On the Usage of Concrete Syntax in Model Transformation Rules

Thomas Baar[1] and Jon Whittle[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
[2] Department of Information and Software Engineering
George Mason University
Fairfax VA 22030 USA
thomas.baar@epfl.ch, jwhittle@ise.gmu.edu

Abstract. Graph transformations are one of the best known approaches for defining transformations in model-based software development. They are defined over the abstract syntax of source and target languages, described by metamodels. Since graph transformations are defined on the abstract syntax level, they can be hard to read and require an in-depth knowledge of the source and target metamodels. In this paper we investigate how graph transformations can be made much more compact and easier to read by using the concrete syntax of the source and target languages. We illustrate our approach by defining model refactorings.

Keywords: Metamodeling, Model Transformation, Refactoring, UML

1 Motivation

One of the key activities of model-based software development [1] is transformation between models. Model transformations are defined in order to bridge two different modeling languages (e.g., to transform UML sequence to UML communication diagrams) or to map between representations in the same language. A well-known example of the latter case is refactorings, i.e., transformations that aim at improving the structure of the source model [2, 3].

Model transformations can be expressed in many formalisms (see [4] for an overview) but graph transformation based approaches [5] are especially popular due to their expressive power. Also the recently adopted OMG standard "Query, Views, Transformations (QVT)" is based on this technique [6]. The problem tackled in this paper is that model transformations written in a pure graph transformation notation can easily become complex and hard to read.

A transformation written in QVT consists of a set of transformation rules. Each rule has a left-hand-side (LHS) and right-hand-side (RHS) which define the patterns for the transformation rule's source and target models. A rule is applied on a given, concrete source model by matching a sub-model of the concrete model with the LHS of the rule and replacing the matched sub-model with the RHS,

where any matchings are applied to the RHS before replacement. Additionally, all conditions imposed by the optional when-clause of the rule must be satisfied. The patterns defining the LHS and RHS are given in terms of the metamodels for the source and target modeling language (note that nowadays all major modeling languages, such as UML [7], are defined in the form of a metamodel). For the sake of simplicity in this paper (but our approach is not restricted to that), we will assume that the modeling languages for the source and target model coincide and thus each transformation rule refers only to the metamodel of one language.

A disadvantage of the graph transformation approach in defining model transformations is that the patterns LHS and RHS refer only to the abstract syntax of the modeling language and the more readable concrete syntax is not used in the transformation rule. Transformations written purely using abstract syntax are not very readable and require the reader to be familiar with the metamodel defining the abstract syntax. To overcome this problem, our approach is to write the transformation rules directly in the concrete syntax of the modeling language where possible. Unfortunately, this cannot be done directly since a number of subtleties of patterns in transformation rules have to be taken into account. In this paper, we make a distinction between the modeling language and the pattern language used to formulate the LHS and RHS. More precisely, we describe how the metamodel of the pattern language can be extracted from that of the modeling language. The extracted metamodel for the pattern language is then the basis to define a concrete syntax for patterns, that is similar to the concrete syntax of the original modeling language.

The rest of the paper is organized as follows. Section 2 gives some background information on defining modeling languages and model transformation techniques, with an emphasis on graph transformations. We show in Section 3 how to improve the readability of transformation rules by exploiting a concrete syntax derived from the source and target modeling language. Section 4 illustrates the strengths and some limitations of the approach by applying it to UML refactoring rules and Section 5 concludes the paper.

1.1 Related work

The authors know of no other work in using concrete syntax for graph-based model transformations. There is a good deal of research in applying graph transformations to software engineering problems — see [8] for an introduction — such as code generation, viewpoint merging and consistency analysis. However, in all applications we have seen, the transformation rules are based on the abstract syntax of the source and target modeling languages.

Approaches addressing issues related to concrete syntax and transformations have been focused somewhat differently than our work. Papers on tool support for model transformations, e.g. [9], have discussed the problem of synchronizing a model and its visual representation after a transformation has been executed. One approach is to extend the metamodels of the modeling languages with a metamodel for the visual representation of models (i.e., the concrete syntax) and to formulate the transformation rules based on this extended metamodel.

2 Defining Model Transformations

2.1 Metamodeling

A modeling language has three parts: (1) the abstract syntax that identifies the concepts of the language and their relationships, (2) the concrete syntax that defines how the concepts are represented in a human-readable format, and (3) the semantics of the concepts. This paper is only concerned with (1) and (2).

The abstract syntax of a modeling language is usually defined in the form of a metamodel. A metamodel is usually described by a (simplified form of a) UML class diagram [7] with OCL [10] invariants. The concepts of the language are defined by classes in the metamodel (i.e., meta-classes). Concept features are given as meta-attributes on meta-classes and relationships between concepts are given by meta-associations.

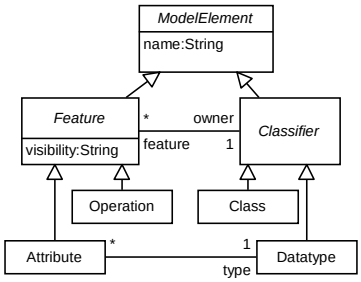
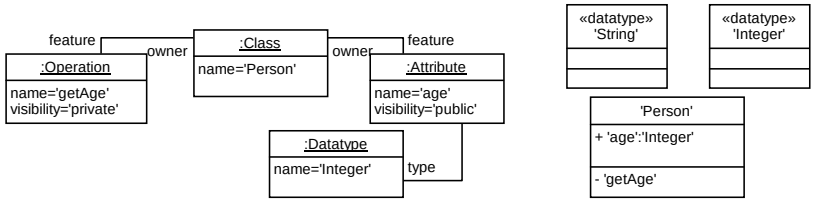


Fig. 1. Metamodel of simplified class diagrams, called *CDSimp*

Figure 1 shows the metamodel of a drastically simplified version of UML class diagrams, called *CDSimp*. The language *CDSimp* will serve as a running example in the remainder of this paper. The metamodel for *CDSimp* consists of metaclasses that correspond directly to concrete model elements, namely `Attribute`, `Operation`, `Class` and `Datatype`, as well as abstract metaclasses that do not have a concrete syntax representation but are introduced for structuring purposes: `ModelElement`, `Feature`, `Classifier`. For instance, the metaclass `ModelElement` declares a metaattribute `name` of type `String` that is inherited by all other metaclasses.

OCL invariants attached to the metamodel impose restrictions that every well-formed model must obey (thus, the invariants are also called *well-formedness rules*). Two invariants are relevant for the examples presented later in this paper. The first invariant says that the names of all features in a class or datatype are pairwise different and the second invariant restricts the values for visibility:

```
context Classifier inv UniqueFeatureName:
  self.feature->forAll(f1,f2| f1.name=f2.name implies f1=f2)
```



(a) Instantiation of metamodel

(b) Graphical notation using concrete syntax

Fig. 2. Two representations of the same class diagram

```
context Feature inv VisibilityDomain:
    self.visibility = 'public' or
    self.visibility = 'private' or
    self.visibility = 'protected'
```

Considering only the abstract syntax of a modeling language, one can say that a *model* written in this modeling language is just an instance of the language's metamodel that obeys the two given well-formedness rules (i.e. the two OCL invariants *UniqueFeatureName* and *VisibilityDomain* are evaluated in every model to true). A model can be depicted as an object diagram (cf. Figure 2(a)) but this is not very readable, because all concrete model concepts are reified as metaclasses in the metamodel. More readable for humans is a graphical representation of the same model that takes the *concrete syntax* of the language into account (cf. Figure 2(b)). The concrete syntax for *CDSimp* resembles that of UML class diagrams. The only difference is that string literals, such as the name of a class or attribute, are given in quoted form (e.g. 'Person' instead of Person). We will need this convention later on.

2.2 Concrete syntax definition

The concrete syntax of a language can be defined as a mapping from all possible instances of the language's metamodel into a representation format (in most cases, a visual language [11]). Despite the recent progress that has been achieved in formalizing diagrams (see, for example, OMG's proposal for Diagram Interchange [12] as an attempt to standardize all graphical elements that can possibly occur in diagrams), it is still current practice to define the concrete syntax of a modeling language informally. For the sake of brevity, we also give an informal definition here, but, as shown in [13], a conversion into a formal definition can be done straightforwardly. The language *CDSimp* has the following concrete syntax definition:

- Classes and datatypes are represented by rectangles with three compartments.
- The first compartment contains the name of the class/datatype. The name of datatypes is stereotyped with `<<datatype>>`.

– The remaining compartments contain the representation of all owned features (attributes are shown in the second, operations in the third compartment). A feature is represented by a line of the form:
visiRepr ' ' *name* [':' *type*]
where *visiRepr* is a representation of the feature's visibility ('+' for 'public', '-' for 'private', '#' for 'protected'), *name* is the actual name of the feature, and, in case of an attribute, *type* is the name of the attribute's type. Note that both visibility and name are mandatory parts of a feature representation.

Concrete syntax definitions are needed only for those concepts that are reified in a concrete model. For example, the abstract metaclass `Feature` does not have a concrete syntax definition.

2.3 Model transformations

The exact format and semantics of model transformation rules is fully described in [6]. In this paper, we consider only the format of the patterns LHS and RHS in each transformation rule, and the relationship of LHS and RHS to the optional when-clause of the rule.

A pattern can be defined as a more general form of object diagram in which all objects are labeled by a unique variable with the same type as the object. Variables are also used in order to represent concrete values in objects for attributes. Unlike usual object diagrams, objects of abstract classes (e.g. `Classifier`) can occur in patterns.

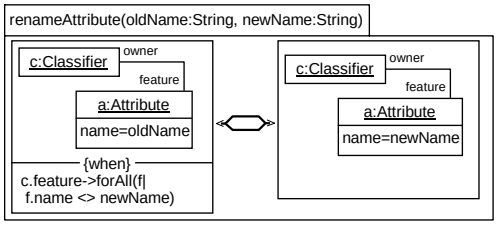


Fig. 3. QVT rule to rename an attribute within classifiers

Figure 3 shows an example transformation for renaming an attribute of a classifier. The pattern LHS specifies the subgraphs to be matched in the source model when the rule is applied. The LHS consists of two objects of type `Classifier` and `Attribute`, respectively, labeled with variables *c* and *a*, which are connected by a link instance of the owner-feature association. The value for metaattribute `name` in object *a* is the same as the value of the rule parameter *oldName* of type `String`. In addition, the when-clause requires object *c* to have no feature with name *newName*. The pattern RHS is identical to LHS with the exception

that variable *oldName* is substituted by *newName*. Informally, the application of `renameAttribute` on a concrete source model involves the following steps: (1) find a classifier (i.e., since metaclass `Classifier` is abstract, a class or datatype) in the source model that has an attribute with name *oldName* (matching the LHS) but no feature with name *newName* (checking the when-clause) and then (2) rename the matching attribute from *oldName* to *newName* and do not make any other change in the model. These steps would be applied iteratively as often as possible. Note that the when-clause implicitly imposes the constraint that *newName* is different than *oldName*. This ensures termination of the rule application. The when-clause also ensures syntactical correctness of the target model. For example, it ensures that the well-formedness rule *UniqueFeatureName* is satisfied in the target.

3 Patterns In Concrete Syntax (PICS)

Graph transformation rules, such as those given by QVT and described in Section 2, are a very powerful mechanism to describe model transformations. Readability, however, can become a serious problem if the patterns LHS and RHS are given in object diagram syntax. The main idea of our approach is to alleviate this problem by exploiting the concrete syntax of the language whose models we want to transform. Unfortunately, we cannot apply the concrete syntax of the modeling language directly for the rendering of patterns because some important information of the pattern would be lost. We will, thus, first analyze the differences between a modeling language and the corresponding pattern language used in transformation rules. Then, the pattern language is defined by its own metamodel, which is, as shown in Section 3.2, a straightforward modification of the original metamodel for the modeling language. Based on the modified metamodel, we finally define a concrete syntax for the pattern language, which is called *PICS* (patterns in concrete syntax). The term *PICS metamodel* refers to the metamodel of the pattern language that has been derived from the metamodel of the modeling language.

3.1 Differences between models and patterns

For defining a concrete syntax for pattern diagrams the following list of differences between models (seen as instances of the modeling language's metamodel) and patterns used in transformation rules has to be taken into account:

1. **Objects in patterns must be labeled**[3] with a unique variable (e.g. the label for `c:Class` is *c*).

[3] In many graph transformation systems including QVT the label is optional. We assume here the strict version since it will make it easier to rewrite a pattern using the concrete syntax.

2. **A pattern usually represents an incomplete model** whereas object diagrams are assumed to be complete, i.e., all well-formedness rules and multiplicities of the metamodel are obeyed. For example, the patterns LHS, RHS in `renameAttribute` (Figure 3) show neither the attribute `visibility` of object *a:Attribute* nor a link to its type (an object diagram could not drop this link due to multiplicity 1 of the corresponding association end at `Datatype`).
3. **Patterns can have objects whose type is an abstract class** whereas the type of objects in object diagrams is always a non-abstract class.
4. **Patterns can contain variables to represent attribute values in objects** whereas in object diagrams such values are always literals or ground-terms.

A pattern language is, due to these differences, more expressive than the language of object diagrams since each object diagram is also a pattern but not vice versa. Note, however, that the last difference is only a minor one. Variables for attribute values could easily be integrated into object diagrams as well if the value of attribute slots are always displayed according to some simple rules: (i) literals of type `String` must be enclosed by quotes and (ii) literals of all other types have to be pre-defined. If, under these conditions, a term occurs in an attribute slot that is neither a composed term nor a literal of type `String` or any other type, then this term would denote a variable.

3.2 Transforming the original metamodel to PICS metamodel

The important differences between models and patterns (points (1) – (3) above) can be formalized by defining a metamodel for pattern diagrams. Fortunately, this metamodel can be automatically derived from the original metamodel by applying the following changes:

- Add attribute `label:String` with standard multiplicity 1..1 to each metaclass. This change captures the mandatory labels of objects in pattern diagrams (see difference (1) in above list of differences).
- Make all attributes in the metamodel optional (by giving them the attribute multiplicity 0..1) and change all association multiplicities from y..x to 0..x. Both changes reflect incompleteness of patterns (see difference (2)).
- Make all abstract classes non-abstract (see difference (3)).

Figure 4 shows the changes on the metamodel for *CDSimp*. The root class `ModelElement` has a new attribute `label` that is inherited by all other classes. The two other attributes `name` and `visibility` became optional by the attribute multiplicity 0..1. The abstract classes `ModelElement`, `Feature`, `Classifier` became non-abstract and finally all multiplicities on association ends were changed to the range 0..OrigMultiplicity (note that multiplicity * is not affected).

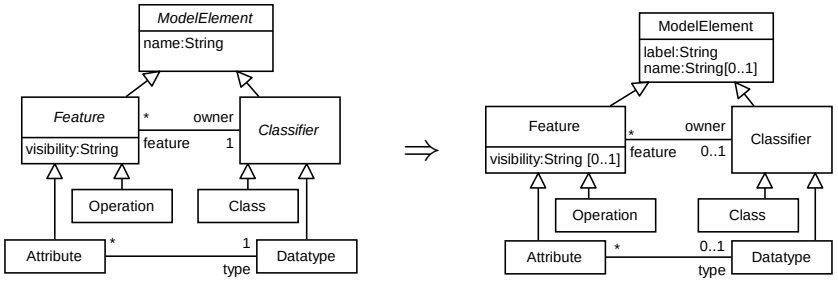
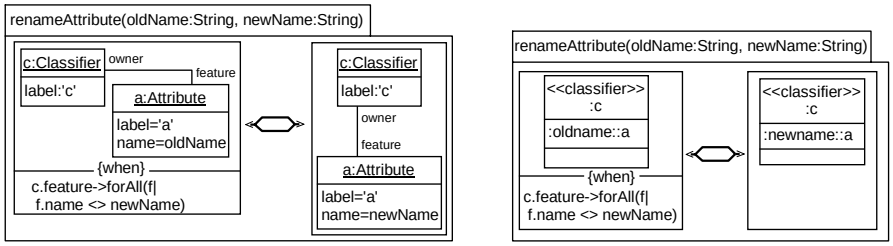


Fig. 4. Original language metamodel and derived PICS metamodel



(a) Rule as instance of PICS metamodel

(b) After applying PICS concrete syntax

Fig. 5. Rule `renameAttribute` as instance of PICS metamodel and in concrete syntax

3.3 Defining concrete syntax for PICS metamodel

After the pattern language has been formalized as the PICS metamodel, we can represent each pattern as an instance of the PICS metamodel. Figure 5(a) shows the transformation rule `renameAttribute` as an example. Please note that Figure 5(a) is just another representation of the original definition given in Figure 3 and conveys exactly the same information. Hence, each representation equivalent to Figure 5(a) is also equivalent to the original definition of the transformation rule.

Defining an equivalent representation for the instances of a metamodel is traditionally done by defining a concrete syntax for the metamodel. For our running example, a concrete syntax for the PICS metamodel shown in Figure 4 could be defined by modifying the concrete syntax for *CDSimp* as follows:

- Instead of the *name*, the first compartment of classes/datatypes shows a line of the form *name* ':' *label* where *name* denotes the value of the optional attribute `name` and *label* the value of the mandatory attribute `label`. Since *name* appears only optionally, a delimiter ':' between *name* and *label* is needed in order to ensure correct parsing. The delimiter must not occur in *name* and *label*.
- An attribute/operation having an owning classifier is shown by a text line in the second/third compartment of the owning classifier. The only difference

to the concrete syntax of *CDSimp* is the usage of delimiter ':' to separate the line items (in order to handle optional occurrences) and that the label of the attribute/operation is added at the end of the line.
In other words, the line has the form [*visiRepr*] ':' [*name*] [':' *type*] ':' *label*
If an attribute/operation does not have an owning classifier (note the multiplicity 0..1 for the association between `Feature` and `Classifier` in the PICS metamodel) then the text line is shown outside any other classifier in a box.
- Instances of `Classifier` are rendered the same as classes/datatypes except that they have a stereotype `<<classifier>>` in the first compartment.
- Features (instances of `Feature`) are rendered the same way as attributes/ operations but, in order to distinguish them, they have to be marked as features. This could be done, for example, by preceding the text line with 'f:'.
- Instances of `ModelElement` are rendered by a one-compartment rectangle labeled with *name* ':' *label*.

The first two items explain how to adapt the renderings of metaclasses that are non-abstract both in the original metamodel of the modeling language *CDSimp* and in the PICS metamodel. The rendering in PICS is very similar to that in *CDSimp*. Merely the label of the object had to be added and a delimiter was introduced to identify the position of an element in a text line. The remaining items explain the rendering of metaclasses that were abstract in the original metamodel but became non-abstract in PICS. Since no rendering of these classes was defined for *CDSimp*, the new renderings for the PICS metamodel had to be invented. An application of the PICS concrete syntax is shown in Figure 5(b) for `renameAttribute`.

To summarize so far, we have defined the abstract syntax (using a metamodel) of the pattern language for defining patterns in the LHS and RHS of a graph transformation rule. Furthermore, we have shown on an example how to define concrete syntax for this pattern language based on the syntax of the associated modeling language.

3.4 Finding a good concrete syntax for the pattern language

Although it is always possible to define a concrete syntax for the PICS metamodel (note that showing the instance of the metamodel just as an object diagram – see Figure 5(a) for an example – would be a trivial version of a concrete syntax) it is usually a challenge to find a non-ambiguous concrete syntax that is still similar to the concrete syntax of the modeling language whose models are being transformed. The definition of a good concrete syntax is of primary importance for the readability and understandability of the transformation rules written in PICS syntax.

There are basically two problems to tackle: (1) Handling of optional occurrences of attribute and links and (2) rendering of classes that were abstract in the metamodel of the modeling language.

The first problem was tackled in the above *CDSimp* example by using delimiters that allow to infer for a rendered object which of its attributes are rendered and which not. This technique, however, needs the assumption that the symbol used as delimiter (here ':') is not used otherwise in order to avoid ambiguity of the representation. Some initial tool support for detecting ambiguities in a concrete syntax definition is described in [14].

In order to solve the second problem, new icons/symbols have to be invented which raises the issue of similarity between the original modeling language and the pattern language. For some classes, e.g. `Feature` and `Classifier`, a suitable rendering can be defined as a straightforward generalization of the renderings of the subclasses. For other classes, e.g. `ModelElement`, this heuristic does not work just because the renderings of the subclasses are too diverse.

As future work, we plan to investigate pattern languages that allow mixing of abstract and concrete syntax. This could be done by leaving the concrete syntax definition for the pattern language incomplete. This would mean that some parts of patterns do not have a rendering in the concrete syntax, and would be done when there exists no rendering that is similar to the modeling language. For example, if a rule needs to refer to an abstract metaclass that has no rendering, we would allow this abstract metaclass to be referenced in the pattern definition. In essence, this allows patterns to mix concrete and abstract syntax. A mechanism would be required to control whether an element is concrete or abstract but this could be done, for example, in a similar way to the quote/anti-quote mechanism in LISP.

4 Case Study: UML Refactoring Rules in PICS Notation

In [15], a number of refactoring rules for UML class diagrams using the abstract syntax of class diagrams has been defined. We present, in the following, a rewriting of these refactoring rules[4] using our PICS approach.

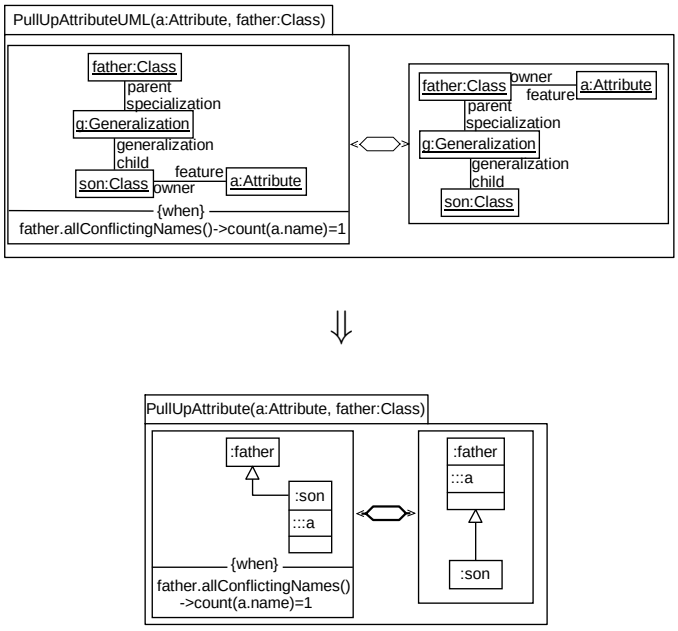
The refactoring rules are written with respect to the metamodel for UML 1.5 (see [15] for the relevant part of the metamodel). The refactoring rules are designed to preserve an important well-formedness rule of the UML 1.5 metamodel, namely that names for attributes, opposite association ends and owned elements are unique within each classifier and all its parents along the generalization hierarchy. In order to ensure this well-formedness rule, most refactoring rules have a when-clause that uses the following additional operation:

```
context Classifier def: allConflictingNames():Bag(String)=
    self.allParents()->including(self)
   ->union(self.allChildren())
   ->iterate(c; acc:Bag(String)=Bag{}|
        acc->union(c.oppositeAssociationEnds().name)
           ->union(c.attributes().name)
           ->union(c.ownedElement.name))
```

[4] More precisely, some improved variants of the rules given in [15] are taken here as a starting point.

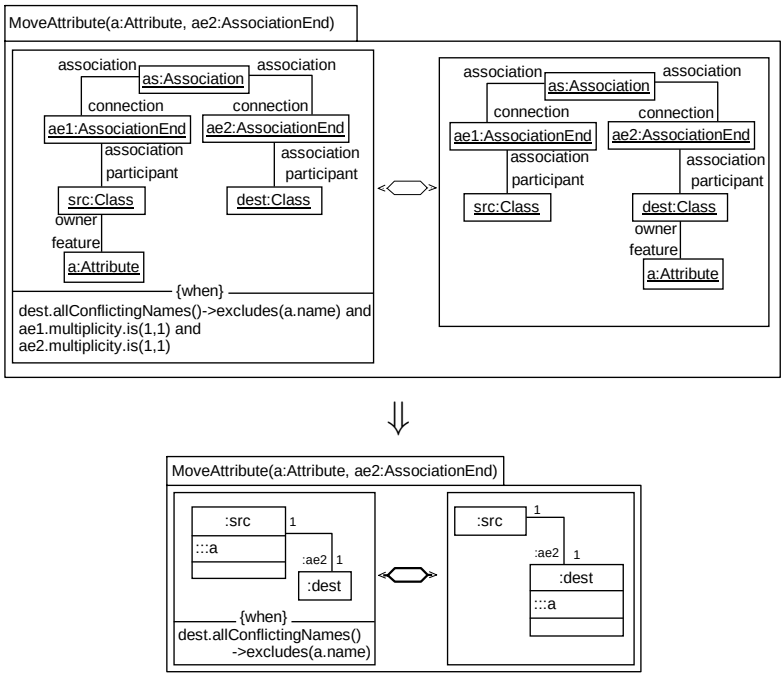
4.1 PullUpAttribute

The rule *PullUpAttribute* moves an attribute from a child class to a parent class. It can be rewritten straightforwardly using the same PICS syntax that has been used for *CDSimp*. Please note that we do not rewrite the when-clause. We show below the *PullUpAttribute* both in its original form and the rewritten version (all the examples will be presented in a similar fashion).



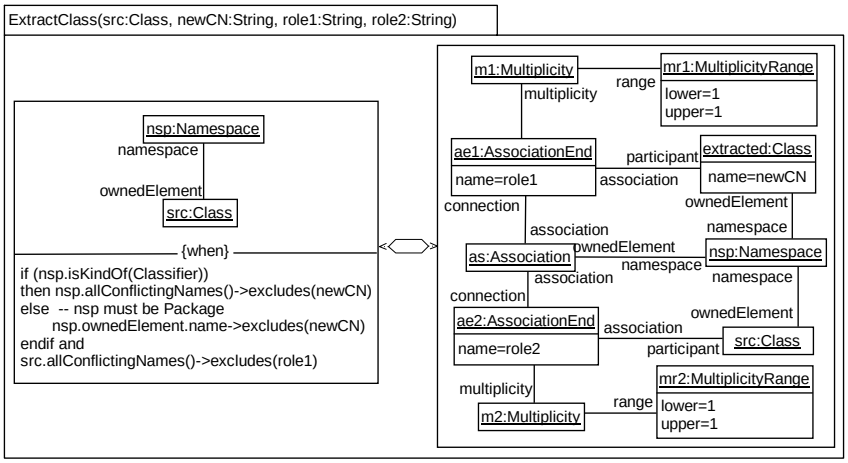
4.2 MoveAttribute

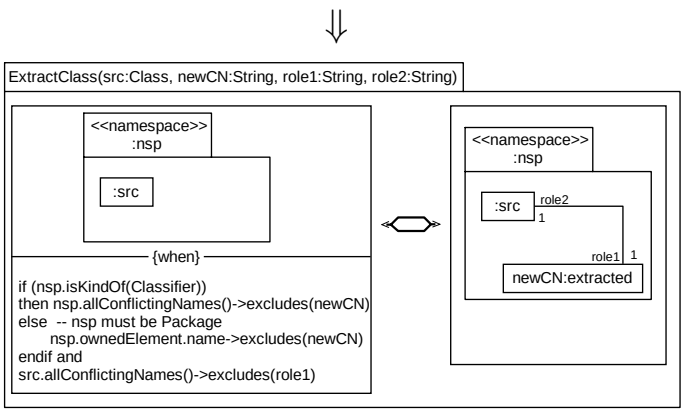
The refactoring *MoveAttribute* moves an attribute from a class on one end of an association to a class on the other end of the association. In the rewritten version, the when-clause is modified. The part of the when-clause stipulating the connecting association to be of multiplicity 1-1 is in the rewritten version rendered by an annotation 1 on the association ends, which is a standard technique in UML. More complicated forms of OCL constraints could be represented in a visual form. We do not discuss this topic here in-depth but refer the interested reader to [16] where graphical notations are defined as abbreviations for complex OCL expressions.



4.3 ExtractClass

The refactoring *ExtractClass* creates for a given class *src* a new class and connects it with *src* by an association with multiplicity 1-1. Furthermore, the created class and association should be placed in the same namespace as class *src*. Please note that the metaclass `Namespace` is abstract in the UML 1.5 metamodel. For this reason, a new rendering for `Namespace` has to be invented for the PICS syntax. Here, a package icon stereotyped with `<<namespace>>` has been chosen.





5 Conclusion and Future Work

This paper addressed how to define model transformation rules in a more readable way by using the concrete syntax of source and target modeling languages when defining the LHS and RHS of the rules. The concrete syntax, however, had to be adapted to the peculiarities of patterns, mainly mandatory labeling of objects and optional occurrence of attributes and links. Another major problem is that the PICS concrete syntax has to invent a new rendering for metaclasses that were abstract in the original metamodel. An alternative, that has been only sketched in this paper, is to show these metaclasses in the abstract syntax notation, that is to allow mixing concrete and abstract syntax presentations within transformation rules. If an intuitive concrete syntax for patterns is found, then transformation rules can be presented in the same way as models of the source/target languages.

We have not addressed in this paper how the when-clause of transformations rules can be rendered in graphical form as well. There is a standard technique in graph-transformation literature how negative constraints can be made visible (known as non-application conditions (NACs)). Our approach could also be extended by the work of Stein, Hanenberg and Unland presented in [16] where visualizations of OCL constraints for the domain of metamodel navigation is discussed.

Acknowledgements The authors would like to thank the anonymous reviewers for their very helpful comments on the initial version of this paper.

References

1. Stuart Kent. Model driven engineering. In *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
2. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

3. Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
4. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proc. OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
5. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
6. OMG. MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01, Nov 2005.
7. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, second edition, 2005.
8. Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In A. Corradini, H. Ehrig, and H.-J. Kreowski und G. Rozenberg, editors, *First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 402–429. Springer, 2002.
9. Esther Guerra and Juan de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, pages 54–69, 2004.
10. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
11. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
12. OMG. Unified Modeling Language: Diagram interchange version 2.0. Convenience Document ptc/05-06-04, June 2005.
13. Frédéric Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In *Proc. European Conference on Model Driven Architecture (ECMDA-FA)*, volume 3748 of *LNCS*, pages 190–204. Springer, 2005.
14. Thomas Baar. Correctly defined concrete syntax for visual models. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings, MoDELS/UML 2006, Genova, Italy*, volume 4199 of *LNCS*, pages 111–125. Springer, October 2006.
15. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
16. Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query models. In *Proc. IEEE 7th International Conference on the Unified Modeling Language (UML 2004)*, volume 3273 of *LNCS*, pages 98–112. Springer, 2004.