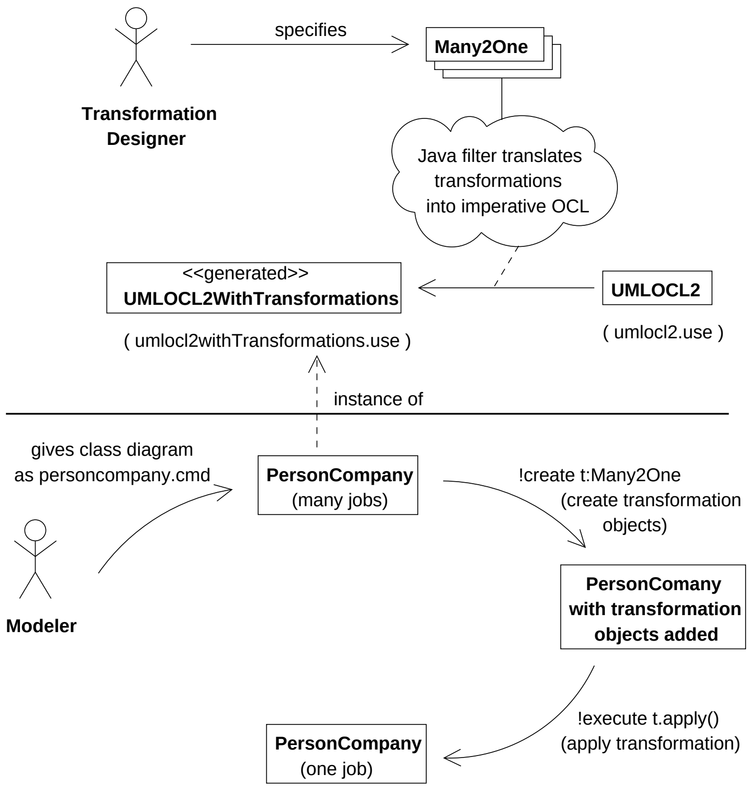


**Fig. 1.** The overall picture

The modeler creates his or her model in USE as a sequence of state manipulation command (a .cmd-file). After reading and executing this sequence, USE holds the PersonCompany class diagram as an instance of UMLOCL2withTransformations. Of course, manually specifying a UML class diagram as a meta-model instance is not a pleasing task. Therefore we created an import filter, that converts USE specifications (.use-files) into instances of the UML meta-model (.cmd-files that are executable w.r.t. UMLOCL2.use). But this is only for convenience and is not required for our approach. Notice that PersonCompany could also be instantiated as an instance of UMLOCL2, as only meta-classes from this meta-model are used up to this point. But since we want to enable transformations on PersonCompany, we are using the latter one.

The modeler can now choose one of the transformations that have been previously defined by the *transformation designer* (explained soon). From the modelers point of view this means to add an instance of a corresponding transformation meta-class (Many2One in Fig. 1) to the meta-level representation of his or her class diagram. This transformation meta-classes contain operations with side-effects that realize the intended effect. By convention, each transformation meta-class defines an entry point operation *apply()*. By invoking *apply()* the transformation object changes its surrounding instances of the UML meta-model. This may require several internal steps. Finally, the transformation object is removed again, leaving a meta-level representation of the modified PersonCompany class diagram (for example, with only one job per person now). Alternatively, transformation objects can also be kept for tracing reasons.

How do the transformation meta-classes come into play? First, the transformations are developed by the aforementioned *transformation designer*. In our case study we employ a formalism which consists of simple transformation steps (described as object diagrams with some minor extensions) and regular expressions controlling the correct execution sequence of the individual steps. Each transformation consists of one control expression and one or many transformation steps. Transformations described in this formalism can be visualized in a UML-like representation (extended object diagrams). For our purpose, we have developed a textual syntax, too.

These transformation descriptions, having a high level of abstraction, are then translated into transformation meta-classes using the minimal imperative language we added to USE. As a result we achieve an enriched version of the original UML meta-model. For each transformation class, several internal operations are created to implement the pattern matching and the application of the transformation steps, and the overall control expression. Furthermore, new associations can be added to the meta-model to allow to specify the context for the transformation (e.g., to specify the target association end in the example in Sect. 3). Finally, the existing meta-classes can be enriched by further operations to implement cross-cutting functionality such as cloning or component exchange (described later on).

Currently, this translation (or compilation) from our high level transformation formalism into the imperative language is realized as a Unix filter like Java program. This program takes the original UML meta-model (UMLOCL2.use) and the textual transformation description (ManyToOneTrans.txt) as input files and yields a modified version of the original .use-file. More than one transformation can be added by applying this step repeatedly.

### 2.2 The imperative extensions to USE specifications

We shortly introduce the new USE specification language concepts. In previous versions of USE, operations could be specified in two ways: a) as OCL queries or b) by providing pre- and postconditions that characterize operation properties.

The first variant is side-effect free. Operations specified by pre- and postconditions are typically not side-effect free, but not automatically executable from USE's point of view neither. They can validated given a manually provided sequence of state manipulation command.

Now we have added a third mechanism to specify operations (with side-effects) in USE. It is a small imperative language that allows us to provide operational specification. It is build around OCL as an expression language. It further adds the following imperative elements:

- Basic state manipulation statements: create and destroy objects, insert and remove links between objects, set attribute values. (These operations were available in USE before in the (still existing) command files.)
- Flow control statements: execute conditionally (if-then-else), execute repeatedly (while), and iterate over collections.
- Invocation of other operations. Other operations can be invoked if they are also defined using this imperative language. Recursive invocation is supported. (Of course, OCL query operations can be used as well, but only in expressions.)

This language resembles the operational mappings language provided by OMG QVT.

### 2.3 The transformation language used for the catalog

Due to the expressive power of the added imperative language, we can now realize arbitrary transformations on instances. This can be done, as illustrated in the beginning, by putting one or many transformation operations into an explicit transformation class. But writing transformations on this level is still a tedious task for complex transformations. Several higher level alternatives exist that are more appropriate to this task, such as graph transformation or relation-based approaches like in QVT.

Therefore, we created a small, more abstract transformation language to formulate our UML class diagram transformation catalog. Instead of implementing this transformation language in USE, too, we defined a mapping onto the small operational language introduced above. This mapping is currently realized by a small, external piece of software which acts like a Unix filter. It reads an existing source specification and one or many transformation descriptions and creates a specification which is enriched by new *transformation classes* that implement the transformations.

In our transformation language, a transformation consists of two parts: a set of transformation steps and a control expression. The transformation steps build the atoms of a transformation. They are specified as extended UML object diagrams. The control expression then specifies when and in which order the individual steps have to be applied. The control expression language is a regular language whose terminal symbols are the transformation steps.

(Transformation steps) Figure 2 shows a simple transformation step, an extended object diagram. A UML diagram diagram can be regarded as a restricted

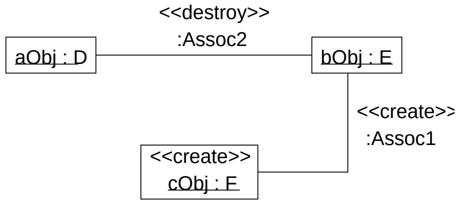


**Fig. 2.** A simple transformation step

form of a graph transformation rule: Unless marked otherwise, the objects and links in the diagram determine the context of a gratra rule. Elements marked with the stereotype «create» are created by the rule. Elements marked with «destroy» are destroyed by the rule. The destroyed and context elements determine the redex of the rule, i.e. the pattern which must be found in the system state in order to apply the rule. In Fig 2, the redex of the transformation step consists of the objects aObj, bObj, and the Assoc2-link that connects them. In order to apply the step, actual assignments for aObj, bObj and the Assoc2-link have to be found in the system state. Given a certain valid redex, applying the rule will create a new object of class F, connected to bObj, and destroy the Assoc2-link. We further allow OCL preconditions to be part of the extended object diagrams. This way one can further restrict what is a valid redex (not shown in Fig 2).

(Translation of transformation steps) For each step two operations are created in the corresponding transformation class (Many2One in the example which is described in detail in Sect. 3). First a *stepname_redex()* operation is defined which returns a redex for the step. The result type is a tuple consisting of all objects that determine the context of the step. In the above example, the signature is *exampleStep_redex() : Tuple(aObj:D, bObj:E)*.[3] Second, a *stepname_apply()* operation is applied which takes a redex tuple and realizes the actual effect of this step. For the example step, this is: *exampleStep_apply(redex:Tuple(aObj:D, bObj:E))*.

(Control expression) A transformation can consists of several steps. The transformation control expression is a regular expression that controls how the steps are combined. An example may look like this:

**MyTransform := Init StepA* (StepB | StepC)* CleanUp**

The syntactical elements are as usual. Star means as long as possible (include zero times), parenthesis group steps, the bar specifies alternatives. Note that we do not allow a recursive language definition (i.e., the derivation tree has to be acyclic). However, it is still possible to create infinite loops, due to '*' expressions. It is the transformation developers responsibility to ensure that the transformation process terminates.

[3] The links are not a part of the redex tuple, because we assume relation semantics for associations. Multiple links of the same association are not allowed between a pair of objects.

To implement our transformation class for a transformation, we create three elements in the class:

1. A *step()* operation that applies one step (if possible). This operation uses the various ..._redex() operations to determine the next applicable step().
2. A *state* attribute which keeps track of the current transformation state (a state of the finite automata created from the regular control expression).
3. An *apply()* operation which simply applies step() as long as possible. This is the entry point to apply the transformation.

## 3 Case Study

In this section, we show how we implemented the aforementioned transformation catalog with the extended version of USE. Our transformation catalog resembles the refactorings catalog of Martin Fowler [Fow99]. It contains class diagram transformations such as moving methods from one class to another, changing generalizations into compositions, modifying associations, inlining, and extracting classes.

The major feature of our catalog is that it considers OCL annotations on the UML class diagrams. Because OCL expressions depend on the underlying class diagram, they have to be incorporated when changing its structure. Of course on this account, the transformations become more complex.

Due to this complexity our catalog provides a good example to validate an approach to model transformation (we think). This section provides an exemplary insight into the catalog and its realization in USE. We pick one transformation (Many2One) which changes an association multiplicity from many to one. In the following, we first explain Many2One in more detail, independent of its realization in USE. Then we describe Many2One by means of the previously explained meta-model transformation steps and its control condition. Finally, we illustrate how one of these steps is translated into a USE specification, i.e., into imperative operations.

### 3.1 Example Transformation: Many2One

Transforming an association multiplicity from *many* to *one* is conceptionally simple but still interesting: a simple modification to the class diagram part has some more demanding consequences on those existing OCL formulas that depend on the modified association.

In OCL, every expression is typed. The type determines which operations can be applied to its values. When navigating through an association end in an OCL expression, the type of this expression is determined by the end's multiplicity. If the multiplicity is `1` or `0..1`, the navigation expression has an object type. Otherwise, the expression results in a collection.

Consequently, when changing the multiplicity of an association end from *many* to *one*, the navigation expression along this end changes from collection-valued to object-valued. Every OCL expression containing this navigation as a sub-expression is affected. Accordingly, the Many2One transformation also consider the OCL expression parts when changing a class diagram. The Person-Company class diagram in Fig. 3 illustrates the necessary modifications.

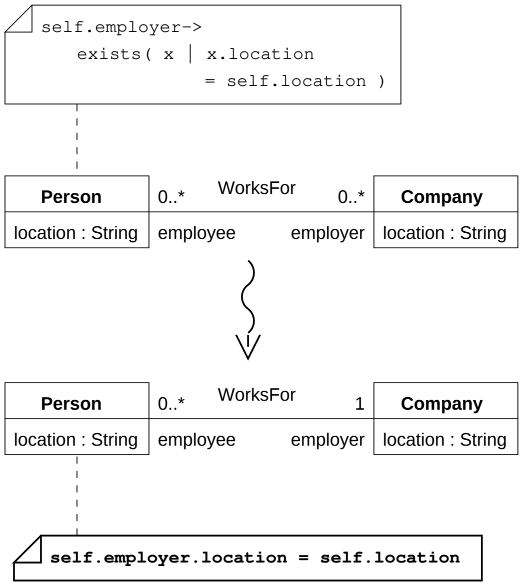


**Fig. 3.** Transforming an association multiplicity from many to one

Both UML class diagram fractions describe an employer-employee relationship where a person resides in a location and works for a company. The upper invariant states that a person has to work for at least one company which is located in the same place. The depicted model transformation changes the multiplicity at the employer end from arbitrary to one, i.e. that after the transformation every person work in exactly one company. Accordingly, the type of the navigation expression `self.employer` changes from collection-valued to object-valued. Therefore, the invariant should be simplified to the expression stated in the lower part of Fig. 3.

Most implied OCL model transformations are dependent on the performed class diagram transformation. If one for example carries out a class diagram transformation which changes an attribute name, all OCL expressions which use this attribute name have to be adapted. However, if an association multiplicity is changed, more complex OCL expression transformations have to be performed, as explained above.

There is a great benefit if one keeps the number of dependent OCL model transformations to a minimum, because these transformations cannot be reused in other contexts. In case of Many2One, a separation between the dependent and the independent transformations can be realized by following the steps below:

1. Transform all OCL collection operations to `iterate`, if possible.

2. Change the association multiplicity (UML model transformation).
3. Adjust all effected OCL expressions.
4. Simplify the OCL expressions.

Only the second step realizes a transformation of the UML class diagram. The other three transformations refer to OCL expression transformations.

The first step maps all OCL collection operations to `iterate`, which is the most powerful collection operation. For almost every collection operation such a mapping can be defined (one exception e.g. is the `including` operation). The advantage of this mapping is that after the transformation nearly all collection operation expressions are iterate expressions. The number of used collection operations is definitely reduced to a minimum and only for this reduced operation set a corresponding transformation to the UML model transformation has to be found. The most important issue about the first step is that it describes equivalence transformations, i.e. these transformations can be applied in any case and does not change the semantic of the expressions. The same holds for the last step which simplifies the OCL expressions either by performing an inverse transformation to the first one if possible or by applying some basic simplification rules.

Only the third step depends on the UML model transformation and can only be applied if this concrete UML model transformation is performed. The UML class diagram transformation causes that the affected OCL expressions which normally result in a collection become object-valued (they result in exactly one element). Thus within this step, the collection operations have to be transformed to equivalent expressions which only uses object operations.

### 3.2 Realization of Many2One with USE

All transformations of the case study consist of control expressions and transformation steps. In this section the control expression and the steps of the Many2One transformation are explained.

As mentioned above, the control expression serves as the control structure which coordinates the execution order of the transformation steps. These control expressions are described using regular expressions. The following one describes the process of the Many2One transformation.

```
Many2One = Iteratorize* changeMultiplicity AdjustCollectionOps*
             Simplify*

Iteratorize = existsToIterate | forAllToIterate | ...

AdjustCollectionOps = AdjustIterate | AdjustIncluding | ...

AdjustIterate = inlineRangeForRangeVar inlineAccuInitForAccuVar
                  inlineIterateBodyForIterate

Simplify = iterateToExists | iterateToForAll | ...
```

These control expressions of Many2One are related to the above mentioned transformation steps. `Iteratorize` refers to the transformation of the collection operations to `iterate`. The association multiplicity of the UML class diagram is changed by the step `changeMultiplicity`. `AdjustCollectionOps` corresponds to the adaptation of the OCL expressions which were affected by the class diagram change. `AdjustIterate` for example transforms an `iterate` expression to an equivalent object-valued expression, if the source expression of iterate refers to exactly one element. This means that during the evaluation of the `iterate` expression always only one iteration is performed. Consequently, the control variables and the result variable can be replaced by their initialization values. These substitution steps are realized by all three `inline` transformation steps. The last step (`Simplify`) is related to the simplification of the OCL expressions. This transformation step realizes the inverse effect of `Iteratorize` and also applies some generic simplification rules.

In the following, some transformation steps of Many2One are exemplarily described to show the functioning of the transformation steps. Figure 4 shows an excerpt of the UML and OCL meta-model which is relevant for the transformation steps of Many2One and consequently fundamental for the understanding of those steps.

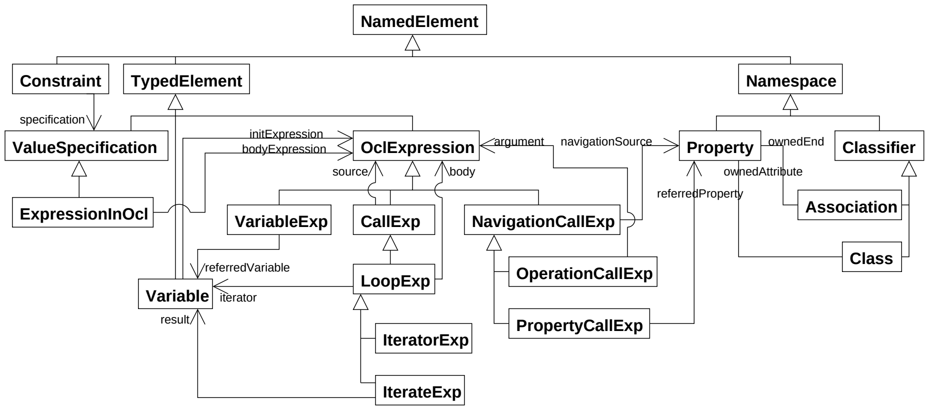


**Fig. 4.** Relevant excerpt of the combined UML and OCL meta-model

At first, we started to define the Many2One transformation using visual transformation rules. However, these rules quickly become very complex and hard to maintain. On this account, we defined some extensions to significantly reduce the complexity of the specifications, i.e. to reduce the number of transformation steps. In the following these extensions will be explained considering some Many2One transformation steps as example.

**Step `existsToIterate`:** If we take the example of Fig. 3 as a basis, the first transformation step which could be applied would be `existsToIterate`. In this example the expected result of this step would be the following expression:

```
self.employer->iterate( x; result:Boolean = false |
     result or x.location = self.location )
```

The transformation schema of the existsToIterate is depicted in the extended object diagram of Fig. 5.

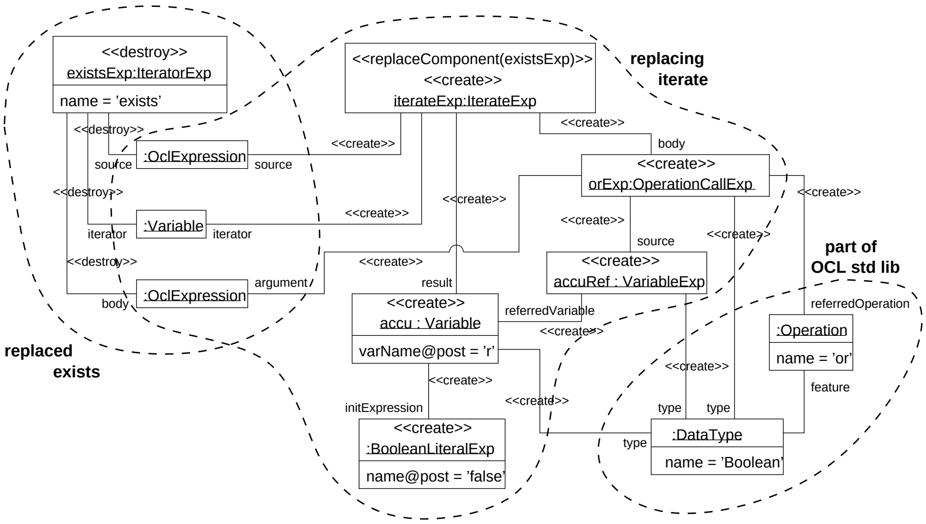


**Fig. 5.** Transformation step of `existsToIterate`

As mentioned in Section 2, all objects which are not created during this transformation step and their specified attribute values form the redex of this step. Within `existsToIterate` this redex consists of an exists expression (labeled with `replaced exists`) and the `or` operation and boolean type from the OCL standard library (marked as `part from OCL std lib`).

As explained before, the `existsToIterate` replaces the `exists` expression with an equivalent `iterate` expression. Consequently, the `replaced exists` part drops out during this step and is replaced by an iterate expression (labeled with `replacing iterate`) which holds the same source expression and control variables. By definition the iterate expression also owns an accumulator variable which is initialized with `false`. The body expression of `exists` now becomes the body expression of `iterate`, but is extended by the expression `result or`.

The replacement of expressions is one of our extensions which was introduced to reduce the number of transformation steps to a minimum. The replacement is realized by the stereotype «replaceComponent» and indicates that the owner of the labeled component should replace its part, specified by the parameter expression, with the caller component (e.g. in this case `exists` should be replaced with `iterate`). The effect is that every step only has to be defined once and not for each possible owner (e.g. if, let, invariant and operation expressions) of the replaced expression. For this purpose we have introduced an ownership and a part relationship within the UML and OCL meta-model which are derived from existing associations.

The `@post` term indicates another extension within the transformation steps which describes the setting or changing of attribute values during the transformation step. An example of this construct can be found in Fig. 5 where the initialization value of the accumulator is set to `false` during the transformation.

**Step `inlineRangeForRangeVar`:** The `existsToIterate` step already needed some of the extensions to be realized. It has also demonstrated how transformation steps are specified within USE in general. The following transformation step (`inlineRangeForRangeVar`) will employ the last remaining extension. As mentioned above `inlineRangeForRangeVar` is one of three steps which realizes the transformation from `iterate` to an object-valued expression. It substitutes the source expression of `iterate` for all occurrences of the control variables within the body expression. Fig. 6 shows the transformation scheme of `inlineRangeForRangeVar`.

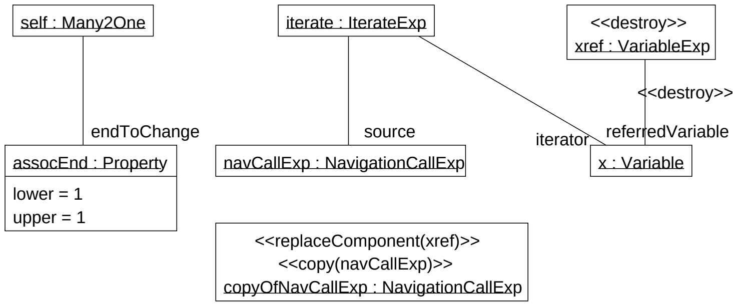


**Fig. 6.** Transformation step of `inlineRangeForRangeVar`

As before, the replacement of the control variable is realized using the «replaceComponent» stereotype. However, the new extension within this transformation step is the copying of an OCL expression. Within the UML and OCL meta-model several compositions, i.e. part-of relationships, are defined. These compositions state that an object should belong to at most one owner object. In our example transformation, the control variable can occur several times within the body expression of `iterate`. Because it should be substituted with the original source expression and because there can occur some part-of relationships to its owner, the source expression has to be copied.

This copying is visualized by the stereotype «copy» and is a mixture of shallow and deep copy. If the expression which should be copied has some composition relationships to its parts, these parts should be copied deeply. Otherwise, a shallow copy (i.e. no copying of the related objects, but insertion of links between the new copy and these objects) is sufficient. The shallow copy also reduces the complexity of this approach.

As shown in this section, the defined extensions highly reduce the number of the transformation steps, especially the «replaceComponent» stereotype. We have also detected that some kind of copying is essential for the definition of more complex transformations.

### 3.3 Excerpt of the resulting USE specification

Finally, the following subsection shows an excerpt of the extended USE specification that is created from the last section's transformation specification by our translation filter. As explained in Sect. 2.3, one _redex() and one _apply() operations are created in the added transformation class (Many2One). We pick the two operations that are created for the step *inlineRangeForRangeVar* because they are short enough and yet implement several of the features that can occur in a step. Following the order of execution, we first show the _redex() operation, which tries to find a valid tuple of objects for inlineRangeForRangeVar (slightly reformatted to improve readability).

```
Many2One::inlineRangeForRangeVar_redex() :
  Tuple(assocEnd:Property, navCallExp:NavigationCallExp,
        iterate:IterateExp, x:Variable, xref:VariableExp)
begin
  declare foundRedex : Boolean
  set foundRedex := false
  for iterate : IterateExp in IterateExp.allInstances do
    for xref : VariableExp in VariableExp.allInstances do
      if (not foundRedex) and
         xref.referredVariable = iterate.iterator and
         Set{self, self.endToChange, iterate.source,
              iterate, iterate.iterator, xref}->size = 6 and
         assocEnd.lower = '1' and assocEnd.upper = '1'
      then
        set result := Tuple{assocEnd = self.assocEnd,
                            navCallExp = iterate.source
                              .oclAsType(NavigationCallExp),
                            iterate = iterate,
                            x = iterate.iterator,
                            xref = xref}
        set foundRedex := true
      endif
    next
  next
end
```

This operation is basically a nested iteration over the potential matches for the step's context objects (lines 5 and 6). As an runtime optimization, the filter program tries to omit iterations for objects that implicitly reachable from other redex objects via to-1 navigations. For example, *navCallExp* is reachable as *iterate.source*. The heart of the loops checks for each combination of candidates if the non-implicit required links exist between them (line 8). If further all objects are pairwise different and not undefined (line 9) and all other preconditions hold (line 10), the found redex is returned from the operation.

Having a valid redex, we can invoke the _apply operation, which looks as follows for our step:

```
1  Many2One::inlineRangeForRangeVar_apply(redex :
     Tuple(assocEnd:Property, navCallExp:NavigationCallExp,
           iterate:IterateExp, x:Variable, xref:VariableExp))
2  begin
3      declare copyOfNavCallExp : NavigationCallExp
4      set copyOfNavCallExp := redex.navCallExp.copy()
         .oclAsType(NavigationCallExp)
5      redex.xref.owner().oclAsType(ModelElement)
         .replace(redex.xref, copyOfNavCallExp)
6      delete (redex.xref, redex.x) from VariableExp_referredVariable
7      destroy redex.xref
8  end
```

This operation actually applies the step. Beside the basic step semantics (creating/destroying links/objects, setting attribute values, cf. [BG06]), this step operation also realizes the «copy» and «replaceComponent» extensions (lines 4 and 5). Both extensions require additional operations to be generated into the UML/OCL meta-classes (copy() and owner()). Both additional operations can be generated automatically by exploiting the compositions (the black diamonds) in the meta-model.

There is a lot more to say about the translation of our transformation language into .use files which does not fit into this paper (for example, the translation of the control condition). However, we hope that we have illustrated the general idea of using an imperative OCL as an assembler for our transformation language.

## 4 Conclusion

In the previous sections we showed how we are realizing UML model transformations with USE. We extended the USE tool itself and added a new imperative language component. We then translated the higher level transformation language that we used to implement our catalog onto this imperative language. We feel that several transformation languages or formalism can be translated this way. Actually, we had to extend our initial transformation language by new elements («copy» and «replaceComponent») which underpins the flexibility of our approach.

Several alternatives exist for the imperative language extensions that we made. We believe that UML Actions (as part of the specification) could be used instead – actually, UML Actions provide a lot more features than we require for our approach. The Kermeta meta-modelling environment could also be used for this purpose, as it supports OCL evaluation. Kermeta also provides several programming language like extensions like exception handling.

We do not aim to provide a "better QVT". Instead, our approach aims towards a support for evaluation and development of both, transformations, and transformation languages. Because USE (technically) and OCL (conceptually) can be reused, new transformations concepts can be developed hands-on.

## References


[BG06] Fabian Büttner and Martin Gogolla. Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Gregorz Rozenberg, editors, *Proc. 3rd Int. Conf. Graph Transformations (ICGT'2006)*, pages 398–412. LNCS 4178, Springer, Berlin, 2006.

[Büt05] Fabian Büttner. Transformation-Based Structure Model Evolution. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS'2005 Conference*, pages 339–340. Springer, Berlin, LNCS 3844, 2005.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Report of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003.

[FDVF06] Franck Fleurey, Zoé Drey, Didier Vojtiseka, and Cyril Faucher. Kermeta language reference manual, 2006. http://www.kermeta.org/docs/KerMeta-Manual.pdf.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, August 1999.

[FUJ] The Fujaba tool suite, http://wwwcs.uni-paderborn.de/cs/fujaba/.

[GR02] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In Hassan Shafazand and A Min Tjoa, editors, *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, pages 228–238. Springer, Berlin, LNCS 2510, 2002.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[MDFH04] Pierre-Alain Muller, Cédric Dumoulin, Frédéric Fondement, and Michel Hassenforder. The topmodL initiative. In *3rd Workshop in Software Model Engineering (WISME UML 2004), 11 October 2004, Lisbon, Portugal, Proceedings of the UML Satellite Activities 2004, Lecture Notes in Computer Science, Volume 3297, Feb 2005*, pages 242–245, 2004.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[Mod] ModelWare, http://www.modelware-ist.org/.

[OMG06] OMG. MOF QVT final adopted specification, 2006.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.