Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

# A Catalogue of Refactorings for Model-to-Model Transformations

Manuel Wimmer[a] Salvador Martínez[b] Frédéric Jouault[b]
Jordi Cabot[b]

a. Vienna University of Technology, Austria
b. INRIA & Ecole des Mines de Nantes, France

Abstract In object-oriented programming, continuous refactorings are used as the main mechanism to increase the maintainability of the code base. Unfortunately, in the field of model transformations, such refactoring support is so far missing. This paper tackles this limitation by adapting the notion of refactorings to model-to-model (M2M) transformations. In particular, we present a dedicated catalogue of refactorings for improving the quality of M2M transformations. The refactorings have been explored by analyzing existing transformation examples defined in ATL. However, the refactorings are not specifically tailored to ATL, but applicable also to other M2M transformation languages.

Keywords Refactoring, Model Transformation, Model Transformation Quality

## 1 Introduction

Model manipulation is a central activity in many model-based software engineering activities [SK03] like, model translations (e.g., translating a UML class model into an ER model), model augmentations (e.g., weaving aspects into a UML class model), and model alignments (e.g., mapping a content model to its GUI view), to mention just a few. Model manipulations are usually implemented by means of model-to-model (M2M) transformations. A M2M transformation transforms a model $Ma$ conforming to a metamodel $MMa$ into a model $Mb$ conforming to a metamodel $MMb$ (where $MMa$ and $MMb$ can be the same or different metamodels).

Current research on model transformation focuses on developing languages for specifying transformations (e.g., cf. [CH06] for a survey). However, there are no available techniques focusing on the maintainability of existing transformations beyond manually applying atomic edit operations on them. Such support is clearly needed, e.g., to improve the readability of transformations and to facilitate their evolution in response to changes on the transformation requirements and/or the source/target

Manuel Wimmer, Salvador Martínez, Frédéric Jouault, Jordi Cabot. *A Catalogue of Refactorings for Model-to-Model Transformations*. In Journal of Object Technology, vol. 11, no. 2, 2012, pages 2:1–40.
doi:10.5381/jot.2012.11.2.a2

metamodels used in the transformation. A maintainable, reusable, and extensible set of transformation definitions is a key aspect in any high-quality model-based solution.

In the area of object-oriented programming, refactorings are the technique of choice for improving the structure of existing code without changing its external behavior [Opd92, Fow99, MT04]. They have proved to be useful to improve the quality attributes of source code, and thus, to increase its maintainability. Unfortunately, no catalogue of refactorings for model transformation exists. Available object-oriented refactoring catalogues are not reusable as they are, because most current transformation approaches follow a rule-based programming paradigm and are very domain-specific. This forces transformation developers to improve model transformation specifications without any kind of dedicated support. Given the potential complexity of model transformations, manual modifications may lead to unwanted side-effects and result in a tedious and error-prone maintenance process.

In this sense, the main contribution of this paper is the provision of a refactoring catalogue for rule-based M2M transformations. The catalogue is based on our experience as MDE (model-driven engineering) researchers plus on the analysis of existing transformation examples from different sources[1] defined in ATL [JK05]. Most of the refactorings are not specifically tailored to ATL, but are applicable also for other M2M transformation languages following the rule-based paradigm such as the QVT transformation language family [OMG11]. It is worth to note that the presented refactorings may improve not only quality attributes related to maintainability such as readability, reusability, and extensibility of the transformations, but also the performance of transformations. Execution of refactorings can be semi-automated by employing higher-order transformations [TJF+09].

The rest of the paper is structured as follows. In Section 2, we introduce the main M2M transformation concepts and present an illustrative example which exhibits some bad smells that should be eliminated. Section 3 presents the notion of refactorings for M2M transformations and Section 4 summarizes the refactoring catalogue and its application on excerpts of the illustrative example. Furthermore, this section also highlights the reusability of the refactorings for other rule-based transformation languages. Section 5 and Section 6 show the impact of the refactorings on internal quality attributes and on the execution performance of transformations, respectively. In Section 7 we report on some details of the implementation of the refactorings. Section 8 discusses related work, and finally, Section 9 concludes with an outlook on future work.

## 2 Illustrative example

We illustrate M2M transformations and motivate the need for transformation refactorings by means of the transformation scenario presented in this section and used as a running example throughout the paper. The goal of this transformation scenario is to transform UML class diagrams into Entity Relationship (ER) diagrams.
Fig. 1 shows the (simplified) metamodels of both modeling languages. Most modeling concepts have a direct counterpart in the other language except for the inheritance concept in UML, which can not be represented in our simplified version of the ER language. Thus, an important task of the transformation is to flatten inheritance trees in the UML model, duplicating the properties of the superclasses in the subclasses when generating entity types in the ER model.

[1] For instance, the transformations available at `www.eclipse.org/m2m/atl/atlTransformations`

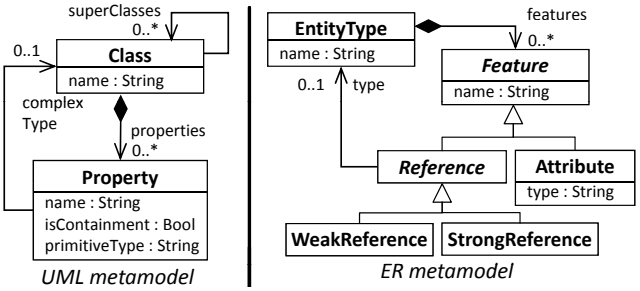


Figure 1 – Metamodels of the UML Class Diagram and Entity Relationship Diagram.

```
1  module UML2ER;
2  create OUT : ER from IN : UML;
3
4  helper context UML!Class def: allClasses() : Sequence(UML!Class) =
5      self.superClasses->iterate(e; acc : Sequence(UML!Class) = Sequence {} |
6      acc->union(Set{e})->union(e.allClasses()) );
7
8  rule Class {
9    from
10     s: UML!Class
11   to
12     t: ER!EntityType (
13       name <- s.name,
14       features <- attributes,
15       features <- weakReferences,
16       features <- strongReferences
17     ),
18     attributes : distinct ER!Attribute foreach(a in
19         s.allClasses().including(s).flatten()
20         ->collect(e | e.ownedProperty).flatten()
21         ->select (e | not e.primitiveType.oclIsUndefined())) (
22         name <- a.name,
23         type <- a.primitiveType
24     ),
25     weakReferences : distinct ER!WeakReference foreach(a in
26         s.allClasses().including(s).flatten()
27         ->collect(e | e.ownedProperty).flatten()
28         ->select(e | not e.complexType.oclIsUndefined() and not e.isContainment)) (
29         name <- a.name,
30         type <- a.complexType
31     ),
32     strongReferences : distinct ER!StrongReference foreach(a in
33         s.allClasses().including(s).flatten()
34         ->collect(e | e.ownedProperty).flatten()
35         ->select(e | not e.complexType.oclIsUndefined() and e.isContainment)) (
36         name <- a.name,
37         type <- a.complexType
38     )
39 }
```

Listing 1 – UML to ER Transformation in ATL.

We have chosen ATL as the transformation language for this paper, because it is one of the most widely used transformation languages, both in academia and industry, and there is mature tool support available. List. 1 shows the UML2ER transformation expressed in ATL. This (correct but poor as discussed below) transformation is a typical solution we frequently encounter in our model engineering labs [BKSW09] where about 150 students have to solve several model transformation problems with ATL.

An ATL transformation is composed of a set of transformation rules and helpers. Each rule describes how (part of) the target model should be generated from (part

of) the source model. There are two kinds of declarative rules, *matched* rules and *lazy* rules. The former are automatically matched by the ATL execution engine according to the rule matching pattern, whereas the latter have to be explicitly called from another rule giving more control over the transformation execution. This is similar to the concept of top and non-top relations in QVT Relations.

A helper can be seen as auxiliary function that enables the possibility of factorizing ATL code used in different points of the transformation. In the transformation example, the rule *Class* matches every class in the UML model to produce an entity type in the output model while the helper *allClasses()* calculates all direct and indirect superclasses of a given class.

Rules are mainly composed of an *input* pattern and an *output* pattern. The input pattern filters the subset of source model elements that are concerned by the rule. The output pattern details how the target model elements are created from the input ones. Each output pattern element can have several bindings that can be used to initialize the values of the elements in the target model. These initializations are performed as a second step after a first step consisting in matching the rules and creating the output elements. This separation in two steps enables the utilization of an automatic resolution algorithm allowing the initialization of target values with other target values by indicating the source values that will produce them. This avoids the need for navigating the target model.

In the example, we have defined one input pattern element, that selects elements of type *class*, and an output pattern that creates four types of elements: *entity types*, *attributes* and both kinds of *references*. Bindings are used, for instance, to initialize the name of entity types with the name of the corresponding classes. Distinct-foreach clauses in the pattern indicate that this part of the pattern can produce more than one output element of that type at once.

Finally, the Object Constraint Language (OCL) is used all along ATL transformations as a query language for traversing the models.

Although the previous transformation does the job, i.e., it correctly produces ER models from UML ones, it has several bad smells that compromise its quality in terms of maintainability and performance:

1. The transformation consists of one complex rule doing all the work, instead of decomposing the transformation based on the different types of elements in the source pattern. This bad smell may be seen as the rule-based paradigm equivalent to the well-known "God Class" bad smell in object-oriented programming. Thus, we name this bad smell "God Rule".

2. Duplicated code hampers evolution. For instance, if the reference *ownedProperty* is renamed in the source metamodel, three complex OCL expressions have to be adapted in the transformation.

3. Unnecessary repetitive calls that compromise the performance. The helper *allClasses()* is called several times for the same element. This results in recalculating the return value every time.

4. Use of deprecated constructs. The distinct-foreach construct used in the transformation has been classified as deprecated because it breaks the internal traceability links of the ATL virtual machine.

Transformation designers may not be aware of these problems or are unsure of how to solve them without breaking the transformation. The refactoring catalogue

we present in the next section clearly improves this situation by contributing to the body of knowledge of transformation engineering.

## 3 Refactorings for M2M Transformations

In this section, we explain how the notion of refactoring is adopted to the field of M2M transformations.

As for any other kind of refactorings, the behavior of the refactored object, the transformation in our case, must be preserved. Unfortunately, there is no consensus about the meaning of behavior preservation of refactorings. Therefore a universal definition of behaviour preservation is missing [MT04]. Consequently, several definitions exist tailored to specific domains and languages.

Adapting the broadest definition [Opd92] to M2M transformations leads to the following definition. Behaviour preservation is assured if for any input model, the produced output model will be the same before and after refactoring the transformation. This definition of behavior preservation may be checked by a comprehensive test suite or in case of having formal semantics by demonstrating the semantic equivalence of the refactored transformation with the original one. There are some efforts to give a formal semantics to transformation languages like ATL, cf. e.g., [TV11], but there is not yet a complete formal semantics for any of them. Thus, due to the rapid feedback and its pragmatic advantage which has been proven useful for software engineers in the field of object-oriented refactorings, we use the test suite approach for checking the behaviour preservation of our refactorings. In particular, by using model comparison frameworks such as EMF Compare[2], we are able to validate that for a set of test input models, the same output models are generated by the initial transformation and the refactored ones. At our project website[3], we provide transformations and test input models we have used to validate the refactorings of our catalogue.

## 4 Catalogue

This section describes our proposed refactoring catalogue. For outlining the refactorings in the paper, we are using a format inspired by Fowler [Fow99]. In particular, we describe the refactorings by (1) giving a name to the refactoring, (2) describing the typical situation where the refactoring should be applied, i.e., the problem, (3) describing the solution to improve the problematic situation, (4) stating the preconditions that must be satisfied to be able to apply the refactoring, (5) stating the parameters needed to provide the necessary information to execute the refactoring, (6) describing the refactoring steps and, for some of them, (7) discussing a concrete example application. Implementation of the proposed refactorings is described in Section 7.

The refactorings mainly focus on language constructs of ATL that are also part of other M2M transformation languages. This includes also the notion of inheritance between transformation rules [WKK+11], which allows to reuse and extend existing transformation rules defined for superclasses for their subclasses. Furthermore, OCL constructs forming also an integral part of many transformation languages are also

[2] `http://www.eclipse.org/emf/compare`

[3] `http://www.emn.fr/z-info/atlanmod/index.php/Model-to-Model_Transformation_Refactorings`

covered by our refactoring catalogue. Thanks to this, most of the refactorings are useful regardless the concrete M2M transformation language of use. We do not attempt to cover imperative code in the current version of the refactoring catalogue. This may be already covered by existing refactorings tackling the improvement of imperative code in the field of object-oriented programming languages [Opd92, Fow99].

The refactorings are structured into four categories:

1. *Renaming*: This category comprises refactorings needed for renaming identifiers as well as their references within the transformation.

2. *Restructuring*: Transformations are composed by rules for generating output elements from input elements and by additional helpers for calculating certain values. Thus, refactorings are needed for improving the structure of a transformation by means of restructuring and introducing rules and helpers.

3. *Inheritance-related*: The refactorings from this category deal with the extraction or elimination of commonalities between rules by introducing and removing inheritance between rules.

4. *OCL Expression Optimization*: Finally, transformation languages are built upon OCL for queries and computation of feature values. Thus, also the improvement of OCL expressions should be possible through the execution of refactorings.

While the first category can be seen as a set of basic refactorings for enhancing the readability of the transformation, category 2-4 is used for heavily changing the structure of a transformation. In particular, we also included refactorings for explicitly removing outdated language constructs and poor coding practices from existing transformations as well as introducing new language constructs such as rule inheritance into already existing transformations. Table 1 summarizes the refactoring catalogue. In the following subsections, we outline each category in more detail.

## 4.1 Renaming Refactorings

As in programming languages [Fow99], one of the easiest but nonetheless very useful things one can do to improve code is simply changing names. In transformation languages, rules and helpers with proper names will give a precise idea of what functionality they are providing, thus saving the time needed to go to the definition itself. Other language constructs such as variables for input/output pattern elements may be renamed as well. In Table 1, refactorings 1 to 3 perform these renaming operations. In the rest of this subsection we provide a short description of each of these refactorings.

**(1) Rename In/Out Pattern Element**
*Problem*: In/Out pattern element name is not descriptive enough.
*Solution*: Substitute the name for a more meaningful one. Normally, associated with source/target meta-element names.
*Preconditions*: There is not an In/Out pattern element with the same name for the given rule.
*Parameters*:
1. The pattern element to be renamed.
2. The new name for the In/Out pattern element.

| N | Name of the refactoring | Purpose |
|---|---|---|
| | **Renaming** | |
| 1 | *Rename In/Out Pattern Element* | Change the name of In/Out Pattern Element for better explaining the intention of the element. |
| 2 | *Rename In/Out Model* | Change the name of In/Out Models for better explaining the intention of the model. |
| 3 | *Rename Rule/Helper* | Change the name of Rules/Helpers for better explaining the intention of the Rule/Helper. |
| | **Restructuring** | |
| 4 | *Extract Helper/Rule** | Extract an additional helper from an existing helper/rule or extract an additional rule from an existing rule. |
| 5 | *Inline Helper/Rule** | Inline a helper into another helper/rule or inline a rule into another rule. |
| 6 | *Merge Rule* | Merge rules into one rule |
| 7 | *Split Rule* | Split a rule into two rules |
| 8 | *Merge Binding* | Merge two bindings into one binding if the same target feature is set. |
| 9 | *Split Binding* | Split a binding into two bindings if several elements are assigned to the one feature. |
| 10 | *Convert Rule Type** | Change the type of a rule by converting a MatchedRule into a LazyRule, and vice versa. |
| | **Inheritance-related** | Applicable on Matched and Lazy rules |
| 11 | *Extract Superrule* | Introduce a common superrule for a set of rules which share common supertypes for input/output elements. |
| 12 | *Pull Up Binding* | Move common bindings of subrules to their common superrule. |
| 13 | *Pull Up Filter* | Move common filters of subrules to their common superrule. |
| 14 | *Eliminate Superrule* | Eliminate a common superrule and the inheritance relationships to the subrules. |
| 15 | *Push Down Binding* | Move the bindings of a superrule to all of its subrules. |
| 16 | *Push Down Filter* | Move the filters of a superrule to all of its subrules. |
| | **OCL-related** | |
| 17 | *Convert Helper Type** | Change the type of a helper by converting a helper operation into a helper attribute, and vice versa. |
| 18 | *Protect Unsafe Target Navigation* | Introduce *resolveTemp* operation for navigating the target model without depending on the transformation state. |
| 19 | *Substituting IF/ELSE Chains with Map* | Complex IF/ELSE expressions can be substituted by maps in case of value conversions. |
| 20 | *Improve Opposite Reference Computations* | If missing backward references in metamodels are calculated by iterator-based operations, substitute this calculation with *refImmediateComposite* operations. |
| 21 | *Shorten Navigation By Context Switch* | Shorten OCL expressions by optimizing the navigation length by setting the appropriate context. |
| 22 | *Replace Select/First with Any* | Substitute select/first operation chains by any operation for finding a specific element fulfilling a certain condition. |
| 23 | *Replace allInstances with Navigation* | Sometimes calculating all instances of a specific type may be replaced by navigating to a specific set of elements. |
| 24 | *Introduce Short-Circuits* | AND and OR expressions can be optimized by surrounding them with IF/ELSE expressions. |

*represent subcategories which comprise several refactorings themselves

Table 1 – Refactoring Catalogue for M2M Transformations

*Refactoring Steps*:
1. Modify In/Out pattern element name.
2. Locate references to the old name and adapt them to the new one.

**(2) Rename In/Out Model**
*Problem*: In/Out model name is not descriptive enough.
*Solution*: Substitute In/Out model name for a more meaningful one.
*Preconditions*: There is not an In/Out model with the same name.
*Parameters*:
1. The model to be renamed.
2. The new name for the model.

*Refactoring Steps*:
1. Modify In/Out model element name.
2. Locate references to the old In/Out model name and adapt them to the new one.

**(3) Rename Rule/Helper**
*Problem*: Rule/Helper name is not descriptive enough.
*Solution*: Substitute rule/helper name for a more meaningful one.
*Preconditions*: There is not a rule/helper with the same name.
*Parameters*:
1. The rule/helper to be renamed.
2. The new name for the rule/helper.

*Refactoring Steps*:
1. Modify rule/helper name.
2. Locate references to the old name and adapt them to the new one (note that for matched rules, this step is not needed since they are not explicitly called from other rules).

## 4.2 Restructuring Refactorings

Besides simple renaming, refactorings are needed to improve the structure of transformations. In particular, this requires to restructure rules and helpers. The refactorings of this category deal with the problem that transformation rules, especially matched rules, tend to grow big, i.e., one rule does most of the work by generating a multitude of elements using a large output pattern block consisting of a huge amount of output pattern elements.

In order to ensure the readability and maintainability of transformation rules, restructuring refactorings aimed at transforming large transformation rules into several smaller ones by either splitting a matched rule into several matched rules or by delegating functionality to additional lazy rules are needed. In Table 1, we provide refactorings aimed at these restructuring needs. In particular, refactorings 4 and 5 are in charge of, respectively, extract and inline rules and helpers. Then, refactorings 6 and 7 allow merging or splitting rules or helpers whereas refactorings 8 and 9 do the same for bindings. Finally, refactoring number 10 allows changing the type of a rule.

For exemplifying these restructuring refactorings, we present in detail three refactorings of the catalogue which are applicable to the running example. First, we extract a global helper which is used for eliminating the duplicated code in the distinct-foreach

output pattern elements of List. 1. Subsequently, we eliminate the distinct-foreach output pattern elements on the one hand by extracting lazy rules and on the other hand by extracting matched rules. Then, after the examples, we also provide a short description for each of the remaining refactorings of this category.

**(4) Extract Global Helper from OCL Expression**
*Problem*: Some OCL expressions are too complex (e.g. sequential application of several iterator-based operations on one collection) or duplicated expressions exist within a transformation. The former results in less readable code and the latter in less maintainable code.
*Solution*: Extract a global helper for computing the requested value(s).
*Preconditions*: None.
*Parameters*:

1. The new name for new helper.
2. The OCL expression.
3. Pointers to the places where the OCL expression is meant to be substituted by the new helper.

*Refactoring Steps*:

1. Determine the context and the parameters for the helper. The context is defined as the type of the element on which the helper should be called. The parameters for the helper are additional values which are used within the computation but not accessible via the context object.
2. Determine the return type of the helper. This is the type of the last executed statement of the computation.
3. Create the helper with a meaningful unique name.
4. Move OCL expression to the helper body.
5. If a context is required, introduce the keyword self as the first statement of the helper body.
6. Add a call to the helper for each place where the extracted code was residing.

*Example*: Consider the OCL expressions used for computing the attributes for the distinct-foreach output pattern element in List. 1. An excerpt of the refactored transformation is shown in the following. In particular, the refactoring is applied twice: first for extracting a helper for computing all attributes of a class and second from this helper a second helper is extracted for computing all properties (intermediate result for computing all attributes).

```
1  helper context UML!Class def : allProperties() : Sequence (UML!Properties) =
2    self.allClasses().including(self).flatten()->collect(e|e.ownedProperty).flatten();
3
4  helper context UML!Class def : allAttributes() : Sequence (UML!Properties) =
5    self.allProperties() -> select (e|not e.primitiveType.oclIsUndefined());
6
7  rule Class {
8    from
9      s: UML!Class
10   to
11     t: ER!EntityType (...),
12     attributes : distinct ER!Attribute foreach(a in s.allAttributes()) (
13     ...
14     ),
15     ...
16 }
```

Listing 2 – Extract global helper example.

**(4) Extract Lazy Rule from Distinct-Foreach Output Pattern Element**
*Problem*: A matched rule uses the deprecated keyword *distinct-foreach* to produce a collection of output elements from a collection of input elements.
*Solution*: Extract a lazy rule from the output pattern element and call it in the matched rule by iterating the collection of input elements.
*Preconditions*:

1. The output pattern element is only used in one binding of the matched rule.
2. The bindings of the output pattern element are only using the iterator variable, i.e., the output pattern element must be self-contained.

*Parameters*:

1. The name for the new lazy rule.
2. The distinct-foreach expression.

*Refactoring Steps*:

1. Determine the types of the input/output pattern elements of the to-be-created lazy rule. The input pattern element type is defined as the element type of the iterator. The output pattern element type is defined as the type of the distinct-foreach output pattern element.
2. Create the lazy rule with a proper name.
3. Move bindings of the distinct-foreach output pattern element to the lazy rule.
4. Call the lazy rule in the matched rule by iterating over the collection of the input elements by using the *collect* operator for collecting the produced elements of the lazy rule calls.

*Example*: Consider the distinct-foreach output pattern element in List. 1 for producing attributes. This definition can be refactored as follows.

```
1  rule Class {
2    from
3      s: UML!Class
4    to
5      t: ER!EntityType (
6        name <- s.name,
7        features <- s.allAttributes()->collect (e|thisModule.Attributes(e),
8        ...
9      )
10 }
11
12 lazy rule Attributes {
13   from
14     s: UML!Property
15   to
16     t: ER!Attribute (
17       name <- s.name,
18       type <- s.primitiveType
19     )
20 }
```

Listing 3 – Extract LazyRule example.

**(4) Extract Matched Rule from Distinct-Foreach Output Pattern Element**
*Problem*: Analogous to previous problem description.
*Solution*: Extract a matched rule from a distinct-foreach output pattern element and adapt the bindings using the distinct-foreach output pattern element by just navigating to the input elements used for generating the output elements.
*Preconditions*:

1. The bindings of the distinct-foreach output pattern element are only using the iterator variable, i.e., it must be self contained.
2. No existing matched rule must match for the same set of input elements.

*Parameters*:

1. The name for the new rule.
2. The distinct-foreach expression.
3. An extra input pattern element for the rule (in case a Cartesian product of input elements is needed).

*Refactoring Steps*:

1. Determine the types of the input/output pattern elements of the to-be-created matched rule. The first element of the input pattern corresponds to the element type of the iterator used in the distinct-foreach output pattern element and the output pattern element type equals the original output pattern element type. Additional input pattern elements may be necessary in order to ensure that the matched rule is executed as many times as the distinct-foreach output pattern element (cf. the following example).
2. Create the matched rule by assigning a proper name for it.
3. Move bindings of the distinct-foreach pattern element to the matched rule.
4. Substitute the bindings that are using the distinct-foreach output pattern element with the navigation to the input elements referred to in the distinct-foreach iterator. The collection of the produced output elements is done automatically by the implicit trace model of ATL.

*Example*: Consider again the distinct-foreach output pattern element in List. 1. This definition is refactored to the following transformation excerpt. Please note that the input pattern element of the rule *Attributes* has to match as many times and for exactly the same elements as previously the distinct-foreach pattern element did. Therefore, it is not sufficient to match only for properties, but in addition the cartesian product of properties and classes has to be generated from which the appropriate combinations have to be selected by an additional filter which checks if a class directly or indirectly owns a certain property.

```
rule Class {
  from
    s: UML!Class
  to
    t: ER!EntityType (
      name <- s.name,
      features <- s.allAttributes() -> collect(p | Tuple {s = p, c = s},
      ...
    )
}

rule Attributes {
  from
    s : UML!Property,
    c : UML!Class (
      c.allAttributes()->includes(s)
    )
  to
    t: ER!Attribute (
      name <- s.name,
      type <- s.primitiveType
    )
}
```

Listing 4 – Extract matched rule example.

**(5) Inline Rule**
*Problem*: Proliferation of small rules with the responsibility of creating very few or only one element makes the transformation more difficult to understand.
*Solution*: Inline the small rule into another rule.
*Preconditions*:
1. There exist a rule that matches a source element from where the source element of the rule to inline can be reached.
2. The small rule is a matched rule or it does not require special control over its execution.

*Parameters*:
1. The rule to be inlined.
2. The target rule.
3. Pointers to rule bindings using the target elements created by the rule to be inlined.

*Refactoring Steps*:
1. Inline the target pattern elements of the to-inline rule into the target rule.
2. Substitute source element references in the to-inline rule bindings for a navigation expression from the target rule source element.
3. If the target elements created by the to-inline rule are used in the target rule, substitute automatic resolution and resolveTemp expression for refereces to target pattern element variables.
4. If the target elements created by the to-inline rule are used in other rules, substitute automatic resolution or resolveTemp expressions for a resolveTemp expression with the desired target element and corresponding source element as parameters.

**(5) Inline Helper**
*Problem*: The code of a helper is as explicit as its name.
*Solution*: Inline the code of the helper into the calling helper or rule.
*Preconditions*: There are no multiple calls to the helpers (if there are multiple calls keeping it separated will enhance reusability and maintainability).
*Parameters*:
1. The helper to be inlined.
2. Pointers to the rules and/or helpers using the to-inline helper.

*Refactoring Steps*:
1. Substitute the calls to the helper for its code.
2. Remove the helper.

**(6) Merge Rule**
*Problem*: There exists in the transformation two similar rules with only small differences in Filter and Bindings.
*Solution*: Merge the two rules in one rule.
*Preconditions*:
1. The two rules match and create the same kind of elements.
2. The matches of the two rules are disjoint (i.e., the filter conditions are disjoint).

*Parameters*:
1. The rule to stay.
2. The rule to remove.

3. Flag telling if the filters need to be combined or removed.
4. Bindings of the rule to stay that should be adapted.

*Refactoring Steps*:
1. Choose a rule to stay and a rule to remove.
2. If the complete set of instances of the input pattern type are matched remove the filter from the rule to stay. If not, connect the two filters by the logical operator OR.
3. Adapt the bindings of the rule to stay.

**(7) Split Rule**
*Problem*: A rule creates two different configurations of output pattern elements by refining the match into two subsets when setting the features of target model elements.
*Solution*: Split the rule in two such that each configuration is created in its own rule.
*Preconditions*:
1. The rule creates different configurations of output pattern elements.
2. The rule uses IF/ELSE expressions in the bindings to create these two subsets.

*Parameters*:
1. The name for the new rule.
2. The original rule.

*Refactoring Steps*:
1. Create a new empty rule.
2. Add the if condition of the bindings to the filter of the original rule and its negative to the new rule.
3. Copy the source pattern and the else part of the bindings to the new rule.
4. Eliminate the else part of the bindings from the original rule.

**(8) Merge Binding**
*Problem*: A target feature is being set by two bindings which is implicitly interpreted in ATL as union. The feature setting will be less complex and easier to understand if it is set in just one binding.
*Solution*: Merge the two bindings setting the same target feature.
*Preconditions*: The target feature of the binding is multi-valued.
*Parameters*:
1. The binding to stay.
2. The binding to remove.

*Refactoring Steps*:
1. Select one of the two bindings to stay.
2. Add the other binding by using the union operation at the end of the binding to stay.
3. Delete the discarded binding.

**(9) Split Binding**
*Problem*: A target feature is being set by a binding which code is too complex and difficult to read and understand.
*Solution*: Split the binding in two bindings setting the same target feature.
*Preconditions*:
1. The target feature of the binding is multi-valued.
2. The binding initialization expression is decomposable.

*Parameters*:
1. The binding to split.
2. The subexpression of the binding to be transferred to a second binding.

*Refactoring Steps*:
1. Create a new binding setting the same target feature.
2. Split the original binding initialization expression.
3. Pass one of the subexpressions to the second binding.

**(10) Convert Rule Type**
*Problem*: A *lazy rule* is defined instead of a *matched rule* Using the latter is the recommended programming style in ATL and should be used when no explicit rule execution control is needed, e.g., for executing the rule for one input element several times.
*Solution*: Change the type of the rule from *Lazy* to *Matched*.
*Preconditions*:
1. The set of source elements to translate does not contain duplicates or source duplicates do not have to be translated to target duplicates.
2. All source elements matching the source pattern and passing the filter are to be transformed (even when there is no reference from other rules).

*Parameters*:
1. The lazy rule.
2. Pointers to the places using the lazy rule.

*Refactoring Steps*:
1. Change the type of the rule from *Lazy* to *Matched*.
2. Locate the bindings using references to the *lazy rule*.
3. Substitute the expression used to call the *lazy rule* to only the source element or the list of the source elements to be assigned in the given binding.

## 4.3 Inheritance-Related Refactorings

As in object-oriented refactoring catalogues, the concept of inheritance, in our domain inheritance between rules, creates its own category of refactorings. Thus, in Table 1, we present some refactorings that work over this concept.

The refactoring number 11 is in charge of extracting superrules (note that, in addition to the common functionality, the input element patterns and output element patterns of the involved rules need common superclasses to be able to introduce a common superrule) whereas refactoring 14 does the opposite job. Common bindings in subrules initializing features of the superclases can be extracted to a common superrule. The same happens with filters. Refactorings 12 and 13 perform this *pulling up* whereas refactorings 14 and 15 allow the opposite, push bindings and filters down from superrules to subrules.

For the implementation of these refactorings knowledge about the input and output metamodels is needed. For example, deciding if a filter can be extracted to a superrule or finding where to place an extracted superrule in an already existing hierarchy needs to exploit this knowledge.

In the following, we present in detail one of these inheritance related refactorings. Afterwards, we provide a description for the rest of the refactorings of this category.

**(11) Extract Superrule out of Matched Rules**
*Problem*: Two or more rules contain similar functionality and share common supertypes for their input/output pattern elements.
*Solution*: Extract a superrule to collect commonalities such as similar bindings and filters.
*Preconditions*: The input/output patterns elements must be of equal type or must have common supertypes.
*Parameters*:

1. The name for the new superrule.
2. The rule to inherit from (optional).
3. Pointers to all the rules to become subrules.
4. Pointers to the bindings and filters to be moved to the superrule.
5. Pointers to the bindings and filters to be removed from the subrules.

*Refactoring Steps*:

1. Find most specific common superclasses in the input/output metamodel for the input/output pattern elements.
2. In case the rules already have a superrule, find the appropriate place of the to-be-created superrule in the rule inheritance hierarchy.
3. Add an abstract rule that acts as superrule.
4. Add inheritance relationships between super/subrules.
5. Pull up common bindings and filters of the subrules.

*Example*: Considering our running example, by extracting for each distinct-foreach output pattern element a matched rule, the resulting matched rules have common bindings. To eliminate these duplicated code fragments, a new abstract rule *Property* is introduced acting as superrule for the extracted matched rules. By further applying the Pull Up Bindings refactoring, common bindings of the matched rules are now encapsulated into the superrule.

```
1  abstract rule Property{
2    from
3      s : UML!Property,
4      c : UML!Class
5    to
6      t: ER!Feature (
7        name <- s.name,
8              ...
9      )
10 }
11
12 rule Attribute extends Property{...}
13
14 rule WeakReference extends Property{...}
15
16 rule StrongReference extends Property{...}
```

Listing 5 – Extract SuperRule example.

**(12) Pull Up Binding**
*Problem*: Two rules define exactly the same binding duplicating code and hampering maintainability. Typically, this refactoring comes after the *Extract Superrule* refactoring.
*Solution*: Pull up the binding to a superrule.
*Preconditions*:

1. There are two rules holding the same binding definition.

2. There exists a superrule in the inheritance hierarchy for the rules which is able to define this binding.

*Parameters*:
1. The binding to pull up.
2. The rules using the binding.
3. The rule where to put the binding.

*Refactoring Steps*:
1. Find a superrule in the inheritance hierarchy that can hold the binding.
2. Copy the binding to the superrule.
3. Delete the binding from the subrules.

**(13) Pull Up Filter**
*Problem*: Two rules define exactly the same filter duplicating code and hampering maintainability. Typically, this refactoring comes after the *Extract Superrule* refactoring.
*Solution*: Pull up the filter to a superrule.
*Preconditions*:
1. There are two rules holding the same filter definition.
2. There exist a superrule in the inheritance hierarchy for the rules holding the filter.

*Parameters*:
1. The filter to pull up.
2. The rules using the filter.
3. The rule where to put the filter.

*Refactoring Steps*:
1. Find a superrule in the inheritance hierarchy that can hold the filter.
2. Copy the filter to the superrule.
3. Delete the filter from the subrules.

**(14) Eliminate Superrule**
*Problem*: A superrule and its subrules are too similar that the inheritance hierarchy is adding complexity to the transformation without adding any value.
*Solution*: Eliminate superrule.
*Preconditions*: None.
*Parameters*:
1. The superrule.
2. The subrules.
3. Pointers to places using the superrule (needed only for lazy rules).

*Refactoring Steps*:
1. Use *Push Down Filter* and *Push Down Binding* to move all the generalized behaviour to the subrules.
2. Adjust references pointing to the superrule to point to the subrule (only needed for *lazy rules*).
3. Eliminate superrule.

**(15) Push Down Binding**
*Problem*: A superrule's binding is only required in specific subrules.
*Solution*: Push down the binding to the rule using it.
*Preconditions*:

1. There exist subrules in the inheritance hierarchy that require the binding.
2. The binding is only required by the subrules that are going to receive the binding.

*Parameters*:

1. The superrule.
2. The subrules.
3. The binding to push down.

*Refactoring Steps*:

1. Copy binding to the subrules.
2. Eliminate binding from the superrule.

**(16) Push Down Filter**
*Problem*: A superrule's filter is only relevant for specific subrules.
*Solution*: Push down the filter to the subrules where it is relevant.
*Preconditions*:

1. There exist subrules in the inheritance hierarchy that require the filter.
2. The filter is required only by the subrules that are going to receive it.

*Parameters*:

1. The superrule.
2. The subrules.
3. The filter to push down.

*Refactoring Steps*:

1. Copy the filter to the subrules.
2. Eliminate the filter from the superrule.

## 4.4 OCL Refactorings

OCL is heavily used in model transformations for queries and value computations. For providing refactorings for OCL expressions within transformations, we reuse existing OCL design rules such as those introduced in [CT07] to improve the quality of OCL expressions in terms of readability and maintainability (in Table 1, refactorings 20 to 24 address these improvements). Furthermore, we introduce new refactorings that are especially tailored to transformations such as querying the target model or optimizing helpers which are based on OCL expressions. The refactoring number 17 allows the optimization of helpers whereas refactoring 18 is aimed to remove unsafe target model navigations. Refactoring 19 helps to simplify transformations by removing complex IF/ELSE chains for data conversions—a typical functionality needed in transformations.

To illustrate the usage of OCL refactorings, we provide an example of a frequently applicable refactoring, the number 17, namely to convert an operation helper into an attribute helper. This refactoring aims at improving the execution time due to usual caching techniques of transformation engines for attribute helpers when they are called for the same context element and/or using the same parameters. Subsequently,

as it we did for the other categories, each refactoring of this category is described.

**(17) Convert Helper Operation into Helper Attribute**
*Problem*: A computation intensive helper operation needs to be called for the same element(s) several times.
*Solution*: Convert the helper operation into a helper attribute to take advantage of caching support.
*Preconditions*: The helper does not have any parameters.
*Parameters*: The helper to be converted.
*Refactoring Steps*:

1. Convert the operation helper into an attribute helper.
2. Replace operation calls with attribute calls.

*Example*: Considering our running example, we can apply this refactoring to the helper operation for computing all superClasses for a given class. In the concrete syntax only the round brackets have to be eliminated to convert the operation to an attribute. Please note that the computation of all superClasses is recursive, so also the call of the allClasses operation within the body has to be changed to an attribute call. Although this refactoring seems to be only a minimal modification, it can lead to huge performance boost which is evaluated in the next section.

```
1 helper context UML!Class def: allClasses : Sequence(UML!Class) =
2     self.superClasses->iterate(e; acc : Sequence(UML!Class) = Sequence {} |
3     acc->union(Set{e})->union(e.allClasses) );
```

Listing 6 – Convert Operation Helper into Attribute Helper Example.

**(18) Protect Unsafe Target Navigation**
*Problem*: Target model navigation, which is strongly discouraged, is used to access target elements created in other rules. This can lead to unexpected results as ATL does not impose any order in the execution of the rules.
*Solution*: Substitute the unsafe target navigation for the provided *resolveTemp* operation.
*Preconditions*: None.
*Parameters*:

1. The rule where the required target element is created.
2. The name of the required output pattern element.
3. The unsafe target navigation expression.

*Refactoring Steps*:

1. Find the name of the rule where the required target element is created.
2. Find the name of the output pattern element creating the target element.
3. Substitute the unsafe target navigation for the *resolveTemp* operation passing the source object of the rule and the output pattern element variable name as parameters.

**(19) Substituting IF/ELSE Chains with Map**
*Problem*: Complex nested IF/ELSE chains are used to convert values from the source model into values needed for the target model, e.g., convert Java to SQL data types.
*Solution*: Substitute IF/ELSE chains with a map representing the relation between the different values and a helper that access it.

*Preconditions*: Source values must not have several correspondences to target values, because the source values represent the keys of the map that have to be, of course, unique.
*Parameters*:

1. The name for the new map.
2. The name for the new helper.
3. The IF/ELSE chain.

*Refactoring Steps*:

1. Identify through IF/ELSE conditions the mappings between the values.
2. Create a map representing the mapping of values.
3. Create a helper that takes a source value and returns the corresponding target value from the map.
4. Substitute the call to the IF/ELSE chains with a call to the new helper.

**(20) Improve Opposite Reference Computations**
*Problem*: Opposite containment references are calculated in a complex and inefficient way using allInstances operations.
*Solution*: Substitute the opposite reference calculation for the *refImmediateComposite* operation.
*Preconditions*: The reference to calculate is a containment reference.
*Parameters*: The expression calculating the opposite reference.
*Refactoring Steps*:

1. Substitute the call to the references calculation for a call to the *refImmediate-Composite* operation.
2. Delete the code in charge of calculating opposite references if it is not used for other purposes.

**(21) Shorten Navigation By Context Switch**[4]
*Problem*: Use of very long OCL expression impacting the readability and understandability of the transformation.
*Solution*: Change the context of the OCL expression in order to shorten the navigation length.
*Preconditions*:

1. The expression is defined using a single instance of the context type.
2. The context to switch can be navigated from the original one and/or belongs to the same taxonomy.

*Parameters*:

1. The OCL expression to be shortened.
2. The new context type.

*Refactoring Steps*: Redefine the expression over the new context type.

**(22) Replace Select/First with Any**
*Problem*: An OCL expression containing Select/First operation chains is difficult to read and unnecessarily expensively calculated.
*Solution*: Substitute the operation chain for any operation for finding a specific element fulfilling a certain condition.

[4]This refactoring is quite complex. Thus, only a summarized version of its description is provided. See [CT07] for a complete description.

*Preconditions*: None.
*Parameters*: The OCL expression.
*Refactoring Steps*:
1. Locate corresponding Select/First operation chain
2. Substitute the chain by the OCL *Any* Operation using the condition of the *Select* operation.

**(23) Replace allInstances with Navigation**
*Problem*: allInstances operation is used when the same result can be achieved by using cheaper operations. As the possibilities of use of the allInstances operation are huge, we focus only on expression that uses allInstances followed by a condition that refines the result set.
*Solution*: Substitute the allInstances operation for a navigation expression reaching the same set of resulting elements.
*Preconditions*: The type over which allInstances is applied coincides with the context type of the expression. They may not be applied if the expression already contains any explicit or implicit reference to the self variable.
*Parameters*:
1. The OCL expression.
2. The navigation expression used to substitute the allInstances operation.

*Refactoring Steps*: Substitute the allInstances operation for a normal navigation expression that applies the condition over self.

**(24) Introduce Short-Circuits**
*Problem*: A complex boolean expression is used. As many OCL implementations do not provide short-circuit evaluation, all the conditions are evaluated, which could seriously impact performance when using large models.
*Solution*: Surround the OCL AND and OR expression by IF/ELSE.
*Preconditions*: The OCL implementation does not implement short-circuit evaluation.
*Parameters*: The OCL expression.
*Refactoring Steps*:
1. Group OR or AND expression.
2. If the expression is *cond1() or cond2()* substitute it for *if cond1() then cond2() else false endif*. Else, if the boolean expression is AND, substitute if for *if cond1() then true else cond2() endif*.

## 4.5 Reuse Potential of the presented Refactorings

The proposed refactoring catalogue has been developed by analyzing a set of exogenous, out-place model transformations (for the terminology see [MG06]), although most of the refactorings are also applicable to transformations written in the ATL refining mode [TMJC11], that performs endogenous, in-place transformations. To show the reuse potential of the refactorings for other model transformation languages, we have selected three prominent transformation languages and evaluated which refactorings of the afore presented catalogue are applicable for them.

In this context, we examined two languages of the QVT language family, namely the imperative transformation language QVT Operational (QVT-O) and the declarative transformation language QVT Relational (QVT-R) [OMG11]. By this, we cover a wide spectrum of current transformation languages ranging from pure declarative to

| | QVT-R | QVT-O | ETL |
|---|---|---|---|
| **Renaming** | | | |
| Rename In/Out Pattern Element | ✓ | ✓ | ✓ |
| Rename In/Out Model | ✓ | ✓ | ✓ |
| Rename Rule/Helper | ✓ | ✓ | ✓ |
| **Restructuring** | | | |
| Extract Helper/Rule | ✓ | (✓) | (✓) |
| Inline Helper/Rule | ✓ | (✓) | (✓) |
| Merge Rule | ✓ | ✓ | ✓ |
| Split Rule | ✓ | ✓ | ✓ |
| Merge Binding | ✓ | ✓ | ✓ |
| Split Binding | ✓ | ✓ | ✓ |
| Convert Rule Type | ✓ | × | ✓ |
| **Inhertitance-related** | | | |
| Extract Superrule | × | (✓) | ✓ |
| Pull Up Binding | × | ✓ | ✓ |
| Pull Up Filter | × | ✓ | ✓ |
| Eliminate Superrule | × | (✓) | ✓ |
| Push Down Binding | × | ✓ | ✓ |
| Push Down Filter | × | ✓ | ✓ |
| **OCL related** | | | |
| Convert Helper Type | × | × | ✓ |
| Protect Unsafe Target Navigation | ✓ | ✓ | ✓ |
| Substituting IF/ELSE Chains with Map | × | ✓ | ✓ |
| Improve Opposite Reference Computations | × | ✓ | ✓ |
| Shorten Navigation By Context Switch | ✓ | ✓ | ✓ |
| Replace select/first with any | ✓ | ✓ | ✓ |
| Replace allInstances with Navigation | ✓ | ✓ | ✓ |
| Introduce Short-Circuits | ✓ | ✓ | ✓ |

| ***Legend*** | |
|---|---|
| ✓ | fully applicable |
| (✓) | partially applicable |
| × | not applicable |

Figure 2 – Reuse Potential of the Refactoring Catalogue for QVT-R, QVT-0, and ETL.

pure imperative languages. Furthermore, we also included the hybrid Epsilon Transformation Language (ETL) [KPP08] to evaluate if the refactorings developed for ATL are applicable for other hybrid languages as well. Examining the QVT languages allows to investigate if the refactorings are applicable for languages following either the imperative paradigm or the declarative paradigm.

Figure 2 summarizes the results by stating for each refactoring category if it is fully applicable, i.e., all refactorings are reusable, partially applicable, i.e., some refactorings are reusable, or not applicable at all.

The first section of Figure 2 regards the renaming refactorings which are all applicable for QVT-R, QVT-O, and ETL. The reason is that all these languages, as ATL, use variable names for the in/out models, rules, helpers, and in/out pattern elements.

The second section is about restructuring refactorings for which the results vary for each language. For ETL, nearly all refactorings are reusable, except the refactorings

which are tailored to the foreach-distinct output pattern elements of ATL which has no corresponding concept in ETL. This concept is only shared by QVT-R where an output pattern element may represent a set of elements. Thus, all refactorings of this section can be applied to QVT-R. For QVT-O the only exception is the convert rule type refactoring given that QVT-O only supports lazy rules due to its imperative nature. Instead, QVT-R allows to define top relations and non-top relations which corresponds to ATL matched and lazy rules, respectively. Thus, the convert rule type refactoring is applicable for QVT-R, and also for ETL where also a kind of matched rules (which are automatically executed by the transformation engine) and lazy rules are supported. Interestingly, also the merge/split binding refactorings can be reused, because QVT-R assumes implicitly—such as ATL—to build the union of assigned feature values calculated by different bindings. In ETL as well as in QVT-O there is not only the possibility to override already existing bindings for a feature, but also to extend them. For example, ETL allows to use the *addALL* operation which inserts elements into an collection of elements and QVT-O provides a specific assignment operator which is extending the collection with additional elements.

The inheritance-related refactorings are reusable for ETL and QVT-O, because these languages also provide inheritance between rules similar as in ATL. However, QVT-O does not supported matched rules, thus extracting/eliminating super matched rules is not needed for QVT-O. QVT-R has no support for inheritance between relations, thus this category of refactorings is not reusable at all for QVT-R.

OCL related refactorings are reusable in particular for ETL and QVT-O, because these languages provide similar OCL support as ATL does. More specifically, the last four OCL-related refactorings are only based on standard OCL, thus they are applicable for all transformation languages, because all of them are based on the core of OCL. However, the first four OCL-related refactorings are based on OCL extensions provided by ATL. As can be seen by the evaluation, ETL and QVT-O provide similar extensions to OCL. For example, in QVT-O there is the *container* operation introduced for computing the inverse reference and in ETL there is the possibility to reuse operations defined by the underlying EMF framework, in this context the *eContainer* operation for computing the inverse reference. Furthermore, also the *Map* data type of ATL has corresponding concepts in ETL (there is also a *Map* data type) and in QVT-O (there is the data type *DictionaryType*). Thus if/else chains may be also substituted with instantiating these data types. However, for QVT-R, there are no such OCL extensions foreseen, thus this kind of refactorings is mostly not reusable for QVT-R. Furthermore, in QVT-O and ETL there are several operations provided for making the navigation to the target model save by having several different *resolve* operations which can be used for this purpose. In QVT-R, there is no such operation, but because relations are executed with the check-before-enforce semantics, i.e., an element is only created when it is not already existing in the target model, the relations may be triggered several times which allows to access already created elements without recreating them. Finally, ETL is the only language of the three examined that also supports cached queries, thus only for this language the convert helper refactoring is applicable.

## 5 Impact of Refactorings on Quality Attributes

To provide more insights on the applicability of the proposed refactorings as well as their impact on quality attributes of model transformations, we discuss refactoring

possibilities of the illustrative example from Section 2. More specifically, we describe two different chains of refactorings, present essential metrics of the transformations before and after each refactoring, and finally, we elaborate on their implications on the transformation quality attributes.

## 5.1 Refactoring Chains

Two different chains of refactorings (cf. Fig. 3) are applied to the initial version (T1) of the transformation (cf. List. 1) resulting, after several intermediate steps, in a version which is solely based on matched rules following the pure declarative approach of ATL (called T4 in the following) and in a version consisting only of one matched rule which delegates to lazy matched rules mixing the declarative and imperative programming styles (called T6 in the following). The rationale for using two different refactoring chains is to illustrate that there are several ways to improve a model transformation, depending on the designer's goal. The refactoring chains result in different solutions whereby each has its own properties. Please note that the six versions of the transformation depicted in Fig. 3 are later used in Section 6 for evaluating the impact of the refactorings on the execution performance of transformations.

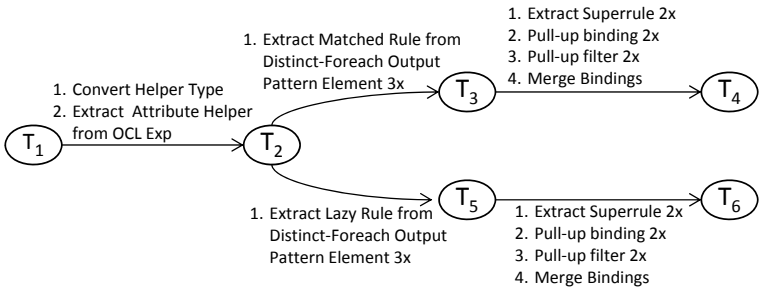


Figure 3 – Refactoring chains producing six different transformation versions.

The transformation T4 is shown in List. 7. First, the existing helper `allClasses` is converted from an operation helper to an attribute helper, and subsequently, the additional helper `getAllProperties` is produced by applying the *Extract Helper* refactoring for eliminating the duplicated code for querying all direct and indirect possessed properties of a class. Second, three matched rules (`Attribute`, `WeakReference`, and `StrongReference`) have been created by extracting the distinct-foreach constructs from the rule `EntityType`. Third, a superrule is extracted from the rules `WeakReference` and `StrongReference`, and subsequently, from the combination of this resulting rule with the rule `Attribute`, the rule `Feature` is extracted. Fourth, the redundant bindings and filters of the subrules are pulled up to the new superrules. Finally, the feature assignments in the rule `EntityType` are merged by exploiting polymorphism and retrieving properties from the trace model in one step.

```
1  module UML2ER;
2  create OUT : ER from IN : UML;
3
4  helper context UML!Class def: allClasses : Sequence(UML!Class) =
5      self.superClasses->iterate(e; acc : Sequence(UML!Class) = Sequence {} |
6      acc->union(Set{e})->union(e.allClasses) );
7
8  helper context UML!Class def : getAllProperties : Sequence (UML!Properties) =
9    self.allClasses.including(self).flatten()->collect(e | e.ownedProperty).flatten();
10
11 rule EntityType {
12   from s: UML!Class
13   to t: ER!EntityType (
```

```
14        name <- s.name,
15        features <- s.getAllProperties -> collect(e | Tuple {s = e, c = s}) )}
16
17  abstract rule Feature{
18    from s: UML!Property,
19       c : UML!Class (c.getAllProperties->includes(s) )
20    to t: ER!Feature (
21        name <- s.name)}
22
23  rule Attribute extends Feature{
24    from s: UML!Property,
25      c : UML!Class (not s.primitiveType.oclIsUndefined() )
26    to t: ER!Attribute (
27        type <- s.primitiveType )}
28
29  abstract rule Reference extends Feature{
30    from s: UML!Property,
31      c : UML!Class (not s.complexType.oclIsUndefined())
32    to t: ER!Reference (
33        type <- s.complexType )}
34
35  rule WeakReference extends Reference{
36    from s: UML!Property,
37      c : UML!Class ( not s.isContainment )
38    to t: ER!WeakReference }
39
40  rule StrongReference extends Reference{
41    from s: UML!Property,
42      c : UML!Class (s.isContainment )
43    to t: ER!StrongReference }
```

Listing 7 – UML to ER Transformation (T4).

The transformation T6 is shown in Listing 7. The applied refactoring chain is similar to the previous refactoring chain, except that lazy rules are extracted for distinct-foreach constructs instead of matched rules. Thus, also the resulting transformation code is similar to the previous transformation version, however, the input pattern of the rules are simpler compared to the matched rule version. Moreover, a major difference is in the execution of both refactored transformations. While in T4 all rules are executed independently, in T6 only the first rule is automatically executed which delegates to the lazy rules when appropriate. In the following, we discuss the initial version T1 as well as both refactored versions (T4 and T6) based on dedicated model transformation metrics. Please note that the impact on the execution performance of the refactorings is discussed in Section 6.

```
9   -- header and helpers same as in T4
10
11  rule EntityType {
12    from s: UML!Class
13    to t: ER!EntityType (
14        name <- s.name,
15        features <- s.getAllProperties->collect (e | thisModule.Feature(e)) )}
16
17  lazy abstract rule Feature{
18    from s: UML!Property
19    to t: ER!Feature (
20        name <- s.name ) }
21
22  lazy rule Attribute extends Feature{
23    from s: UML!Property (not s.primitiveType.oclIsUndefined())
24    to t: ER!Attribute (
25        type <- s.primitiveType )}
26
27  lazy rule Reference extends Feature {
28    from s: UML!Property (not s.complexType.oclIsUndefined() )
29    to t: ER!Reference (
30        type <- s.complexType)}
31
32  lazy rule WeakReference extends Reference{
```

```
33   from s: UML!Property (not s.isContainment)
34   to t: ER!WeakReference }
35
36 lazy rule StrongReference extends Reference{
37   from s: UML!Property (s.isContainment)
38   to t: ER!StrongReference }
```

Listing 8 – UML to ER Transformation (T6).

## 5.2 Metric-based Evaluation

An established way of evaluating the impact of refactorings on the quality attributes of a software artefact is to compute metrics on its initial version and on the refactored version. With this purpose, we have surveyed recent work on model transformation metrics [vALvdB09, KGBH10, vAvdB11].

Nevertheless, metrics alone do not provide a clear answer to the question of whether the refactorings improve the quality attributes of the software artefact. For that, it is necessary to find an alignment of metrics to quality attributes, i.e., whether a lower/higher value of a metric improves/worsens a given quality attribute. This is still an open issue in the model transformation field due to the lack of large empirical studies [KGBH10]. However, some initial empirical studies have been already conducted [vAvdB11] that will help us to justify the quality improvements of our refactorings (apart from relying on our own experience in the field). Besides these works from the model transformation field, we also base our argumentation on existing work on metrics for OCL [RGP04, CT06], object-oriented programming languages [CK94, BBM96], and rule-based systems [DV92].

### 5.2.1 Metric Setup

We use a number of metrics to evaluate the quality of transformations. First we present traditional code metrics for measuring the transformation size such as lines of code (LoC) for the textual representation of transformations. ATL transformation code can be automatically transformed to a so-called transformation model [BBG$^+$06] which represents the abstract syntax of the transformation. The transformation model allows us to easily compute the total number of model elements, number of rules, abstract rules (ARules), concrete rules (CRules), bindings, helpers and so on by reusing a transformation from van Amstel et al. [vAvdB11] for producing a metric model.

Second, we use metrics tailored to detect bad smells such as *code duplicates* (CD) by stating the total number of CDs, *god rules* (GD) by stating the total number of them, and *overlong OCL expressions* by stating the maximum OCL expressions length (MEL) in a transformation. Our experience is that a MEL higher than 10 leads to hardly understandable OCL expressions, especially when several navigations and iterator-based operations are used in sequence.

Third, we measure the dependencies of rules based on *Fan-in* a.k.a. afferent coupling and *Fan-out* a.k.a efferent coupling metrics. Fan-in values are calculated by counting the incoming dependencies. More precisely, in case of lazy rules, we measure how often a rule is called from others and, in case of matched rules, we measure how often the trace model entries produced by a rule are accessed by other rules. To reflect inheritance between rules, we evaluate for how many other rules the respective rule acts as superrule. Fan-out values are calculated by counting the outgoing dependencies of a rule. Outgoing dependencies comprise references to superrules, calls to lazy rules, access to trace information created by matched rules, and finally calls to

helpers. In addition to intra-transformation dependencies, we also include measuring the dependencies of a rule to the metamodel, i.e., how many metamodel elements are used by a rule. By considering this kind of information, we aim to evaluate the maintainability of a transformation in case of metamodel evolution.

Fourth, we measure the complexity of rules by computing *Val-in* and *Val-out* metrics. In the context of rule-based model transformations, val-in value corresponds to the number of input pattern elements of a rule, and analogously, val-out value equals the number of output pattern elements. Please note that distinct-foreach output pattern elements produce collections of objects, in contrast to simple output pattern elements which produce single objects. Because retrieving an output element generated for a given input element from such collections requires for additional code, distinct-foreach output pattern elements are counted with a factor of 3 instead of 1 as is used for simple output pattern elements as an approximation for the extra effort.

Finally, we state the *abstractness* of a transformation by comparing the number of abstract rules to the total number of rules.

Table 2 depicts the values of all introduced metrics for the initial as well as for both refactored versions of the illustrative example.

Table 2 – Metrics Overview: Initial version (T1) vs. Matched rule version (T4) vs. Lazy rule version (T6)

| ***Metric*** | ***T1*** | ***T4*** | ***T6*** |
|---|---|---|---|
| LoC | 39 | 44 | 38 |
| #Elements | 122 | 132 | 116 |
| #Rules | 1 | 6 | 6 |
| #ARules | 0 | 2 | 2 |
| #CRules | 1 | 4 | 4 |
| #Helpers | 1 | 2 | 2 |
| #Bindings | 10 | 5 | 5 |
| #CD | 3 | 0 | 0 |
| #GD | 1 | 0 | 0 |
| MEL | 13 | 6 | 6 |
| Avg Fan-in | 0 | 0,83 | 0,83 |
| Avg Fan-out | 18 | 6,17 | 4,8 |
| -internal | 1 | 1 | 1 |
| -external | 17 | 5,17 | 3,8 |
| Avg Val-in | 1 | 1,83 | 1 |
| Avg Val-out | 10 | 1 | 1 |
| Abstraction | 0 | 0,33 | 0,33 |

### 5.2.2 Impact on quality attributes

The goal of the refactorings is to enhance quality attributes such as readability, extendability, maintainability, understandability, and reusability. Thus, in the following, we discuss the implications of the refactorings on quality attributes based on the aforementioned metrics. Again, note that the two transformations resulting from the two refactoring chains present slightly different quality attributes. There is not a single best transformation, it all depends on the quality attributes the designer is most interested in her specific context.

*Transformation size.* As can be seen in Table 2, the initial version of the transformation has almost the same number of LoCs as T6 and less than T4. However, when considering the total number of model elements, T6 is the shortest version. In general, it seems that measuring the size of transformations is more appropriately done on the abstract syntax level and not on the concrete syntax level to be inde-

pendent of code formatting rules [Jon94]. T4 is definitely the longest version, on the concrete as well as on abstract syntax level, due to the fact that more complex input patterns are needed for the matched rule solution. Furthermore, both refactored versions comprise additional elements for defining abstract transformation rules. In the illustrative example only a small amount of features is used for each metamodel class, thus the reuse of the bindings between the rules is marginal. Nevertheless, in practical settings more bindings may be reused and shared in abstract rules which definitely pays off the space needed for the additional abstract rules. The largest gain in reducing the transformation size is that duplicated OCL expressions are eliminated based on introducing one additional helper for calculating the direct and indirectly contained properties of a class. To conclude, although the size could not be significantly reduced by the refactorings, considering other aspects in the measurement may show an enhancement of the readability and understandability as studies in the field of object-oriented programming indicate [Jon94].

*Bad smells.* Three bad smells existed in the initial version, namely *code clones*, *overlong OCL expressions*, and one *god rule*, whereas all of them could be removed in both refactored versions. With the help of the refactoring *Extract Helper* on the one hand code clones could be eliminated and on the other hand the overlong OCL expressions could be split into several smaller OCL expressions. This is reflected by the maximum expression length of the OCL expression that could be reduced by 50 % in comparison to the initial transformation. Finally, also the god rule could be eliminated by extracting additional rules out of the distinct-foreach output pattern elements. By this, several aspects which have been intermingled in the god rule could be separated into several different rules providing a higher cohesion of the resulting rules.

*Rule dependencies.* The initial version consisted of only one rule, hence the average fan-in value was zero. This means that this rule may be changed without requiring further adaptations outside of the rule. The average fan-out value was 18, meaning that this rule had 18 dependencies to other elements. In particular, the rule depends on 17 different metamodel elements. This shows that the rule covers mostly all elements of the input and output metamodels and thus, it represents the whole transformation indicating again the god rule bad smell. Concerning the maintainability of this rule, it may be argued that it is easily changeable, because no other rules have to be adapted, but at the same time it has to be mentioned that nearly the complete transformation logic as well as the complete input and output metamodels have to be conceived and understood as a prerequisite, which significantly mitigates the maintainability especially for larger transformations. The situation is quite different for the refactored versions which have a much smaller average fan-out value, thus the transformation rules may be easier maintained. Furthermore, in case of metamodel evolution, e.g., an attribute is renamed in the metamodel, the refactored transformation are easier to adapt to the metamodel changes, because the rules have a much clearer focus. However, the separation of the transformation logic into several rules comes with the price of having coupling between rules, but it can be seen on the average fan-in value that the coupling is still very low. This can be considered as necessary coupling level as pointed out by [Ber93], because a modular system without any coupling is useless in general.

*Rule complexity.* The average complexity of the rules has been tremendously lowered by the refactorings. The initial transformation consisting of one rule had a very high average val-out value, which aggravates understandability, extensibility,

and reusability. The problem with the god rule w.r.t. reusability is that the chance of reuse is practically not existing, because it comprises the complete transformation. Furthermore, extensibility is low due to the following two reasons. First, the generated elements of this rule may only be accessed by `Class` objects, so if someone wants to retrieve the generated element for one particular `Property` object to add an additional link within a new rule to this object, the container, i.e., the containing `Class` object, has to be retrieved before the trace model may be queried. Second, a huge bunch of elements is generated by the rule, thus the result of querying the trace model is quite complex. In particular, it is a four tuple where the last three entries are again collections of elements which have been generated by the distinct-foreach output pattern elements. In contrast to the initial version, the refactored versions allow for a higher chance of reusing rules, especially T6 which only uses one-to-one transformation rules. Furthermore, the understandability of the refactored versions is higher, because the rules are much smaller and more focused. Finally, the extensibility of the transformation is higher, because all generated target elements may be retrieved by their direct counterparts in the source model.

However, there may be a trade-off between the size of rules and the degree of fragmentation of transformation logic over a set of rules. Of course, the complexity of the initial rule has been reduced by introducing additional rules, but this comes with additional dependencies between rules. However, the minor increase in fan-in values seems to be reasonable compared to the decrease in the val-in values.

*Abstractness.* For reusability and extendability concerns, the abstractness rate of the transformation under study has been increased by introducing abstract rules which may be used later on as extension points for introducing new subrules in case new classes are added to the input or output metamodels. Furthermore, using inheritance between rules allows to provide more concise rules which are in general also more understandable as it has been observed by [vAvdB11]. However, as for object-oriented programs, the abstractness may have a negative impact on the understandability in case the inheritance hierarchies are getting too deep. Here we have to mention that the inheritance hierarchies of transformation rules are normally closely aligned to the metamodel inheritance hierarchy of classes, thus they strongly depend on the quality of the inheritance hierarchies in the input and output metamodels.

### 5.2.3 Synopsis

By refactoring the transformation T1 using the two refactoring chains, several quality attributes could be improved, while the transformation size remains more or less the same. Especially, by removing the god rule bad smell, future extensions of the transformation should be more easily achieved without ending up with one complex transformation rule realizing the complete transformation. Furthermore, the lazy rule version T6 provides slightly better results compared to the matched rule version T4 when considering fan-out and val-in values. The reason for this is that the matched rules require for this example a more complicated input pattern than the lazy rule equivalents. The next section shows that this small metric value difference results in huge execution performance differences of the discussed transformations.

## 6 Impact of Refactorings on Execution Performance

After showing the possibilities of refactorings to improve a transformation's internal structure, we proceed with an evaluation of their impact on the execution performance

of transformations. Although performance is usually not considered to be the main objective of refactorings, several works showed that refactorings may come along with significant impacts on performance (e.g., cf. [Dem05, OXJF05]). Therefore, we aim to demonstrate that the presented refactorings do not necessarily worsen performance. For the performance evaluation, we again resort to the illustrative example used in the previous sections.

## 6.1 Setup

The following transformation versions have been evaluated (cf. Fig. 3): (T1) initial transformation (List. 1), refactored transformation after (T2) changing operation helpers to attribute helpers, (T3) extract matched rules, (T4) extract matched rules with inheritance, (T5) extract lazy rules, (T6) extract lazy rules with inheritance.

We choose this set of transformation to answer the following questions with respect to performance:

- Impact of changing operation helpers to attribute helpers (T1 → T2)
- Impact of extracting rules from distinct-foreach patterns (T2 → T3, T2 → T5)
- Impact of introducing inheritance between rules (T3 → T4, T5 → T6)

In the experiment, three different synthetic input models are used for evaluating the performance of the transformations. The models have in common that they all comprise 1.000 classes, whereas each class contains 10 properties (properties representing attributes and properties representing references are equally distributed). The significant differences between the models are coming from their different inheritance structures. The first model does not even use a single inheritance relationship and is therefore called *loose* model. The second model is called *deep* model, because the Maximum Depth of Inheritance Tree (MDIT) is 20 and the Average Number of Children (including direct and indirect subclasses) (ANOC) is 10,55. Finally, the third model is called *flat* model, because MDIT is 2 and ANOS is 1,89.

For executing the transformations, we employed the ATL Regular Virtual Machine in its version 3.1.2. The performance figures have been measured on a Lenovo T410s with an Intel(R) Core(TM) i5 CPU M 560 @ 2.67 GHz 2.67 GHz, with 8 GB of physical memory, and running the Windows 7 Professional 64 bits operating system.

## 6.2 Results

The results of the execution performance evaluation are depicted in Table 3, Table 4, Table 5, for the loose input model, flat input model, and deep input model, respectively. For measuring the performance, we are using the following metrics:

- **CPU Time**: Execution time in seconds - without the time for loading the input model and serializing the target model.
- **Instructions**: Processed instructions - how many byte code statements are executed to produce the target model.
- **Speedup**: Ratio between the execution time of the initial transformation and the refactored transformation.

Please note that we measured the CPU time by executing each transformation 10 times and calculated the arithmetic mean of these 10 runs.

## 6.3 Discussion

As one can see in the three tables, our chain of refactorings does not necessarily worsen execution performance and some of them clearly have a positive impact on the performance of the transformation. For example, it is obvious that using attribute helpers instead of operation helpers improves the performance by at least 20 %. These experiments have also been useful to learn more about the performance of ATL itself.

Table 3 – Results of the performance evaluation (loose model)

| Transformations | Instructions | CPU Time | Speedup |
|---|---|---|---|
| **T1** | 1.876.111 | 1,42 s | 1,00 |
| **T2** | 1.504.135 | 1,10 s | 1,28 |
| **T3** | 241.907.174 | 163,34 s | 0,01 |
| **T4** | 507.073 | 56,59 s | 0,03 |
| **T5** | 1.289.112 | 0,98 s | 1,45 |
| **T6** | 1.361.117 | 1,02 s | 1,39 |

Table 4 – Results of the performance evaluation (flat model)

| Transformations | Instructions | CPU Time | Speedup |
|---|---|---|---|
| **T1** | 4.919.541 | 4,10 s | 1,00 |
| **T2** | 3.876.615 | 3,07 s | 1,25 |
| **T3** | 244.808.854 | 171,37 s | 0,02 |
| **T4** | 84.850.488 | 63,21 s | 0,44 |
| **T5** | 3.464.157 | 2,56 s | 1,26 |
| **T6** | 3.643.337 | 2,70 s | 1,47 |

Table 5 – Results of the performance evaluation (deep model)

| Transformations | Instructions | CPU Time | Speedup |
|---|---|---|---|
| **T1** | 17.125.461 | 18,07 s | 1,00 |
| **T2** | 13.380.985 | 15,14 s | 1,19 |
| **T3** | 256.444.024 | 186,31 s | 0,10 |
| **T4** | 97.064.018 | 53,20 s | 0,34 |
| **T5** | 12.237.967 | 9,07 s | 1,99 |
| **T6** | 12.888.967 | 10,65 s | 1,70 |

For instance, though using declarative matched rules is the recommended programming style in ATL, in this example the application of matched rules does not scale compared to the initial transformation and the lazy rule solutions. The reason is the expensive computation of the Cartesian product for duplicating attributes of superclasses for all subclasses. This requirement is more efficiently implemented using lazy rules, because they can be called from the appropriate context, i.e., classes for which all direct and indirect contained properties can be efficiently calculated by using helpers. In contrast, using matched rules which are automatically applied by the transformation engine, this context has to be expensively computed by building the Cartesian product of properties and classes. Obviously, this computation has the matching complexity $|Class| \times |Property|$. Thus, we can conclude that the transformation T3 and T4 both include a performance anti-pattern `Wasted Rule Execution Context` meaning that a rule has to build expensively the context for its execution, although the context is already available in another part of the transformation from which the rule may be called.

Another observation is that inheritance between transformation rules may improve or worsen performance. For matched rules we have a performance improvement, because only the top rules have to be matched (rules having no super rule) and these matches are refined down to the subrules. Thus, instead of calculating three times the Cartesian product of properties and classes in T3, this is only done once in T4. However, for the lazy rule versions, the performance decreases when introducing inheritance. Lazy rules are not automatically executed by the ATL execution engine,

but are called by other rules. If a lazy rule is called, the dispatching strategy used in ATL checks first the types of the given parameter values (by using oclIsKindOf operations) and subsequently the filter conditions. When going from the top rule to the most specific rule, each time the type of the input parameters are checked, even if the type is always the same and only the filter conditions vary as it is the case in our example.

To sum up the discussion, we may conclude that (1) switching from operation helpers to attribute helpers definitely improves the performance of transformations. This is also true for large input models and transformation logic which potentially call the same query on the same context element several times. Thus, caching queries to input models should be definitely considered when designing and refactoring transformations. (2) Refactoring distinct-foreach patterns to rules does not have to worsen execution time, but when extracting matched rules having more than one input pattern element the performance will decrease. (3) Inheritance improves the performance for matched rules because of the matching strategies of ATL, but decreases the performance for lazy rules due to the expensive dispatching phase applied by the ATL virtual machine.

## 6.4 Threats to Validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results, specially regarding the performance results.

*Internal validity—are there factors which might affect the results in the context of ATL?* Concerning the trade-off of using helper operations vs helper attributes, the advantage of helper attributes can decrease if the input model is so large that we cannot benefit from the caching mechanisms of the ATL VM.

In the presented transformation example, the matched rule versions did not scale due to the complex match computations. It has to be noted that this evaluation does not allow to conclude on how extracting matched rules with one input pattern, i.e., matching only one element, from distinct-foreach constructs affects the execution performance. Therefore, we have experimented with an additional transformation which is similar to the presented example, but without duplicating properties of superclasses for subclasses. Thus, simple input patterns are sufficient for the matched rules. The results of this experiment showed that in this case the matched rule versions have a similar performance as the corresponding lazy rule versions.

Concerning the impact of using inheritance between transformation rules on the performance, we concluded that inheritance has a positive impact in case of matched rules and a negative impact in case of lazy rules. However, this result may be influenced by the complex input patterns of the matched rules, thus we also evaluated this aspect by the slightly modified transformation mentioned in the previous paragraph. This additional experiment showed a similar performance impact of using rule inheritance.

Another threat to validity is the dependence on the specific ATL environment used in the evaluation. For executing ATL transformations, two different VMs are currently available, namely the `Regular VM` and the `EMF-specific VM`. It is important to note that the compiler from ATL to VM code is the same for both VMs, but they use different implementations for executing the VM code, e.g., how to retrieve and access model elements. In our experiment we relied on the Regular VM. To gain evidence that the refactorings have similar impacts on the performance for the EMF-specific VM, we ran our tests again on this VM. We explored that the execution times varies

between the both VMs, but the impact of the refactorings on the performance is comparable.

*External validity—to what extent is it possible to generalize the findings for transformation languages in general?* So far, we cannot claim any performance results outside the context of ATL. Nevertheless, the refactorings can indeed be applied on other transformation languages (cf. Section 4.5). Thus, replaying the presented experiments for those transformation languages should enable the possibility of reasoning about the performance impact of the refactorings for those languages as well.

# 7 Implementation

Following a pure MDE approach, we propose to implement transformation refactorings as transformations themselves. In ATL, transformations are expressed as models [BBG+06]. Therefore, they may be input or output of other transformations, so-called Higher Order Transformations (HOT) [TJF+09].

As most of the model representing the transformation will remain unchanged, we propose to use in-place transformations for the implementation of the refactorings. This way we will only need to write transformation rules for the elements that are meant to change during the refactoring. In ATL, this kind of transformations may be developed by using the ATL *refining mode* [TMJC11] which provides dedicated transformation language facilities as well as a dedicated transformation engine.

We want to stress that for some refactorings type inference is needed, e.g., for extracting new superrules. Thus, for HOTs implementing such refactorings, they need to have as input not only the transformation model, but in addition, the metamodels of the input and output models of the transformation subject of refactoring.

For some refactorings an additional input model may be needed, comprising user input and pointers to transformation elements which have to be manipulated during the refactoring step.

List. 9 shows an excerpt of the HOT for the *Extract Superrule* refactoring. Fig. 4 illustrates the metamodel for the additional input model as well as an example model which represents the selected elements for producing the resulting transformation code shown in Section 4.3 (List. 5) after applying the refactoring.

In the current development stage, our refactorings are designed to be semi-automatically executed. For instance, for the *Extract Superrule* refactoring, the transforma-

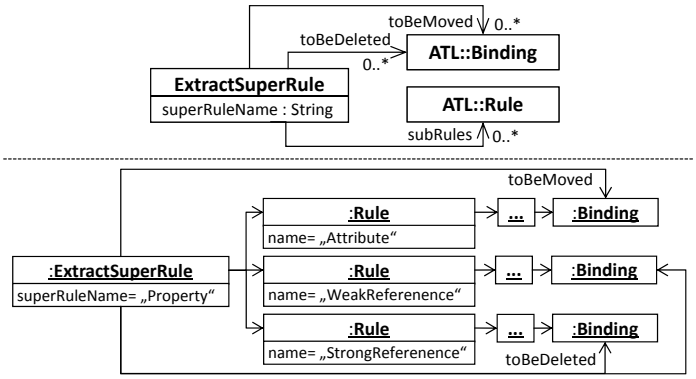


Figure 4 – Parameter Metamodel and Example Model for “Extract Super Rule” refactoring

tion engineer has to select the rules from which a new superrule is extracted as well as the bindings which should be pulled up to the new superrule. By selecting the appropriate elements, the additional input is produced which may be verified by OCL constraints in order to verify the pre-conditions of the refactoring. Please note that the production of the additional input is totally independent from the transformation which is used to perform the refactoring. For future work, we aim at providing dedicated detector rules for refactorings which automatically find the right places for applying refactorings in transformations and produce the additional input models without requiring user interaction.

The refactoring *Extract Superrule* is implemented as is shown in List. 9. The first rule creates an additional abstract rule which gets its name and its subrules from the parameter model (cf. lines 7-17). Furthermore, the InPattern and OutPattern of this rule are calculated from the subRules, but appropriate superclasses have to be found for substituting the more concrete InPatternElement and OutPatternElement types of the subrules (cf. lines 18-21). Subsequently, the bindings to be pulled up of the first subrule are linked to the superrule (cf. line 24). Finally, the bindings to be pulled up of the remaining subrules are deleted by using the drop functionality of the ATL refining mode (cf. lines 28-31).

```
1  module ExtractSuperRule;
2  create OUT : ATL refining IN1 : ATL, IN2 : Params, IN3 : MMs, IN4 : MMt;
3
4  helper def : pars : Params!ExtractSuperRule =
5      Params!ExtractSuperRule.allInstances() -> first();
6
7  rule extractSuperRule{
8    from
9      m : ATL!Module
10   to
11   -- Create New SuperRule
12   newSuperRule: ATL!MatchedRule(
13     isAbstract <- true,
14       name <- thisModule.pars.superRuleName,
15       children <- thisModule.pars.subRules,
16       ...
17     ),
18   -- Copy Input Pattern + type inference based on MMs
19   newInPattern: ATL!InPattern (...)
20   -- Copy Output Pattern + type inference based on MMt
21   newOutPattern: ATL!OutPattern (...),
22   -- Pull up repeated bindings
23   newSimpleOutPatternElm: ATL!SimpleOutPatternElement (
24     bindings <- thisModule.pars.toBeMoved
25   )
26 }
27
28 rule dropBindingsInSubRules{
29     from
30         b : ATL!Binding (thisModule.pars.toBeDeleted->includes(b))
31     to drop
32 }
```

Listing 9 – Excerpt of the Extract SuperRule HOT.

## 8 Related Work

Model transformations have been used to implement refactorings for models but the problem of refactoring model transformations themselves has not been addressed. Implementing refactorings with model transformation technologies has been extensively studied within the last decade. One of the first investigations in this area was done by

Sunyé et al. [SPLTJ01], who define a set of UML refactorings on the conceptual level by expressing pre- and postconditions in OCL. In [MVEDJ05], object-oriented programs are represented as graphs before applying graph transformations for refactoring this abstract representation. Furthermore, Mens [Men05] and Bottoni et al. [BPPT03] use graph transformations to describe refactorings for models. The application of this formalism comes with the additional benefit of formal analysis possibilities of dependencies between different refactorings [MTR07]. Besides graph transformations, also other transformation formalisms have been used for implementing model refactorings, e.g., [Por05], [ZLG05], [KPPR07] to name just a few. Mentioned works have in common that they are focused on implementing refactorings with model transformation formalisms, but not on refactoring model transformations themselves.

There is some dedicated work on refactoring OCL expressions which is of course relevant for refactoring M2M transformations incorporating OCL expressions. For our refactoring catalogue, we are reusing some refactorings/equivalences from previous work [CT07, CW07, GL05] which are applicable for transformations such as *Shorten Navigation By Context Switch*. Our catalogue complements these refactorings by providing transformation-specific refactorings such as *Convert Helper Type* or *Eliminate Unsafe Target Navigation*. Furthermore, we have developed refactorings for retrospectively introducing the optimization patterns of [CJMB08] in existing model transformations such as *Improve Opposite Reference Computations*.

To the best of our knowledge, only one work explicitly mentions refactoring of model transformations. In [EEE09], the authors present how to co-evolve graph transformations in case the metamodels of the models to transform evolve. Nevertheless, this is a totally different notion of refactoring. While we are using the term refactoring for improving the transformation without changing its semantics, in [EEE09] the semantics of the transformation are changed due to the changes in the metamodels. The notion of refactoring used in [EEE09] is not concerned with enhancing the quality of a transformation, but it is more related to adapting the transformations to the new metamodel versions.

## 9 Conclusion and Future Work

This paper has outlined how to improve maintainability of M2M transformations by adopting the notion of refactoring. In particular, we have presented an extensive refactoring catalogue for model transformations, its application to an illustrative example, and discussed reuse possibilities for several model transformation languages. Furthermore, we discussed the impact of the refactorings on the transformations internal qualities and on their execution performance as well as how refactorings are implemented using the in-place transformation mode of ATL. Our results emphasize the great potential of using refactorings for enhancing the quality of model transformations. We hope this work acts as a stimulus for establishing a body of knowledge of transformation engineering in order to accomplish the transition from implicit to explicit knowledge.

We believe the next step should be improving user support and guidance through the refactoring process. For this, the following research challenges have to be addressed.

**Automation**. Currently, the refactorings are performed semi-automatically. Users must manually identify which (and where) transformations should be refactored and determine the most suitable refactorings for them. Once this is done, the refac-

toring(s) is/are automatically applied by executing the corresponding higher-order transformation(s). We plan to develop a set of patterns to identify bad smells in transformations (as done for code refactorings) that help designers to identify potential candidate transformation for refactoring and suggest possible refactorings for them.

**Code layout preservation**. Refactorings are easier to implement on the abstract syntax level of languages (like our transformation models). However, refactorings at this level do not preserve textual aspects of the transformation like the code layout, which is considered important by some transformation designers. In the future, special synchronization mechanisms have to be developed in order to protect the layout aspects of the transformation after the refactoring.

**Language-independence**. As has been discussed in Subsection 4.5, there is a reuse potential of the refactoring catalogue to provide refactoring support for other model transformations as well. Thus, our goal is to make the refactorings as generic as possible both at the specification and implementation level. A more generic (language-independent) definition of M2M transformations would facilitate both. This generic M2M metamodel would be useful in other scenarios as well, e.g., to compare the language features of existing model transformation languages or to act as a pivot metamodel for model transformation exchange between different transformation tools.

## References


[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006. doi:10.1007/11880240_31.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996. doi:10.1109/32.544352.

[Ber93] Edward V. Berard. *Essays on object-oriented software engineering (vol. 1)*. Prentice-Hall, 1993.

[BKSW09] Petra Brosch, Gerti Kappel, Martina Seidl, and Manuel Wimmer. Teaching Model Engineering in the Large. In *Proceedings of the Educators' Symposium @ MoDELS'09*, 2009.

[BPPT03] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying Integrated Refactoring with Distributed Graph Transformations. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Proceedings of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 220–235. Springer, 2003. doi:10.1007/978-3-540-25959-6_16.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006. doi:10.1147/sj.453.0621.

[CJMB08] Jesús Sánchez Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin. Optimization Patterns for OCL-Based Model Transformations. In Michel R. V. Chaudron, editor, *Models in Software Engineering - Reports and Revised Selected Papers of Workshops and Symposia at MoDELS'08*, volume 5421 of *LNCS*, pages 273–284. Springer, 2008. doi:10.1007/978-3-642-01648-6_29.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi:10.1109/32.295895.

[CT06] Jordi Cabot and Ernest Teniente. A metric for measuring the complexity of OCL expressions. In *Proceedings of the Model Size Metrics Workshop @ MoDELS'06*, 2006.

[CT07] Jordi Cabot and Ernest Teniente. Transformation techniques for OCL constraints. *Science of Computer Programming*, 68(3):179–195, 2007. doi:10.1016/j.scico.2007.05.001.

[CW07] Alexandre L. Correa and Cláudia Werner. Refactoring Object Constraint Language Specifications. *Software and System Modeling*, 6(2):113–138, 2007. doi:10.1007/s10270-006-0023-y.

[Dem05] Serge Demeyer. Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls? In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 627–630. IEEE Computer Society, 2005. doi:10.1109/ICSM.2005.74.

[DV92] Paul Doyle and Renaat Verbruggen. Applying Metrics to Rule-Based Systems. In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering (SEKE'92)*, pages 123–130. Knowledge Systems Institute, 1992. doi:10.1109/SEKE.1992.227938.

[EEE09] Hartmut Ehrig, Karsten Ehrig, and Claudia Ermel. Refactoring of Model Transformations. *ECEASST*, 18, 2009.

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

[GL05] Martin Giese and Daniel Larsson. Simplifying Transformations of OCL Constraints. In Lionel C. Briand and Clay Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *LNCS*, pages 309–323. Springer, 2005. doi:10.1007/11557432_23.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Proceedings of Satellite Events at the MoDELS'05 - Revised Selected Papers of International Workshops, Doctoral Symposium, Educators Symposium*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005. doi:10.1007/11663430_14.

[Jon94] Capers Jones. Software Metrics: Good, Bad and Missing. *Computer*, 27:98–100, 1994. doi:10.1109/2.312055.

[KGBH10] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating Maintainability with Code Metrics for Model-to-Model

Transformations. In George T. Heineman, Jan Kofron, and Frantisek Plasil, editors, *Proceedings of the 6th International Conference on the Quality of Software Architectures (QoSA'10)*, volume 6093 of *LNCS*, pages 151–166. Springer, 2010. doi:10.1007/978-3-642-13821-8_12.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008. doi:10.1007/978-3-540-69927-9_4.

[KPPR07] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007. doi:10.5381/jot.2007.6.9.a3.

[Men05] Tom Mens. On the Use of Graph Transformations for Model Refactoring. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *LNCS*, pages 219–257. Springer, 2005. doi:10.1007/11877028_7.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. doi:10.1016/j.entcs.2005.10.021.

[MT04] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007. doi:10.1007/s10270-006-0044-6.

[MVEDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005. doi:10.1002/smr.316.

[OMG11] OMG. *MOF Query/View/Transformation V1.1.* Object Management Group, 2011.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[OXJF05] Jeffrey Overbey, Spiros Xanthos, Ralph Johnson, and Brian Foote. Refactorings for Fortran and high-performance computing. In *Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications*, pages 37–39. ACM, 2005. doi:10.1145/1145319.1145331.

[Por05] Ivan Porres. Rule-based Update Transformations and their Application to Model Refactorings. *Software and System Modeling*, 4(4):368–385, 2005. doi:10.1007/s10270-005-0088-z.

[RGP04] Luis Reynoso, Marcela Genero, and Mario Piattini. Towards a metric suite for OCL Expressions expressed within UML/OCL models. *Journal of Comptuer Science and Technology*, 4:38–44, 2004.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003. doi:10.1109/MS.2003.1231150.

[SPLTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001. doi:10.1007/3-540-45441-1_11.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009. doi:10.1007/978-3-642-02674-4_3.

[TMJC11] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Refining Models with Rule-based Model Transformations. Technical report, AtlanMod, INRIA & École des Mines de Nantes, 2011.

[TV11] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:1–29, 2011. doi:10.5381/jot.2011.10.1.a5.

[vALvdB09] Marcel van Amstel, Christian F. J. Lange, and Mark van den Brand. Using Metrics for Assessing the Quality of ASF+SDF Model Transformations. In Richard F. Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, volume 5563 of *LNCS*, pages 239–248. Springer, 2009. doi:10.1007/978-3-642-02408-5_17.

[vAvdB11] Marcel van Amstel and Mark van den Brand. Using Metrics for Assessing the Quality of ATL Model Transformations. In *Proceedings of the Workshop on Model Transformation with ATL (MtATL) @ ICMT'11*, volume 742, pages 20–34. CEUR, 2011.

[WKK+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Denis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 31–46. Springer, 2011. doi:10.1007/978-3-642-21732-6_3.

[ZLG05] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development—Research and Practice in Software Engineering*, pages 199–217. Springer, 2005. doi:10.1007/3-540-28554-7_9.

## About the authors

**Manuel Wimmer** is working as a post-doc researcher at the Business Informatics Group of the Vienna University of Technology. His research interests comprise Web engineering and model engineering; in particular model transformations based on formal methods, generating transformations by-example as well as applying model transformations to deal with model (co-)evolution. Currently, he is on leave working as visiting researcher at the Software Engineering Group of the University of Málaga (Spain). For further information about his research activities, please visit `http://www.big.tuwien.ac.at/staff/mwimmer` or contact him at `wimmer@big.tuwien.ac.at`.

**Salvador Martínez** is a PhD candidate at the AtlanMod team of the École des Mines de Nantes, France. His research interests include model-driven security, model-driven reverse engineering (particularly of security-related aspects) and model transformation languages. Contact him at `salvador.martinez_perez@inria.fr`, or visit `http://www.emn.fr/z-info/atlanmod/index.php/User:SMartinez`.

**Frédéric Jouault** is a researcher in the AtlanMod team. He received his Ph.D. in September 2006 from the University of Nantes. He did a postdoc at the University of Alabama at Birmingham in 2007. His research interests involve model engineering, model transformation, and their application to Domain-Specific Languages (DSLs) and model-based legacy reverse engineering. Frédéric created ATL (AtlanMod Transformation Language), a DSL for model-to-model transformation. He is now leading the development of ATL (language and toolkit) on Eclipse.org. He is in charge of the Eclipse modeling M2M project as well as a member of the modeling PMC. Contact him at `frederic.jouault@inria.fr`.

**Jordi Cabot** is currently leading the AtlanMod team, an INRIA research group at École des Mines de Nantes (France). Previously, he has been a post-doctoral fellow at the University of Toronto, a senior lecturer at the UOC (Open University of Catalonia) and a visiting scholar at the Politecnico di Milano. He received the BSc and PhD degrees in Computer Science from the Technical University of Catalonia. His research interests include conceptual modeling, model-driven and web engineering, formal verification and social aspects of software engineering. He has written more than 70 publications in international journals and conferences in the area. Apart from his scientific publications, he writes and blogs about all these topics in his Modeling Languages portal (`http://modeling-languages.com`). Contact him at `jordi.cabot@inria.fr`, or visit `http://jordicabot.com/`.

**Acknowledgments** We would like to thank Marcel F. van Amstel for providing us the ATL2Metrics transformations which have been used for the computation of metric values for the different versions of the example transformation.

This work has been partially funded by the Austrian Science Fund (FWF) under grant J 3159-N23 and by the OPEES ITEA2 European project.