A kernel transformation language for metamodel evolution and reversible model co-evolution

Mickaël Kerboeuf, Paola Vallejo, Jean-Philippe Babau

To cite this version:

Mickaël Kerboeuf, Paola Vallejo, Jean-Philippe Babau. A kernel transformation language for metamodel evolution and reversible model co-evolution. 2013. hal-00842789

HAL Id: hal-00842789
https://hal.univ-brest.fr/hal-00842789
Submitted on 9 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A kernel transformation language for metamodel evolution and reversible model co-evolution

Mickaël Kerboeuf, Paola Vallejo, and Jean-Philippe Babau

University of Brest (France), Lab-STICC, MOCS Team
{kerboeuf,vallejoco,babau}@univ-brest.fr

Abstract. This report defines \( \mu \text{Dif} \), a kernel transformation language for metamodel evolution and reversible model co-evolution. To begin with, a kernel subset of \( \text{Ecore} \) is highlighted and formally defined thanks to a suitable denotational semantics. Then \( \mu \text{Dif} \) is formally defined upon this subset. In a first step, the focus is put on metamodel evolution provided by a set of refactoring operators. In a second step, the focus is put on model co-evolution which is intended to be reversible thanks to a dedicated pair of transformations respectively called migration and recontextualization. Each \( \mu \text{Dif} \) operator is also provided with a dedicated predicate which explains the sufficient conditions for a model to remain valid after these transformations.

1 \( \mu \text{Ecore} \)

Figure 1 depicts the metamodel of \( \text{Ecore} \) [1]. We focus on concepts whose refactoring have side-effects on instances. Thus, we do not take into account sub-packages\(^1\) and we consider the absolute name\(^2\) of an \( \text{Ecore} \) classifier as its actual identifying name. Operations and annotations are discarded as well. We also put several features out of the scope of the transformations we target. For instance, we do not distinguish between primitive data types and enumerations. And finally, many properties of \( \text{Ecore} \) concepts like uniqueness or order for attributes are discarded. In the end, we obtain the simplified version of \( \text{Ecore} \) we called \( \mu \text{Ecore} \), and whose metamodel is depicted by figure 2.

1.1 Textual syntax of \( \mu \text{Ecore} \)

In order to make easier the formal definition of \( \mu \text{Dif} \), we introduce in figure 3 a feather light textual syntax for metamodels conforming to the \( \mu \text{Ecore} \) metamodel of figure 2.

A metamodel is an unordered set of data types and classes. A data type is only defined by its name. A class is defined by its name, and three optional features. The first one (a) specifies an abstract class. The second one is a set of inherited classes names. The last one is an unordered set of attributes and references. An

\(^1\) Only a unique root package is needed.\(^2\) i.e. the complete name including the ordered sequence of nested packages’ names.
Fig. 1. Ecore metamodel
\( m, n_{\text{class}}, n_{\text{type}}, n_{\text{attrib}}, n_{\text{ref}} \in \mathbb{N} \)  
\( m \in \mathcal{M} = \{(x, y) \in \mathbb{N} \times (\mathbb{N}^* \cup \{\infty\}) | x < y \} \)  

\[ m m := (c \mid d)^* \]  
\[ c := (\langle a \rangle^* n (\mid n_{\text{class}})^* (a \mid r)^* )_c \]  
\[ d := (n)_d \]  
\[ a := [n, m, n_{\text{type}}]_a \]  
\[ r := [c]^* n, m, n_{\text{class}} (\leftarrow n_{\text{ref}})^* ]_r \]

**Fig. 2. \( \mu \text{Ecore} \) metamodel**

\( \langle \text{Int} \rangle_d \langle \text{Bool} \rangle_d \)  
\( \langle a \text{ A } [i, (0, 1), \text{Int}]_a [b, (0, \infty), \text{Bool}]_a [y, (0, 1), Y \leftarrow a]_r \rangle_c \)  
\( \langle X / A [j, (1, 1), \text{Int}]_a \rangle_c \)  
\( \langle Y / A [c, a, (1, 2), A \leftarrow y]_r [x, (0, \infty), X]_r \rangle_c \)

**Fig. 3. Textual syntax of \( \mu \text{Ecore} \)**

**Fig. 4. Textual and graphical form of a \( \mu \text{Ecore} \) metamodel**
attribute is defined by its name, its multiplicity and the name of its data type. A reference is defined by its name, its multiplicity and the name of the class it refers to. Two last optional features specify a potential containment (c) and a potential opposite reference (←).

As an illustration, figure 4 shows a metamodel conforming to \(\mu\text{Ecore}\) together with its equivalent textual specification.

1.2 Denotational semantics of \(\mu\text{Ecore}\)

The formal semantics of \(\mu\text{Ecore}\) is defined by a mapping between the language constructs and a semantic domain including sets and partial functions.

**Semantic domain** Figure 5 defines the name spaces of the semantic domain. They correspond to alphabets, i.e., finite non empty sets of symbols.

\[
\begin{align*}
\mathcal{N} & : \text{named elements} \\
\mathcal{C} & \subseteq \mathcal{N} : \text{classes} \\
\mathcal{D} & \subseteq \mathcal{N} \setminus \mathcal{C} : \text{data types} \\
\mathcal{R} & \subseteq \mathcal{C} \times \mathcal{N} : \text{references} \\
\mathcal{A} & \subseteq (\mathcal{C} \times \mathcal{N}) \setminus \mathcal{R} : \text{attributes}
\end{align*}
\]

Fig. 5. \(\mu\text{Ecore}\) semantics: name spaces

Figure 6 defines multiplicity, which is by definition a set of pairs composed of natural numbers extended with the special mark \(\infty^3\). This definition excludes the irrelevant multiplicities \((0, 0)\) and \((\infty, \infty)\).

\[
\mathcal{M} \triangleq \{(x, y) \in \mathbb{N} \times (\mathbb{N}^* \cup \{\infty\})| x < y\} \quad \text{multiplicity}
\]

Fig. 6. \(\mu\text{Ecore}\) semantics: multiplicity

Figure 7 defines the main denotations of the semantic domain, namely partial functions. They allow to gather classes, data types, attributes and references according to inheritance and structural links.

By definition, a metamodel \(\mathfrak{m} \in \mathfrak{M}\) is a pair of name spaces \(\mathfrak{n} \in \mathfrak{N}\) and partial functions \(f \in \mathfrak{F}\) (see figure 8). Name spaces are given by a sequence of 6 sets (including 2 subsets). Partial functions are given by a sequence of 4 partial functions whose definition domains are the name spaces of the metamodel. This definition equates the sets of partial functions to their corresponding power sets.

\[\text{by definition, } \forall n \in \mathbb{N}, n < \infty\]
\[ \delta_I : C \rightarrow C \] inheritance link
\[ \delta_A : A \rightarrow M \times D \] attribute link
\[ \delta_R : R \rightarrow M \times C \] reference link
\[ \delta_{\text{opp}} : R \rightarrow R \] opposite link

**Fig. 7.** \( \muEcore \) semantics: partial functions

\[
\begin{align*}
\mathcal{M} & \triangleq \mathcal{N} \times \mathcal{\delta} \\
\mathcal{N} & \triangleq \mathcal{P}(C) \times \mathcal{P}(C_A) \times \mathcal{P}(D) \times \mathcal{P}(R) \times \mathcal{P}(R_C) \times \mathcal{P}(A) \\
\mathcal{\delta} & \triangleq \mathcal{P}(\delta_I) \times \mathcal{P}(\delta_A) \times \mathcal{P}(\delta_R) \times \mathcal{P}(\delta_{\text{opp}})
\end{align*}
\]

**Fig. 8.** \( \muEcore \) semantics: metamodels

**Valuation function**

*Notation* In order to define efficiently the valuation function of \( \muEcore \), we first introduce the following notation on a given *parsed* metamodel \( mm \):

\[ \text{pattern} \sqsubseteq mm \]

This notation stands for a *proposition* stating that a *class* or a *data type* matching with the given *pattern* can be found in \( mm \). This pattern corresponds to what can be derived from \( c \) or \( d \) according to the syntax specified by figure 3.

More formally, if we note \( \mathcal{L}(c) \) and \( \mathcal{L}(d) \) the sets of words respectively yielded from \( c \) and \( d \) in figure 3, then by definition: \( \text{pattern} \in \mathcal{L}(c) \cup \mathcal{L}(d) \).

For instance, the following proposition states that an *abstract class* named \( n_1 \) appears among the parsed elements of \( mm \), and this class has an attribute named \( n_2 \) of type \( n_3 \) and multiplicity \((0,1)\):

\[ \langle \text{a} \ n_1\ [n_2,(0,1),n_3]_a \rangle_c \sqsubseteq mm \]

As an illustration, this proposition is *true* with the following example of metamodel:

\[ mm \triangleq (\langle n_3 \rangle_d \langle \text{a} \ n_1 \ / \ n_4\ [n_2,(0,1),n_3]_a \ [n_r,(0,\infty),n_4]_r \rangle_c \langle \text{a} \ n_4 \rangle_c ) \]

**Valuation** We note \( \mathcal{L}(mm) \) the set of words yielded from \( mm \) in figure 3. Figure 9 shows the definition of the valuation function. Its maps \( \muEcore \) to the semantic domain \( \mathcal{M} \) defined by figure 8.

**1.3 Example**

Let \( x \) be the metamodel of figure 4. Its denotation is given by \( \| x \|_{mm} = m = (n,f) \).

Figure 10 shows \( m \) in details, and figure 11 shows a *graph-based* representation of it. In this representation, the different name spaces \( n \) corresponding to classes, data types, attributes and references are depicted by four kinds of dedicated vertices.
\[ \llbracket \cdot \rrbracket_{mm} : \mathcal{L}(mm) \to \mathfrak{M} \]
\[ x \mapsto [x]_{mm} = (\mathcal{C}, \mathcal{C}_A, D, \mathcal{R}, \mathcal{R}_C, A), (\delta_I, \delta_A, \delta_R, \delta_{app}) \]  
where:

\[ \mathcal{C} = \{ n \in \mathcal{N} \mid (n) \in x \} \]
\[ \mathcal{C}_A = \{ n \in \mathcal{N} \mid (a n) \in x \} \]
\[ D = \{ n \in \mathcal{N} \mid (n) \in x \} \]
\[ \mathcal{R} = \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, \exists n_3 \in \mathcal{N}, (n_1 [n_2, m, n_3] \in M \cap \mathcal{C} \} \]
\[ \mathcal{R}_C = \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, \exists n_3 \in \mathcal{N}, (n_1 \in \mathcal{N} \cap \mathcal{C} \} \]
\[ A = \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, (n_1 [n_2, m, n_3] \in M \cap \mathcal{C} \} \]
\[ \delta_I = \{(n_1, n_2) \in \mathcal{N}^2 \mid (n_1, n_2) \in x \} \]
\[ \delta_A = \{(n_1, n_2), (m, n_3) \in \mathcal{A} \times (\mathcal{M} \times \mathcal{D}) \mid (n_1 [n_2, m, n_3] \in M \cap \mathcal{C} \} \]
\[ \delta_R = \{(n_1, n_2), (m, n_3) \in \mathcal{R} \times (\mathcal{M} \times \mathcal{C}) \mid (n_1 [n_2, m, n_3] \in M \cap \mathcal{C} \} \]
\[ \delta_{app} = \{(n_1, n_2), (m, n_3) \in \mathcal{R}^2 \mid \exists m \in \mathcal{M}, (n_1 [n_2, m, n_3] \in M \cap \mathcal{C} \} \]

**Fig. 9.** \(\mu\text{Ecore}\) semantics: valuation

\[ [x]_{mm} = m = (n, j) = (\mathcal{C}, \mathcal{C}_A, D, \mathcal{R}, \mathcal{R}_C, A), (\delta_I, \delta_A, \delta_R, \delta_{app}) \]  
where:

\[ \mathcal{C} = \{A, X, Y\} \]
\[ \mathcal{C}_A = \{A\} \]
\[ D = \{\text{Bool, Int}\} \]
\[ \mathcal{R} = \{(Y, a), (X, j), (A, y)\} \]
\[ \mathcal{R}_C = \{(Y, a)\} \]
\[ A = \{(A, i), (A, b), (X, j)\} \]
\[ \delta_I = \{X \mapsto A, Y \mapsto A\} \]
\[ \delta_A = \{(A, i) \mapsto ((0, 1), \text{Int}), (A, b) \mapsto ((0, \infty), \text{Bool}), (X, j) \mapsto ((1, 1), \text{Int})\} \]
\[ \delta_R = \{(Y, a) \mapsto ((0, 2), A), (Y, x) \mapsto ((0, \infty), X), (A, y) \mapsto ((0, 1), Y)\} \]
\[ \delta_{app} = \{(A, y) \mapsto (Y, a), (Y, a) \mapsto (A, y)\} \]

**Fig. 10.** Semantics of a \(\mu\text{Ecore}\) metamodel

**Fig. 11.** Graph view of the semantics of a \(\mu\text{Ecore}\) metamodel
Edges represent (and are labeled by) the partial functions of \( f \). For instance, \((n_1 \mapsto n_2) \in \delta_l\) is represented by an edge from a class vertex \( n_1 \) to a class vertex \( n_2 \). This edge is labeled by \( \delta_l \).

Partial functions \( \delta_A \) and \( \delta_R \) are represented by pairs of edges. For instance, \(((n_1, n_2) \mapsto (m, n_3)) \in \delta_A\) is represented by an edge from a class vertex \( n_1 \) to an attribute vertex \( n_2 \), followed by an edge from the same attribute vertex \( n_2 \) to a data type vertex \( n_3 \). The first edge is labeled by \( \delta_A \). The second one is labeled by \( m \).

Partial functions \( \delta_{opp} \) are represented by edges between reference vertices. For instance, \(((n_1, n_2) \mapsto (n_3, n_4)) \in \delta_{opp}\) is represented by an edge from a vertex \( n_2 \) to a vertex \( n_4 \). This edge is labeled by \( \delta_{opp} \). By construction, vertices \( n_2 \) and \( n_4 \) are themselves respectively linked to class vertices \( n_1 \) and \( n_3 \) by means of edges labeled by \( \delta_R \).

2 Metamodel evolution with \( \mu \text{Dif} \)

\( \mu \text{Dif} \) is a refactoring language whose scope encompass metamodels conforming to \( \mu \text{Ecore} \). It is intended to perform reversible model migration together with metamodel refactoring. In this section, we focus on metamodel refactoring.

2.1 Overview of \( \mu \text{Dif} \)

\( \mu \text{Dif} \) is basically a metamodel refactoring language gathering CRUD operations. A \( \mu \text{Dif} \) specification is an ordered sequence of refactoring operators applied to an input \( \mu \text{Ecore} \) metamodel. Therefore, each \( \mu \text{Dif} \) operator has an implicit parameter corresponding to a \( \mu \text{Ecore} \) metamodel. We name context and note \( mm \) this metamodel associated to each \( \mu \text{Dif} \) operator.

\[
\begin{align*}
\text{spec} &::= \{ \text{mm} \} \ (\text{op})^+ \quad \text{(specification)} \\
\text{op} &::= \text{cr} \mid \text{u} \mid \text{d} \quad \text{(operator)}
\end{align*}
\]

Fig. 12. Textual syntax of \( \mu \text{Dif} \)

Figure 12 is a partial view of the syntax of \( \mu \text{Dif} \). It extends the textual syntax of \( \mu \text{Ecore} \) defined by figure 2.

A specification is a context (defined by a metamodel \( mm \)) followed by a non-empty ordered sequence of operators. Operators are divided into three CRUD categories, namely create (cr), update (u) and delete (d). The next sections present these categories in details. A following section presents the formal semantics of \( \mu \text{Dif} \) in regard to the semantic domains we introduced for \( \mu \text{Ecore} \).
2.2 μDif creation

Creation is related to concepts that are depicted by black classes in figure 2. The concept of EPackage (depicted in grey) is supposed to be instantiated once (the root package) and it remains out of the scope of refactoring.

Figure 13 outlines the creation operators. It completes the syntax of μDif introduced in figure 12.

\[
\text{cr} ::= \text{createClass}(n) \quad \text{(creation)} \\
\quad \text{createDataType}(n) \\
\quad \text{createAttribute}(n, n_{\text{class}}, n_{\text{type}}) \\
\quad \text{createReference}(n, n_{\text{class}}, n_{\text{targetedClass}})
\]

Fig. 13. μDif creation

Each of these operators is parameterized by a name \( n \) which is supposed to be new among the named element of the corresponding context. For a class or a data type, this context corresponds to the whole metamodel. For an attribute or a reference, this context corresponds to the containing class plus all its ancestors in regard to inheritance. More precisely:

- \( \text{createClass}(n) \) creates in \( mm \) a new concrete class without super classes and without features; precondition: \( n \) does not already appear in the context \( mm \).
- \( \text{createDataType}(n) \) creates in \( mm \) a new data type; precondition: \( n \) does not already appear in the context \( mm \).
- \( \text{createAttribute}(n, n_{\text{class}}, n_{\text{type}}) \) creates an attribute with default multiplicity 0..1; this attribute is attached to the class of \( mm \) identified by \( n_{\text{class}} \) and it is typed by the data type named \( n_{\text{type}} \) in \( mm \); precondition: \( n \) does not already appear among the features associated to \( n_{\text{class}} \) nor to any of its ancestors.
- \( \text{createReference}(n, n_{\text{class}}, n_{\text{targetedClass}}) \) creates a new reference with default multiplicity 0..1; this reference is attached to the class of \( mm \) identified by \( n_{\text{class}} \) and it targets the class named \( n_{\text{targetedClass}} \) in \( mm \); precondition: \( n \) does not already appear among the features associated to \( n_{\text{class}} \) nor to any of its ancestors.

2.3 μDif deletion

As for creation, deletion is related to concepts that are depicted by black classes in figure 2.

Figure 14 outlines the deletion operators. It completes the syntax of μDif introduced in figure 12.

Each of these operators is parameterized by a name which is supposed to be related to an existing element of the corresponding context (i.e. the whole metamodel or a class).

A common precondition states that the element to be deleted must not be targeted by any other element. More precisely:
Fig. 14. μDif deletion

- deleteClass(n_class) deletes the class (and its features) identified by n_class in mm; precondition: n_class is not a super class and it is not targeted by any reference.
- deleteDataType(n_type) deletes the data type identified by n_type in mm; precondition: n_type is not targeted by any attribute.
- deleteAttribute(n_attrib, n_class) deletes the attribute named n_attrib in the class of mm named n_class; no precondition
- deleteReference(n_ref, n_class) deletes the reference named n_ref in the class of mm named n_class; precondition: n_ref is not targeted by an opposite reference.

2.4 μDif update

There are three categories of update operators: value updates (i.e., updates of values conforming to meta-attributes), containment updates (i.e., updates of links conforming to meta-compositions), and link updates (i.e., updates of links conforming to non-composite meta-references).

Figure 15 outlines the update operators. It completes the syntax of μDif introduced in figure 12 and it introduces a syntactic root node for the following grammar complements.

Fig. 15. μDif update

Value update Value update is related to the meta-attributes name, lowerBound, upperBound, abstract and container in figure 2. A set operator is associated to each of them.

Figure 16 outlines the value update operators. It completes the grammar rule introduced in figure 15.

Each of these operators is parameterized by a name and the new value of the corresponding meta-attribute. Most of them are subjected to specific preconditions over the new values. More precisely:

- setName( (n | n_class, n ) , n_new ) sets n_new as the new name of n (class or data type) or of the feature n (attribute or reference) of class n_class; precondition: the context mm (or the class n_class) does not already embed a classifier (or a feature) with the same name n_new.
\[ 
\text{vu} :::= \text{name}(n | n_{\text{class}}.n), n_{\text{new}}) \text{ (value update)} \\
| \text{setLowerBound}(n_{\text{class}}, n, i) \\
| \text{setUpperBound}(n_{\text{class}}, n, i) \\
| \text{setAbstract}(n_{\text{class}}, b) \\
| \text{setContainer}(n_{\text{class}}, n, b) 
\]

\textbf{Fig. 16. \mu\text{Dif} value update}

- \text{setLowerBound}(n_{\text{class}}, n, i) sets a new lower bound \(i\) for the feature \(n\) (attribute or reference) of class \(n_{\text{class}}\); precondition: \(i\) is lower than or equal to the associated upper bound.
- \text{setUpperBound}(n_{\text{class}}, n, i) sets a new upper bound \(i\) for the feature \(n\) (attribute or reference) of class \(n_{\text{class}}\); precondition: \(i\) is upper than or equal to the associated lower bound.
- \text{setAbstract}(n_{\text{class}}, b) makes class \(n_{\text{class}}\) abstract or concrete according to the boolean value \(b\); no precondition.
- \text{setContainer}(n_{\text{class}}, n, b) makes reference \(n\) of class \(n_{\text{class}}\) a composition or a simple reference according to the boolean value \(b\); no precondition.

\textbf{Containment update} Containment update is related to targets of compositions in the metametamodel. In the metamodel of \(\mu\text{Ecore}\) (fig. 2), it corresponds to a unique element, namely \(\text{ETypedElement}\). Its container is mandatory. Therefore, there is only one relevant refactoring operator associated to this element. It allows to move it from its current container to another one.

\[ 
\text{cu} :::= \text{moveFeatureTo}(n_{\text{class}}, n, n_{\text{new}}) \text{ (containment update)} 
\]

\textbf{Fig. 17. \mu\text{Dif} containment update}

Figure 17 introduces the syntax of this unique \textit{containment update} operator. It completes the grammar rule introduced in figure 15.

This operator is parameterized by the new class where an existing feature has to be moved. A precondition prevents name clashes:

- \text{moveFeatureTo}(n_{\text{class}}, n, n_{\text{new}}) moves the feature \(n\) (attribute or reference) of class \(n_{\text{class}}\) to another class of \(mm\) named \(n_{\text{new}}\); precondition: \(n\) does not already appear among the features associated to \(n_{\text{new}}\) and all its ancestors.

Adding or removing a feature (attribute or reference) actually consist in creating or deleting this feature. These operations are already provided by \textit{creation} and \textit{deletion operators} (see figures 13 and 14).

\textbf{Link update} Link update is related to targets of non-composite references in the metametamodel. In the metamodel of \(\mu\text{Ecore}\) (fig. 2), it corresponds to \(\text{eAt-}
tributeType, eReferenceType, eSuperType and eOpposite. The relevant refactoring operators associated to these targets depends on their multiplicity.

Links conforming to references whose target’s multiplicity is 1 are related to elements that can only be moved. Links conforming to references whose target’s multiplicity is 0..* are related to collections from which elements can be added, moved or removed. Finally, links conforming to references whose target’s multiplicity is 0..1 are related to optional elements that can be set, replaced or removed.

Figure 18 outlines these three categories of link update operators. It completes the grammar rule introduced in figure 15 and it introduces a syntactic root node for the three sub-categories of link update.

\[
lu ::= mlu \mid clu \mid olu \quad \text{(link update)}
\]

\[
mlu ::= \text{moveReferenceTargetTo}(n_{class}, n_{oClass}) \quad \text{(mandatory link update)}
\mid \text{moveAttributeTypeTo}(n_{class}, n_{type})
\]

\[
clu ::= \text{addSuperType}(n_{class}, n_{oClass}) \quad \text{(collection link update)}
\mid \text{removeSuperType}(n_{class}, n_{sClass})
\mid \text{moveSuperTypeTo}(n_{class}, n_{sClass}, n_{oClass})
\]

\[
olu ::= \text{moveOppositeTo}(n_{class}, n_{oClass}, n_{ref}) \quad \text{(optional link update)}
\mid \text{removeOpposite}(n_{class}, n_{t})
\]

Fig. 18. μDif link update

In the case of optional link update, the same operator moveOppositeTo allows both to replace an existing opposite reference and add a new opposite reference.

Multiplicity 1 In the metamodel of μEcore (fig. 2), non-composite references whose target’s multiplicity is 1 are eAttributeType and eReferenceClass:

- moveReferenceTargetTo\((n_{class}, n, n_{oClass})\) moves the reference target of \(n\) to \(n_{oClass}\); if an opposite reference exists, it is updated as well; precondition: if an opposite reference exists, it has not the same name a any direct or inherited feature of \(n_{oClass}\).
- moveAttributeTypeTo\((n_{class}, n, n_{type})\) changes the type of \(n\) to \(n_{type}\); no precondition.

Multiplicity 0..* In the metamodel of μEcore (fig. 2), the only non-composite reference whose target’s multiplicity is 0..* is eSuperType:

- addSuperType\((n_{class}, n_{oClass})\) adds \(n_{oClass}\) to the set of classes inherited by \(n_{class}\); precondition: \(n_{class}\) is not an ancestor of \(n_{oClass}\).
- removeSuperType\((n_{class}, n_{sClass})\) removes \(n_{sClass}\) from the set of classes inherited by \(n_{class}\); no precondition.
- `moveSuperTypeTo(n_{class}, n_{sClass}, n_{oClass})` replaces `n_{sClass}` by `n_{oClass}` among the set of classes inherited by `n_{class}`; precondition: `n_{class}` is not an ancestor of `n_{oClass}` and `n_{sClass}` is actually a direct super class of `n_{class}`.

**Multiplicity 0..1** In the metamodel of μEcore (fig. 2), the only non-composite reference whose target’s multiplicity is 0..* is eOpposite:

- `moveOppositeTo(n_{class}, n_{ref}, n_{oClass})` moves the opposite target of `n` to the reference `n_{ref}` of class `n_{class}`; precondition: `n_{ref}` does not already have an opposite reference.
- `removeOpposite(n_{class}, n)` removes the opposite target of `n`; no precondition.

<table>
<thead>
<tr>
<th>μDif operators x</th>
<th>denotation</th>
<th>$x^\top_{\text{op}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>create</strong></td>
<td>createClass($n$)</td>
<td>$cc : \mathbb{M} \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>createDataType($n$)</td>
<td>$cdt : \mathbb{M} \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>createAttribute($n$, $n_{class}$, $n_{type}$)</td>
<td>$ca : \mathbb{M} \times \mathbb{N}^3 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>createReference($n$, $n_{class}$, $n_{targetClass}$)</td>
<td>$cr : \mathbb{M} \times \mathbb{N}^3 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>deleteClass($n_{class}$)</td>
<td>$dc : \mathbb{M} \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>deleteDataType($n_{type}$)</td>
<td>$ddt : \mathbb{M} \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>deleteAttribute($n_{attrib}$, $n_{class}$)</td>
<td>$da : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>deleteReference($n_{ref}$, $n_{class}$)</td>
<td>$dr : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>setName($n$, $n_{new}$)</td>
<td>$sn_c : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>setName($n_{class}$, $n_{new}$)</td>
<td>$sn_f : \mathbb{M} \times \mathbb{N}^3 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>setLowerBound($n_{class}$, $n$, $i$)</td>
<td>$slb : \mathbb{M} \times \mathbb{N}^2 \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>setUpperBound($n_{class}$, $n$, $i$)</td>
<td>$sub : \mathbb{M} \times \mathbb{N}^2 \times \mathbb{N} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>setAbstract($n_{class}$, $b$)</td>
<td>$sa : \mathbb{M} \times \mathbb{N} \times \mathbb{B} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>setContainer($n_{class}$, $n$, $b$)</td>
<td>$sc : \mathbb{M} \times \mathbb{N}^2 \times \mathbb{B} \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>moveFeatureTo($n_{class}$, $n_{ref}$)</td>
<td>$mft : \mathbb{M} \times \mathbb{N}^4 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td><strong>link</strong></td>
<td>moveReferenceTargetTo($n_{class}$, $n_{ref}$)</td>
<td>$mrt : \mathbb{M} \times \mathbb{N}^+ \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>moveAttributeTypeTo($n_{class}$, $n_{type}$)</td>
<td>$matt : \mathbb{M} \times \mathbb{N}^3 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>addSuperType($n_{class}$, $n_{parent}$)</td>
<td>$asc : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>removeSuperType($n_{class}$, $n_{parent}$)</td>
<td>$rsc : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>moveSuperTypeTo($n_{class}$, $n_{parent}$, $n_{parent}$)</td>
<td>$msct : \mathbb{M} \times \mathbb{N}^3 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>moveOppositeTo($n_{class}$, $n_{ref}$)</td>
<td>$mot : \mathbb{M} \times \mathbb{N}^4 \rightarrow \mathbb{M}$</td>
</tr>
<tr>
<td></td>
<td>removeOpposite($n_{class}$, $n$)</td>
<td>$ro : \mathbb{M} \times \mathbb{N}^2 \rightarrow \mathbb{M}$</td>
</tr>
</tbody>
</table>

Fig. 19. Valuation of μDif operators

### 2.5 μDif semantics

The denotational semantics of μDif is based upon the semantics domain of μEcore noted $\mathbb{M}$ and introduced in figure 8.
We note \( L(\text{op}) \) the sets of words yielded from \( \text{op} \) in figure 12. Basically, \( L(\text{op}) \) contains 22 refactoring operators.

We introduce a valuation function noted \([\text{\cdot}]_\text{op}\). It applies to \( L(\text{op}) \) and it maps each operator to a dedicated function whose domain is a tuple including \( \mathcal{M} \), and whose codomain is \( \mathcal{M} \).

Informally said, a \( \mu\text{Dif} \) operator is described by a function from metamodels (plus specific parameters) to metamodels. Figure 19 gathers the \( \mu\text{Dif} \) operators together with their corresponding functional denotations.

### 2.6 Notations

The functions of figure 19 are detailed in the following paragraphs. In each case, the first parameter is noted \( m \in \mathcal{M} \). It corresponds to the input metamodel. By definition, \( m \) corresponds to the following pair:

\[
m \triangleq \left( (\mathcal{C},\mathcal{C}_A,\mathcal{D},\mathcal{R},\mathcal{R}_C,\mathcal{A}), (\delta_I,\delta_A,\delta_R,\delta_{opp}) \right)
\]

**Metamodel component** We note \( m.x \) the \( x \) component of \( m \) (e.g. \( m.\delta_I \)).

**Union of metamodel component** We note \( m.(x \cup y) \) the union of components \( x \) and \( y \) of \( m \) (e.g. \( m.(\mathcal{C} \cup \mathcal{D}) \)):

\[
m.(x \cup y) \triangleq m.x \cup m.y
\]

**Substitution of metamodel component** We note \( m.[x = y] \) the metamodel \( m \) where \( y \) has been substituted to the \( x \) component of \( m \) (e.g. \( m.[\delta_I = \{\ldots\}] \)). If \( y \) is an expression including components of \( m \), then the explicit mention of \( m \) is not needed (e.g. \( m.[\delta_I = \mathcal{C} \cup \{\ldots\}] \) instead of \( m.[\delta_I = m.\mathcal{C} \cup \{\ldots\}] \)).

**General substitution** Let \( S \) be a given set. Knowing \( (a,b) \in S^2 \), we note \( m[a/b] \) the metamodel \( m \) where \( a \) has been substituted to each occurrence of \( b \) (in each component of \( m \)). Examples:

- \( \forall (a,b) \in \mathcal{N}^2, m[a/b] \triangleq m \) where each occurrence of \( b \) has been replaced by \( a \) (regardless the kind of elements that are named \( b \)).
- \( \forall (a,b,n_1,n_2) \in \mathcal{N}^4, m[(a,n_1)/(b,n_2)] \triangleq m \) where the pair \( (a,n_1) \) (i.e. attribute or reference \( n_1 \) of class \( a \)) has been substituted to each occurrence of the pair \( (b,n_2) \) (in each component of \( m \)).
- \( \forall (a,b,a',b') \in \mathcal{N}^4, m[(a' \mapsto b')/(a \mapsto b)] \triangleq m \) where the mapping between \( a' \) and \( b' \) has been substituted to each occurrence of the mapping between \( a \) and \( b \) (here in each functional component of \( m \)).
**Classifier substitution** Knowing \((a, b) \in \mathcal{N}^2\), we note \(m[a/b]_c\) the *metamodel* \(m\) where the *classifier name* \(a\) (*i.e.* class name or data type name) has been *substituted* to each occurrence of the classifier name \(b\) (in each component of \(m\)). This notation is a restriction of the previous one. It allows to specifically target classifiers. Example:

\[- \forall (a, b) \in \mathcal{N}^2, m[a/b]_c \triangleq m \text{ where each occurrence of } b \text{ has been replaced by } a. \text{ *Features* named } b \text{ are out of the scope of this substitution.}\]

**Direct ancestors** We note \(\Delta_I(c)\) the set of direct ancestors of class \(c\) in regard to *inheritance*:

\[
\Delta_I : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C})
\]
\[
c \mapsto \{c' \in \mathcal{C} \mid \delta_I(c) = c'\}
\]

We note \(\Delta'_I(c)\) the set of direct ancestors of class \(c\), extended by class \(c\) itself:

\[
\forall c \in \mathcal{C}, \Delta'_I(c) \triangleq (\Delta_I(c) \cup \{c\}).
\]

**All ancestors** We note \(\alpha_I(c)\) the set of all ancestors of class \(c\) in regard to *inheritance*:

\[
\alpha_I : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C})
\]
\[
c \mapsto \bigcup_{c' \in \Delta_I(c)} (\{c'\} \cup \alpha_I(c'))
\]

We note \(\alpha'_I(c)\) the set of all ancestors of class \(c\), extended by class \(c\) itself:

\[
\forall c \in \mathcal{C}, \alpha'_I(c) \triangleq (\alpha_I(c) \cup \{c\})
\]

### 2.7 Operators

Each operator of figure 19 is formally defined by one of the following functions.

**Create class** Creation of a new *concrete* class without super classes and without features: a new class name is added.

\[
c c : \mathfrak{M} \times \mathcal{N} \rightarrow \mathfrak{M}
\]
\[
(m, n) \mapsto \begin{cases} n \in m.((\mathcal{C} \cup \mathcal{D}) : m) \\ n \notin m.((\mathcal{C} \cup \mathcal{D}) : m,[\mathcal{C} = \mathcal{C} \cup \{n\}]) \end{cases}
\]

**Create data type** Creation of a new data type: a new data type name is added.

\[
c dt : \mathfrak{M} \times \mathcal{N} \rightarrow \mathfrak{M}
\]
\[
(m, n) \mapsto \begin{cases} n \in m.((\mathcal{C} \cup \mathcal{D}) : m) \\ n \notin m.((\mathcal{C} \cup \mathcal{D}) : m,[\mathcal{D} = \mathcal{D} \cup \{n\}]) \end{cases}
\]
**Create attribute** Creation of a new attribute with default multiplicity 0..1: a new attribute name is added and a new structural link is added as well.

\[
ca : \mathcal{M} \times \mathcal{N}^3 \rightarrow \mathcal{M} \\
(m, n_a, n_c, n_d) \mapsto \\
\begin{cases}
    n_c \notin m.C \lor n_d \notin m.D \lor \exists n \in \alpha_I(n_c), (n, n_a) \in m.(A \cup R) : m \\
    n_c \in m.C \land n_d \in m.D \land \forall n \in \alpha_I(n_c), (n, n_a) \notin m.(A \cup R) : m \\
    m.[A = A \cup \{(n_c, n_a)\}] \\
    .[\delta_A = \delta_A \cup \{(n_c, n_a) \mapsto ((0, 1), n_d)\}]
\end{cases}
\]

**Create reference** Creation of a new reference with default multiplicity 0..1: a new reference name is added and a new structural link is added as well.

\[
cre : \mathcal{M} \times \mathcal{N}^3 \rightarrow \mathcal{M} \\
(m, n_r, n_c, n'_c) \mapsto \\
\begin{cases}
    n_c \notin m.C \lor n'_c \notin m.C \lor \exists n \in \alpha_I(n_c), (n, n_r) \in m.(A \cup R) : m \\
    n_c \in m.C \land n'_c \in m.C \land \forall n \in \alpha_I(n_c), (n, n_r) \notin m.(A \cup R) : m \\
    m.[R = R \cup \{(n_c, n_r)\}] \\
    .[\delta_R = \delta_R \cup \{(n_c, n_r) \mapsto ((0, 1), n'_c)\}]
\end{cases}
\]

**Delete class** Deletion of an existing class which is not a super class and which is not targeted by any reference: the class, its features, and the corresponding structural links are removed.

\[
dc : \mathcal{M} \times \mathcal{N} \rightarrow \mathcal{M} \\
(m, n) \mapsto \\
\begin{cases}
    \exists n' \in C, \delta_I(n') = n \lor \exists (r, m) \in R \times M, \delta_R(r) = (m, n) : m \\
    \forall n' \in C, \delta_I(n') \neq n \land \forall (r, m) \in R \times M, \delta_R(r) \neq (m, n) : m \\
    m.[C = C \setminus \{n\}] \\
    .[C_A = C_A \setminus \{n\}] \\
    .[R = R \setminus \{(n, n_r) \mid n_r \in N\}] \\
    .[R_c = R_c \setminus \{(n, n_r) \mid n_r \in N\}] \\
    .[A = A \setminus \{(n, n_a) \mid n_a \in N\}] \\
    .[\delta_I = \delta_I \setminus \{n \mapsto n' \mid n' \in C\}] \\
    .[\delta_A = \delta_A \setminus \{(n, n_a) \mapsto a \mid (n_a, a) \in N \times (M \times D)\}] \\
    .[\delta_R = \delta_R \setminus \{(n, n_r) \mapsto r \mid (n_r, r) \in N \times (M \times C)\}] \\
    .[\delta_{app} = \delta_{app} \setminus \{x \mapsto (n, n_r) \mid (n_r, x) \in N \times R\}] \\
    .[\delta_{app} = \delta_{app} \setminus \{x \mapsto (n, n_r) \mid (n_r, x) \in N \times R\}]
\end{cases}
\]

**Delete data type** Deletion of an existing data type which is not targeted by any attribute: the data type is simply removed.

\[
ddt : \mathcal{M} \times \mathcal{N} \rightarrow \mathcal{M} \\
(m, n) \mapsto \\
\begin{cases}
    \exists (a, m) \in A \times M, \delta_A(a) = (m, n) : m \\
    \forall (a, m) \in A \times M, \delta_A(a) \neq (m, n) : m.[D = D \setminus \{n\}] \\
\end{cases}
\]
Delete attribute  Deletion of an existing attribute: the attribute and its corresponding structural link are removed.

\[\text{da} : \mathcal{M} \times \mathcal{N}^2 \to \mathcal{M}\]
\[(m, n_a, n_c) \mapsto \{ m.\{A = A \setminus \{(n_c, n_a)\} \}, \delta_A = \delta_A \setminus \{(n_c, n_a) \mapsto a \mid a \in (\mathcal{M} \times \mathcal{D})\} \}\]

Delete reference  Deletion of an existing reference which is not targeted by an opposite reference: the reference and its corresponding structural link are removed.

\[\text{dr} : \mathcal{M} \times \mathcal{N}^2 \to \mathcal{M}\]
\[(m, n_r, n_c) \mapsto \left\{ \begin{array}{l}
\exists r' \in \mathcal{R}, \delta_{\text{opp}}(r') = (n_c, n_r) : m \\
\forall r' \in \mathcal{R}, \delta_{\text{opp}}(r') \neq (n_c, n_r) : m[\mathcal{R} = \mathcal{R} \setminus \{(n_c, n_r)\}] \\
.\left[\mathcal{R}_c = \mathcal{R}_c \setminus \{(n_c, n_r)\}\right] \\
.\left[\delta_R = \delta_R \setminus \{(n_c, n_r) \mapsto r \mid r \in (\mathcal{M} \times \mathcal{C})\}\right]
\end{array} \right.\]

Set classifier name  Setting of a new classifier name: provided the new name does not imply name clashes in the metamodel, the new name is substituted to the old one wherever it appears.

\[\text{sn}_{c} : \mathcal{M} \times \mathcal{N}^2 \to \mathcal{M}\]
\[(m, n_c, n_{c'}) \mapsto \{ n_c \notin m.(C \cup D) \lor n_{c'} \in m.(C \cup D) : m \\
 n_c \in m.(C \cup D) \land n_{c'} \notin m.(C \cup D) : m[n_{c'}/n_c] \}
\]

Set feature name  Setting of a new feature name: provided the new name does not imply name clashes along inheritance links, the new name is substituted to the old one wherever it appears.

\[\text{sn}_{f} : \mathcal{M} \times \mathcal{N}^3 \to \mathcal{M}\]
\[(m, n_c, n_f, n_{f'}) \mapsto \left\{ \begin{array}{l}
(n_c, n_f) \notin m.(\mathcal{R} \cup A) \lor \exists n \in \alpha(f)(n_c), (n, n_{f'}) \in m.\{(\mathcal{R} \cup A)\} : m \\
(n_c, n_f) \in m.\{(\mathcal{R} \cup A)\} \land \forall n \in \alpha(f)(n_c), (n, n_{f'}) \notin m.\{(\mathcal{R} \cup A)\} : m[n_{f'}/(n_c, n_f)]
\end{array} \right.\]

Set lower bound  Setting of a new lower bound for a given feature name: provided the new lower bound is not greater than the corresponding upper bound, the mapping associating the feature to its multiplicity is updated.

\[\text{slb} : \mathcal{M} \times \mathcal{N}^2 \times \mathbb{N} \to \mathcal{M}\]
\[(m, n_c, n_f, i) \mapsto \left\{ \begin{array}{l}
(n_c, n_f) \notin m.(A \cup R) : m \\
(n_c, n_f) \in m.\{(A \cup R)\} : m \\
\text{let } \delta_{\mathcal{F}} = m.(\delta_A \cup \delta_R) \text{ and } ((x, y), n) = \delta_{\mathcal{F}}(n_c, n_f) \text{ in:}
\quad y < i : m \\
\quad y \geq i : m[(n_c, n_f) \mapsto ((i, y), n) / (n_c, n_f) \mapsto ((x, y), n)]
\end{array} \right.\]
**Set upper bound** Setting of a new upper bound for a given feature name: provided the new upper bound is not lower than the corresponding lower bound, the mapping associating the feature to its multiplicity is updated.

\[\text{sub} : \mathcal{M} \times \mathcal{N}^2 \times \mathbb{N}^+ \cup \{\infty\} \rightarrow \mathcal{M}\]

\[
\begin{align*}
(m, n_c, n_f, i) & \mapsto \\
& \begin{cases}
(n_c, n_f) \notin m.\mathcal{A} \cup \mathcal{R} : m \\
(n_c, n_f) \in m.\mathcal{A} \cup \mathcal{R} : \\
& \text{let } \delta_F = m.\delta_A \cup \delta_R \text{ and } ((x, y), n) = \delta_F(n_c, n_f) \text{ in:} \\
& \begin{cases}
& x > i : m \\
& x \leq i : m[(n_c, n_f) \mapsto ((x, i), n) / (n_c, n_f) \mapsto ((x, y), n)]
\end{cases}
\end{cases}
\end{align*}
\]

**Set abstract** Setting a class abstract or concrete: according to a boolean parameter, the class is added to, or removed from the set of abstract classes.

\[\text{sa} : \mathcal{M} \times \mathcal{N} \times \mathbb{B} \rightarrow \mathcal{M}\]

\[
\begin{align*}
(m, n_c, b) & \mapsto \\
& \begin{cases}
& n_c \notin m.\mathcal{C} : m \\
& n_c \in \mathcal{C} \land b : m.\mathcal{C} = \mathcal{C} \cup \{n_c\} \\
& n_c \in \mathcal{C} \land \neg b : m.\mathcal{C} = \mathcal{C} \setminus \{n_c\}
\end{cases}
\end{align*}
\]

**Set container** Setting a reference composite or not: provided the reference is not targeted by an opposite reference, according to a boolean parameter, the reference is added to, or removed from the set of composite references.

\[\text{sc} : \mathcal{M} \times \mathcal{N}^2 \times \mathbb{B} \rightarrow \mathcal{M}\]

\[
\begin{align*}
(m, n_c, n_r, b) & \mapsto \\
& \begin{cases}
& (n_c, n_r) \notin m.\mathcal{R} : m \\
& (n_c, n_r) \in m.\mathcal{R} \land b \land \exists r \in m.\mathcal{R}_C \land \delta_{\text{opp}}(r) = (n_c, n_r) : m \\
& (n_c, n_r) \in m.\mathcal{R} \land b \land \forall r \in m.\mathcal{R}_C \land \delta_{\text{opp}}(r) \neq (n_c, n_r) : m.\mathcal{R} = \mathcal{R}_C \cup \{(n_c, n_r)\} \\
& (n_c, n_r) \in m.\mathcal{R} \land \neg b : m.\mathcal{R} = \mathcal{R}_C \setminus \{(n_c, n_r)\}
\end{cases}
\end{align*}
\]

**Move feature** Moving a feature from a class to another one (the target class): provided the feature does not imply name clashes along inheritance links of the target class, the new structural link (between the target class and the feature) is substituted to the old one wherever it appears.

\[\text{mft} : \mathcal{M} \times \mathcal{N}^3 \rightarrow \mathcal{M}\]

\[
\begin{align*}
(m, n_c, n_f, n'_c) & \mapsto \\
& \begin{cases}
& (n_c, n_f) \notin m.(\mathcal{R} \cup \mathcal{A}) \lor n'_c \notin m.\mathcal{C} \lor \exists n \in \alpha'_f(n'_c), (n, n_f) \notin m.(\mathcal{A} \cup \mathcal{R}) : m \\
& (n_c, n_f) \in m.(\mathcal{R} \cup \mathcal{A}) \land n'_c \in m.\mathcal{C} \land \forall n \in \alpha'_f(n'_c), (n, n_f) \notin m.(\mathcal{A} \cup \mathcal{R}) : m[(n'_c, n_f)/(n_c, n_f)]
\end{cases}
\end{align*}
\]

**Move reference target** Moving the target of a reference from a class to another one (the new target class): the new structural link (between the reference and
the new target class) is substituted to the old one wherever it appears; if an
opposite reference exists, then this opposite reference is moved to the new target
class, provided its name does not imply name clashes along inheritance links of
the new target class.

\[
m\texttt{rtt} : \mathcal{M} \times \mathcal{N}^3 \rightarrow \mathcal{M}
\]
\[
(m, n_c, n_r, n'_c) \mapsto
\begin{cases}
(n_c, n_r) \notin m.\mathcal{R} \lor n'_c \notin m.\mathcal{C} : m \\
(n_c, n_r) \in m.\mathcal{R} \land n'_c \in m.\mathcal{C} : m
\end{cases}
\]
let \((m, n) = m.\delta_\mathcal{R}(n_c, n_r)\) in:
\[
\begin{cases}
\exists n_{ro} \in \mathcal{N} , \delta_{opp}(n, n_{ro}) = (n_c, n_r) : m \\
\forall n' \in \alpha'_f(n'_c) , (n', n_{ro}) \notin m.(\mathcal{R} \cup \mathcal{A}) : m \\
m[(n'_c, n_{ro}) \mapsto (n, n_{ro})] \\
\forall n_{ro} \in \mathcal{N} , ((n, n_{ro}) \mapsto (n_c, n_r)) \notin \delta_{opp} : m[(n_c, n_r) \mapsto (m, n'_c) / (n_c, n_r) \mapsto (m, n)]
\end{cases}
\]

**Move attribute type** Moving the type of an attribute from a data type to
another one (the new data type): the new structural link (between the attribute
and the new data type) is substituted to the old one wherever it appears.

\[
m\texttt{att} : \mathcal{M} \times \mathcal{N}^3 \rightarrow \mathcal{M}
\]
\[
(m, n_c, n_a, n_d) \mapsto
\begin{cases}
(n_c, n_a) \notin m.\mathcal{A} \lor n_d \notin m.\mathcal{D} : m \\
(n_c, n_a) \in m.\mathcal{A} \land n_d \in m.\mathcal{D} : m
\end{cases}
\]
let \((m, n) = m.\delta_\mathcal{A}(n_c, n_a)\) in:
\[
m[(n_c, n_a) \mapsto (m, n_d) / (n_c, n_a) \mapsto (m, n)]
\]

**Add super class** Adding a new super class to a class: provided the new super
class is not also a sub-class, the new inheritance link is added to the set of super
classes.

\[
\text{asc} : \mathcal{M} \times \mathcal{N}^2 \rightarrow \mathcal{M}
\]
\[
(m, n_c, n'_c) \mapsto
\begin{cases}
n_c \notin m.\mathcal{C} \lor n'_c \notin m.\mathcal{C} \lor n_c \notin \alpha'_f(n'_c) : m \\
n_c \in m.\mathcal{C} \land n'_c \in m.\mathcal{C} \land n_c \notin \alpha'_f(n'_c) : m
\end{cases}
\]
\[
m.\delta_f = \delta_f \cup \{n_c \mapsto n'_c\}
\]

**Remove super class** Removing an existing super class from a class: the corre-
spending inheritance link is removed from the set of super classes.

\[
r\texttt{sc} : \mathcal{M} \times \mathcal{N}^2 \rightarrow \mathcal{M}
\]
\[
(m, n_c, n'_c) \mapsto m.\delta_f = \delta_f \backslash \{n_c \mapsto n'_c\}
\]
Move super class Replacement of an existing link between a class and a super class by a link between the same class and a new super class: provided the new super class is not also a sub-class, the new inheritance link is substituted to the old one wherever it appears.

\[
mset : \mathcal{M} \times \mathcal{N}^3 \to \mathcal{M}
\]

\[(m, n_c, n'_c, n''_c) \mapsto \begin{cases} 
  n''_c \notin m \cdot C \lor n_c \in \alpha_f(n''_c) : m \n  n'_c \in m \cdot C \land n_c \notin \alpha_f(n'_c) : m 
  \end{cases}
\]

Move opposite Setting of a new opposite reference to a reference, whenever it already has an opposite reference or not: provided the new opposite reference is not itself associated to another opposite reference, the new opposite references are added and if necessary, and the old opposite references are removed.

\[
mot : \mathcal{M} \times \mathcal{N}^4 \to \mathcal{M}
\]

\[(m, n_c, n_r, n'_c, n'_r) \mapsto \begin{cases} 
  \exists r \in m \cdot R, \delta_{opp}(r) = (n'_c, n'_r) : m 
  \forall r \in m \cdot R. \delta_{opp}(r) \neq (n_c, n_r) : m 
  \forall r \in m \cdot R. \delta_{opp}(r) = (n_c, n_r) : m 
  \forall r \in m \cdot R. \delta_{opp}(r) = (n_c, n_r) : m 
  \end{cases}
\]

Remove opposite Removing an opposite reference: the old opposite references are removed.

\[
ro : \mathcal{M} \times \mathcal{N}^2 \to \mathcal{M}
\]

\[(m, n_c, n_r) \mapsto \begin{cases} 
  \forall r \in m \cdot R. \delta_{opp}(r) \neq (n_c, n_r) : m 
  \exists r \in m \cdot R. \delta_{opp}(r) = (n_c, n_r) : m 
  \end{cases}
\]

2.8 Specifications

We note \( \mathcal{L}(spec) \) the sets of words yielded from \( spec \) in figure 12. Basically, \( \mathcal{L}(spec) \) contains a specification made of one metamodel \( mm \) plus an ordered non-empty set of operators applied to \( mm \).

We note \([op]_{param} \) the set of specific parameters of the operator \( op \), in accordance to figure 19. For instance, we have:

\[
\{createReference(n, n_{class}, n_{targetedClass})\}_{param} \triangleq \{ n, n_{class}, n_{targetedClass} \} \in \mathcal{N}^3
\]
\[
\llbracket \text{spec} \rrbracket : \mathcal{L}(\text{spec}) \rightarrow \mathcal{M}
\]

\[
s \mapsto \begin{cases} 
    \text{s matches with } \{ \text{mm} \} \text{ op} \ 
    : \llbracket \text{op} \rrbracket_{\text{mm}} \left( \llbracket \text{mm} \rrbracket_{\text{mm}}, \llbracket \text{op} \rrbracket_{\text{param}} \right) \\
    \text{s matches with } (s' \text{ op}) \text{ where } s' = (\{ \text{mm} \} \ (\text{op}')^*) 
    : \llbracket \text{op} \rrbracket_{\text{spec}} \left( \llbracket s' \rrbracket_{\text{spec}}, \llbracket \text{op} \rrbracket_{\text{param}} \right)
\end{cases}
\]

Fig. 20. Valuation of \( \mu \text{Dif} \) specifications

We introduce in figure 20 a valuation function noted \( \llbracket \cdot \rrbracket_{\text{spec}} \). It applies to \( \mathcal{L}(\text{spec}) \). It maps a specification to an output metamodel. It is obtained by a recursive application of the functional denotations corresponding to each operator in accordance to figure 19.

3 \( \mu \text{Ecore} \) models

In order to state the principles of model co-evolution with \( \mu \text{Dif} \), we first need to formally define what a model conforming to a \( \mu \text{Ecore} \) metamodel is. For that purpose, we first introduce a syntactical extension of \( \mu \text{Ecore} \). It allows the specification of instances. Then we define a denotational semantics for these instances. This semantics extends the semantic domain of \( \mu \text{Ecore} \) so that a metamodel and a conforming model can be gathered within a same logical framework.

3.1 Syntax extension

\[
\begin{align*}
    n, n_{\text{inst}}, n_{\text{class}}, n_{\text{attrib}}, n_{\text{ref}} & \in \mathcal{N} \quad \text{(name)} \\
    s & \in \mathcal{S} \quad \text{(scalar)} \\
    \text{mod} & ::= \text{from } \text{mm} : \ (i^+) \quad \text{(model)} \\
    i & ::= \ (n_{\text{inst}} : n_{\text{class}} \ (v \mid l)^*) \quad \text{(instance)} \\
    v & ::= \ [n_{\text{attrib}} : s] \quad \text{(value)} \\
    l & ::= \ [n_{\text{ref}} : n_{\text{inst}}] \quad \text{(link)}
\end{align*}
\]

Fig. 21. Textual syntax of \( \mu \text{Ecore} \) models

Figure 21 presents the textual syntax of \( \mu \text{Ecore} \) models. It extends the syntax of \( \mu \text{Ecore} \) metamodels introduced in figure 3. A model is defined by a given metamodel \( \text{mm} \) followed by a non-empty and non-ordered set of instances. An instance is named and it is related to a metaclass whose name is supposed to appear in the metamodel \( \text{mm} \). It is also composed of a sequence of values and links. A value relates an attribute to a scalar value. A link binds a reference to another instance via its name. Several values or links can have the same name.
within an instance if these values or links refer to features corresponding
to collections (i.e. whose multiplicity’s upper bound is greater than 1).

As an illustration, figure 22 shows a model conforming to a \textmu Ecore metamodel
together with its equivalent textual specification. The \textmu Ecore metamodel of this
figure is taken from figure 4.

```
from { Int } \_d { Bool } \_d
  { a A [i, (0, 1), Int] \_a [b, (0, \infty), Bool] \_a [y, (0, 1), Y ↷ a] \_c
    { X / A [j, (1, 1), Int] \_a }
  { Y / A [c a, (1, 2), A ↷ y] \_r [x, (0, \infty), X] \_r }

{ iy1 : Y [i : 3] [b : true] [b : false] [a : iy2] [x : ix] }
{ iy2 : Y [a : ix] [y : iy1] }
{ ix : X [i : 4] [j : 4] [y : iy2] }
```

\textbf{Fig. 22.} Textual and graphical form of a \textmu Ecore model

### 3.2 Semantics of a \textmu Ecore model

The formal semantics of \textmu Ecore models is defined by a mapping between the
models constructs we introduced in figure 21 and a semantic domain including
sets and partial functions. This domain is intended to be an extension of the
semantics of \textmu Ecore metamodels.

### 3.3 Semantic domain

Figure 23 defines the name spaces of the semantic domain. They are compatible
with name spaces \(N, C\) and \(D\) defined for metamodels (see figure 5).

Figure 24 defines the partial functions of the semantic domain. They allow
to gather instances, values, links and metaclasses according to instantiation and
structural links. These functions complete the set of partial functions defined
\[ S \] : scalar values
\[ \mathcal{I} \subseteq \mathcal{N} \setminus (\mathcal{C} \cup \mathcal{D}) \] : instances

**Fig. 23.** \( \mu \text{Ecore} \) models semantics: name spaces

for metamodels (see figure 7) and they are compatible with name spaces \( \mathcal{C}, \mathcal{A} \) and \( \mathcal{R} \) also defined for metamodels (see figure 5). Note the codomains of \( \delta_\nu \) and \( \delta_\ell \) are power sets. Thus, a set is associated to a collection specified by a given multiplicity. As a consequence, duplicated values or duplicated references are not taken into account.

\[
\begin{align*}
\delta_{\text{inst}} &: \mathcal{I} \rightarrow \mathcal{C} \setminus \mathcal{A} \quad \text{instanciation} \\
\delta_\nu &: \mathcal{I} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S}) \quad \text{values} \\
\delta_\ell &: \mathcal{I} \times \mathcal{R} \rightarrow \mathcal{P}(\mathcal{I}) \quad \text{links}
\end{align*}
\]

**Fig. 24.** \( \mu \text{Ecore} \) model semantics: partial functions

Figure 25 defines the notion of model. It corresponds to a triplet composed of name spaces \( n \in \mathcal{N}_i \), partial functions \( f \in \mathcal{F}_i \) and one metamodel \( m \in \mathcal{M} \) (defined in figure 8).

\[
\begin{align*}
\mathcal{I} &\triangleq \mathcal{N}_i \times \mathcal{F}_i \times \mathcal{M} \quad \text{models} \\
\mathcal{N}_i &\triangleq \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) \quad \text{models’ name spaces} \\
\mathcal{F}_i &\triangleq \mathcal{P}(\delta_{\text{inst}}) \times \mathcal{P}(\delta_\nu) \times \mathcal{P}(\delta_\ell) \quad \text{models’ partial functions}
\end{align*}
\]

**Fig. 25.** \( \mu \text{Ecore} \) model semantics

Name spaces are given by a sequence of 2 sets. Partial functions are given by a sequence of 3 partial functions whose definition domains are the name spaces of the model. In concrete terms a model \( i \in \mathcal{I} \) corresponds to the following triplet:

\[ i \triangleq ( (\mathcal{S}, \mathcal{I}) , (\delta_{\text{inst}}, \delta_\nu, \delta_\ell) , m ) \]

Thereafter, we note \( i.x \) the \( x \) component of \( i \) (e.g. \( i.m \) or \( i.\delta_{\text{inst}} \)).

### 3.4 Valuation function

**Notation** We note \( mm(x) \) the metamodel part of a given model specification \( x = \text{from meta} : (i^+) \). By definition in this case, \( mm(x) \triangleq \text{meta} \).
We also complete the notations we introduced on a given parsed metamodel $mm$. We consider now a given parsed model $mod$ and we use the following notation:

$$\text{pattern} \sqsubseteq \text{mod}$$

This notation stands for a proposition stating that an instance matching with the given pattern can be found in $mod$. This pattern corresponds to what can be derived from $i$ according to the syntax specified by figure 21. More formally, if we note $L(i)$ the sets of words yielded from $i$ in figure 21, then by definition:

$$\text{pattern} \in L(i).$$

**Valuation** We note $L(mod)$ the set of words yielded from $mod$ in figure 21. Figure 26 shows the definition of the valuation function. Its maps $\mu\text{Ecore}$ models to the semantic domain $I$ defined by figure 25.

$$\begin{align*}
\llbracket \cdot \rrbracket_{mod} : L(mod) &\to I \\
\multimap x &\mapsto [x]_{mod} = (\langle S, I \rangle, (\delta_{\text{inst}}, \delta_v, \delta_l), \text{m}) \text{ where:} \\
\text{m} &\equiv \llbracket \text{mm}(x) \rrbracket_{mm} \\
S &\equiv \{ s \in S \mid \exists (n, n', n'') \in N^3 \langle n : n' \mid \text{true} \rangle \sqsubseteq x \} \\
I &\equiv \{ n \in N \mid \exists n' \in N, \langle n : n' \rangle \sqsubseteq x \} \\
\delta_{\text{inst}} &\equiv \{ (n, n') \in I \times C \mid \langle n : n' \rangle \sqsubseteq x \} \\
\delta_v &\equiv \{ ((n, (n', n'')), X) \in (I \times A) \times P(S) \mid \exists s \in X, \langle n : n' \mid \text{true} \rangle \sqsubseteq x \} \\
\delta_l &\equiv \{ ((n, (n', n'')), X) \in (I \times A) \times P(I) \mid \exists n_r \in X, \langle n : n' \mid n_r \rangle \sqsubseteq x \}
\end{align*}$$

**Fig. 26. $\mu\text{Ecore}$ model semantics: valuation**

$$\begin{align*}
[x]_{mod} = i = (n, f, \text{m}) &\equiv (\langle S, I \rangle, (\delta_{\text{inst}}, \delta_v, \delta_l), \text{m}) \text{ where:} \\
\text{m} &\equiv \llbracket \text{mm}(x) \rrbracket_{mm} \text{ (see figure 10 for details)} \\
S &\equiv \{ 3, 4, \text{true}, \text{false} \} \\
I &\equiv \{ iy_1, iy_2, ix \} \\
\delta_{\text{inst}} &\equiv \{ iy_1 \mapsto Y, iy_2 \mapsto Y, ix \mapsto X \} \\
\delta_v &\equiv \{ (iy_1, (A, i)) \mapsto \{ 3 \}, (iy_1, (A, b)) \mapsto \{ \text{true}, \text{false} \}, \\
&\quad (ix, (A, i)) \mapsto \{ 4 \}, (ix, (X, j)) \mapsto \{ 4 \} \} \\
\delta_l &\equiv \{ (iy_1, (Y, a)) \mapsto \{ iy_2 \}, (iy_1, (Y, x)) \mapsto \{ ix \}, \\
&\quad (iy_2, (Y, a)) \mapsto \{ ix \}, (iy_2, (A, y)) \mapsto \{ iy_1 \}, (ix, (A, y)) \mapsto \{ iy_2 \} \}
\end{align*}$$

**Fig. 27. Semantics of a $\mu\text{Ecore}$ model**
3.5 Example

Let $x$ be the model of figure 22. Its denotation is given by $\llbracket x \rrbracket_{\text{mod}} = i = (n, f, m)$.

Figure 27 shows $i$ in details, and figure 28 shows a graph-based representation of it. It completes the graph-based representation of the metamodel part (see figure 11).

The new kinds of vertices represent instance names and scalar values. The new kinds of edges represent the partial functions of models, namely $\delta_{\text{inst}}$, $\delta_{v}$ and $\delta_{l}$. The partial function $\delta_{\text{inst}}$ is represented by edges between instances and concrete classes. The partial function $\delta_{v}$ is represented by edges between instances and scalar values. They are labeled by the names of the corresponding attributes. The partial function $\delta_{l}$ is represented by edges linking instances to other instances. They are labeled by the name of the corresponding reference.

3.6 Conformity

As depicted by the example of figure 28, the links between the model and its metamodel are denoted by instantiation edges $\delta_{\text{inst}}$. These structural links are not intended to define a valid model in regard to its metamodel. A conformance property remains to be stated.

Since scalar values are not related to data types in our approach, there are no specific constraints on them. Finally, the following criteria define a valid model in regard to its metamodel:
attribute name for each instance $i$, the name of each outgoing link targeting a scalar value corresponds to the name of an attribute of either the class of $i$ or of one of its ancestor in regard to inheritance

**reference name and type** for each instance $i$, the name of each outgoing link targeting another instance $i'$ corresponds to the name of a reference $r$ of either the class of $i$ or of one of its ancestor in regard to inheritance, and $r$ targets either the class of $i'$ or of one of its ancestor in regard to inheritance

**multiplicity** for each instance $i$, the number of outgoing links having the same name $n$ belongs the interval defined by the multiplicity of the corresponding feature $n$ (attribute or reference) of either the class of $i$ or of one of its ancestor in regard to inheritance

**opposite link** for each reference link between instances $i$ and $i'$, if the corresponding reference has an opposite named $r$, then there is another reference link corresponding to $r$ between instances $i'$ and $i$

This conformance property of a given model $i$ is noted $V(i)$ and is formally defined by figure 29.

$$\forall i \in \mathcal{I}, V(i) \triangleq$$

$$\forall i \in \mathcal{I}, \forall (n_c, n_a) \in \mathcal{N}^2,$$

$$i.\delta_l(i, (n_c, n_a)) \neq \emptyset \implies n_c \in \alpha'_l(i.\delta_{inst}(i)) \land \exists (m, d) \in \mathcal{M} \times \mathcal{D}, i.m.\delta_A(n_c, n_a) = (m, d)$$

$$\land$$

$$\forall i \in \mathcal{I}, \forall (n_c, n_r) \in \mathcal{N}^2,$$

$$i.\delta_l(i, (n_c, n_r)) \neq \emptyset \implies n_c \in \alpha'_l(i.\delta_{inst}(i)) \land \exists (m, c') \in \mathcal{M} \times \mathcal{C}, i.m.\delta_R(n_c, n_r) = (m, c')$$

$$\land$$

$$\forall i' \in i.\delta_l(i, (n_c, n_r)), c' \in \alpha'_l(i.\delta_{inst}(i'))$$

$$\land$$

$$\forall i \in \mathcal{I}, \forall (n_c, n) \in \mathcal{N}^2,$$

let $\text{card} = |i.\delta_l(i, (n_c, n))| \text{ in:}$

let $(\text{min}, \text{max}) = i.m.(\delta_R \cup \delta_A)(n_c, n).m$ in:

$\text{card} \leq \text{max} \land \text{card} \geq \text{min}$

$$\land$$

$$\forall (i, i', n_c, n_r) \in \mathcal{I}^2 \times \mathcal{N}^2, i' \in i.\delta_l(i, (n_c, n_r)),$$

$$\exists (n'_c, n'_r) \in \mathcal{N}^2, i.m.\delta_{opp}(n_c, n_r) = (n'_c, n'_r) \implies i \in i.\delta_l(i', (n'_c, n'_r))$$

**Fig. 29.** Conformance property of $\mu$Ecore models

4 Model co-evolution with $\mu$Dif

So far, $\mu$Dif is defined as metamodel refactoring language whose scope encompass the basic metamodel constructs of $\mu$Ecore. Now, we aim at extending this
scope to model migration. For that purpose, each \( \mu \text{Diff} \) operator (see table 19) is associated to a set of dedicated functions that are intended to perform not only model migration, but also model recontextualization and diagnostics. We call recontextualization the reversed migration from the refactored metamodel to the initial metamodel. The diagnostic of a model transformation enables the analysis of elements that have been discarded or added during the transformation.

### 4.1 Syntax extension

Figure 30 extends the syntax of \( \mu \text{Ecore} \), which itself has been extended to models (see figure 21). A migration is a context (defined by a model \( \text{mod} \)) followed by a non-empty ordered sequence of operators.

\[
mig ::= \{ \text{mod} \} (\text{op})^+ \text{(migration)}
\]

Fig. 30. Textual syntax of \( \mu \text{Diff} \) including model migration

These operators have already been defined for metamodel refactoring in figure 12. They are divided into three CRUD categories, namely create (cr), update (u) and delete (d).

Finally, they can be applied to either metamodels (see figure 12) or models (see figure 30). When they are applied to metamodels, they are intended to perform metamodel refactoring only. When they are applied to models, they are intended to perform both metamodel refactoring and model migration.

### 4.2 Notations

We introduce some specific notations in order to state easily set-based operations on a model \( i \in \mathcal{I} \) corresponding to the following triplet:

\[
i \triangleq ( (S, I) \ , \ (\delta_{\text{inst}}, \delta_{v}, \delta_{l}) \ , \ m )
\]

**Model set operations** Let \( \star \) be a usual set operation like for instance \( \cup \), \( \cap \) or \( \setminus \). We note \( i \star i' \) the model whose components are the result of \( \star \) applied to the matching model components of \( i \) and \( i' \), provided \( i \) and \( i' \) have the same metamodel. For instance, we have:

\[
( (S, I) \ , \ (\delta_{\text{inst}}, \delta_{v}, \delta_{l}) \ , \ m ) \cup ( (S', I') \ , \ (\delta_{\text{inst}'} , \delta_{v}', \delta_{l}') \ , \ m ) \\
\triangleq ( (S \cup S' , I \cup I') \ , \ (\delta_{\text{inst}} \cup \delta_{\text{inst}'} , \delta_{v} \cup \delta_{v}' , \delta_{l} \cup \delta_{l}') \ , \ m )
\]

**Sub-model** We note \( i \subseteq i' \) the proposition stating that each model component of \( i \) is contained by the corresponding component of \( i' \), provided \( i \) and \( i' \) have the same metamodel.
**Substitution of model component** We note $i[ x = y ]$ the model $i$ where $y$ has been substituted to the $x$ component of $m$ (e.g. $i[ m = ... ]$).

**Empty model** We note $i^0$ the model which metamodel corresponds to $i.m$, and whose model components are empty:

$$\forall i \in I, \ i^0 \triangleq i[ S = \emptyset ],[ I = \emptyset ],[ \delta_{\text{inst}} = \emptyset ],[ \delta_v = \emptyset ],[ \delta_l = \emptyset ]$$

Note that the metamodel part actually remains unchanged:

$$\forall i \in I, \ i^0.m = i.m$$

Note also that this empty model is always a valid model in regard to its metamodel.

### 4.3 Principles of reversible model migration

Before giving details about the semantics of $\mu$Dif model migration in regard to each operator, we first present here the underlying principles of this migration, its reversibility and its diagnostic facilities.

To illustrate these principles, we note $op$ the functional denotation of a given $\mu$Dif operator taken from table 19. For instance, $op$ can refer to $cc : M \times N \rightarrow M$, the denotation of $\text{createClass}$. According to table 19, $op$ applies to a metamodel and a sequence of extra parameters, and as a result, it provides a refactored metamodel:

$$\text{let } \mathcal{P} \text{ be the parameter domain of } op, \quad op : M \times \mathcal{P} \rightarrow M \quad (m, p) \mapsto m_{\text{refactored}}$$

**Migration** Applying $op$ to a model implies to identify elements that should be discarded and elements that should be added.

**Discarded elements** Let $op$ be the functional denotation of a given $\mu$Dif operator and $\mathcal{P}$ its parameter domain. We note $\partial \delta^r$ the function mapping a model to the elements that have to be removed according to the semantics of $op$. These elements are gathered within a model conforming to the input metamodel (i.e. the metamodel before its refactoring):

$$\partial \delta^r : I \times \mathcal{P} \rightarrow I \quad (i, p) \mapsto i' \text{ such that } i' \subseteq i$$

This definition implies that discarded elements are actually taken from the model which is intended to be migrated.
**Added elements** Let \( \text{op} \) be the functional denotation of a given \( \mu \text{Diff} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overrightarrow{\text{op}}^a \) the function mapping a model to the elements that have to be added according to the semantics of \( \text{op} \). These elements are gathered within a model conforming to the refactored input metamodel (i.e. the metamodel after its refactoring):

\[
\overrightarrow{\text{op}}^a : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I}
\]

\[
(i, p) \mapsto i' \text{ such that } i'.m = \text{op}(i.m, p) \land i' \setminus \{ i.|m = \text{op}(i.m, p) \} = i'
\]

This definition implies that the model we obtain is associated to the refactored metamodel, and that no added elements are already present in the initial model.

**Transformation** Let \( \text{op} \) be the functional denotation of a given \( \mu \text{Diff} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overrightarrow{\text{op}} \) the function mapping a model to the corresponding migrated model according to the semantics of \( \text{op} \).

\[
\overrightarrow{\text{op}} : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I}
\]

\[
(i, p) \mapsto (i \setminus \overrightarrow{\text{op}}^r(i, p)).|m = \text{op}(i.m, p)) \cup \overrightarrow{\text{op}}^a(i, p)
\]

By definition the migrated model corresponds to the initial model where some elements have been firstly discarded, and then where the metamodel has been refactored, and where finally some new elements have been added.

**Validation** For each operator \( \text{op} \) of table 19, the migration is formally defined by the explicit functions \( \overrightarrow{\text{op}}^r \) and \( \overrightarrow{\text{op}}^a \). This definition must be validated in regard to conformity. In concrete terms, we need to prove under what conditions a valid input model is transformed by \( \overrightarrow{\text{op}} \) into a valid output model.

Let \( \text{op} \) be the functional denotation of a given \( \mu \text{Diff} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overrightarrow{\text{op}} \) the predicate giving the sufficient condition under which a valid input model is transformed by \( \overrightarrow{\text{op}} \) into a valid output model:

\[
\overrightarrow{\text{op}} : \mathcal{I} \times \mathcal{P} \rightarrow \mathbb{B}
\]

\[
\forall (i, p) \in \mathcal{I} \times \mathcal{P}, \ (\forall (i) \land \overrightarrow{\text{op}}(i, p)) \implies \forall (\overrightarrow{\text{op}}(i, p))
\]

If there is no specific condition, then by definition, \( \forall (i, p) \in \mathcal{I} \times \mathcal{P}, \overrightarrow{\text{op}}(i, p) = \text{true} \), and thus, a valid input model is always transformed into a valid model in regard to conformity.

**Recontextualization** We focus now on an initial model (conforming to an initial metamodel) which has been migrated, i.e. which has been transformed into a model conforming to a refactored metamodel. We want to transform it back into a model conforming to the initial metamodel. We call this transformation recontextualization. This concern makes sense if the migrated model has been processed and potentially modified, typically by a rewriting tool we aim at reusing.
During the first migration, some elements have been respectively removed or added, and they should be respectively recovered or deleted, as far as it does not challenges the modifications made on the migrated model.

Recontextualization depends on \( op \), the \( \mu \text{Diff} \) operator that has been used for the migration. It also depends on a migrated model and its initial metamodel. As for migration, reversing \( op \) from a migrated model implies to identify elements that should be discarded and added. As for migration, the validity of this transformation is subject to specific conditions.

**Discarded elements** Let \( op \) be the functional denotation of a given \( \mu \text{Diff} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overrightarrow{op}^r \) the function mapping a migrated model to the elements that have to be removed according to the semantics of \( op \). These elements are gathered within a model conforming to a refactored metamodel (i.e. a metamodel after its refactoring):

\[
\overrightarrow{op}^r : \mathcal{I} \times \mathcal{P} \times \mathcal{I} \to \mathcal{I}
\]

\[
(i, p, i_{ini}) \rightarrow i' \text{ such that } i' \subseteq \left( (i \setminus \overrightarrow{op}(i_{ini}, p)) \cup \overrightarrow{op}^r(i_{ini}, p) \right) \\
\land (i = \overrightarrow{op}(i_{ini}, p)) \implies (i' = \overrightarrow{op}^r(i_{ini}, p))
\]

This definition implies that discarded elements are taken from two specific subsets. The first one corresponds to elements that have been added by a tool, i.e. elements that were not included in the original migrated model \((i' \subseteq (i \setminus \overrightarrow{op}(i_{ini}, p)))\). The second specific subset is a part of the elements that have been added by the migration \((i' \subseteq \overrightarrow{op}^a(i_{ini}, p))\). In the case of a non-modified migrated model \((i = \overrightarrow{op}(i_{ini}, p))\), by definition, the set of discarded elements matches with the set of elements that have been added by the migration \((i' = \overrightarrow{op}^a(i_{ini}, p))\).

**Added elements** Let \( op \) be the functional denotation of a given \( \mu \text{Diff} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overrightarrow{op}^a \) the function mapping a migrated model to the elements that have to be added according to the semantics of \( op \). These elements are gathered within a model conforming to the input metamodel (i.e. the metamodel before its refactoring):

\[
\overrightarrow{op}^a : \mathcal{I} \times \mathcal{P} \times \mathcal{I} \to \mathcal{I}
\]

\[
(i, p, i_{ini}) \rightarrow i' \text{ such that } i'.m = i_{ini}.m \land i' \setminus (i.[m = i_{ini}.m]) = i' \\
\land (i = \overrightarrow{op}(i_{ini}, p)) \implies (i' = \overrightarrow{op}^a(i_{ini}, p))
\]

This definition implies that the model we obtain is associated to the initial metamodel, and that this model has no common element with the refactored model.

Adding elements during the recontextualization only makes sense if the migrated model has been modified after the migration. In the other case \((i = \overrightarrow{op}(i_{ini}, p))\), there is no need to add any specific element. Hence the last condition in this case: the set of added elements matches with the set of elements that have been discarded by the migration \((i' = \overrightarrow{op}^r(i_{ini}, p))\).
Transformation Let \( op \) be the functional denotation of a given \( \text{\muDif} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \delta p \) the function mapping a migrated model to the corresponding initial model according to the semantics of \( op \).

\[
\delta p : \mathcal{I} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathcal{I}
\]

\[
(i, p, i_m) \mapsto ((i \setminus \delta p^r(i, p, i_m)), [m = i_m], m) \cup \delta p^a(i, p, i_m)
\]

By definition the initial model corresponds to the refactored model where:
- some model elements are discarded: they include new elements (typically added by a tool) and the elements that had been added during the migration
- some new elements are added: they include specific new elements and a part of the elements that had been discarded during the migration

Validation For each operator \( op \) of table 19, the recontextualization is formally defined by the explicit functions \( \delta p^a \), \( \delta p^r \), \( \delta p^c \) and \( \delta p^r \). As for migration, this definition must be validated in regard to conformity. In concrete terms, we need to prove under what conditions a valid migrated model is actually transformed by \( \delta p \) back into a valid initial model.

Let \( op \) be the functional denotation of a given \( \text{\muDif} \) operator and \( \mathcal{P} \) its parameter domain. We note \( \overline{C_{op}} \) the predicate giving the sufficient condition under which a valid possibly modified migrated model coming from a given valid migrated model is transformed by \( \delta p \) back into a valid initial model:

\[
\overline{C_{op}} : \mathcal{I} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathbb{B}
\]

\[
\forall (i, p, i_m) \in \mathcal{I} \times \mathcal{P} \times \mathcal{I}, \ (V(i) \land V(i_m) \land \overline{C_{op}}(i, p, i_m)) \implies V(\delta p(i, p, i_m))
\]

As for migration, if there is no specific condition, then by definition, \( \forall (i, p, i_m) \in \mathcal{I} \times \mathcal{P} \times \mathcal{I}, \overline{C_{op}}(i, p, i_m) = \text{true} \), and thus, a valid migrated model is always transformed back into a valid initial model in regard to conformity.

Main property Recontextualization is intended to undo migration. Thus, the composition of migration and recontextualization leads to identity:

**Theorem 1** Let \( op \) be the functional denotation of a given \( \text{\muDif} \) operator and \( \mathcal{P} \) its parameter domain.

\[
\forall i \in \mathcal{I}, \forall p \in \mathcal{P}, \ \delta p(\delta p(i, p), p, i) = i
\]

**Proof** Let \( i \in \mathcal{I} \) be a given input model and \( p \in \mathcal{P} \) be a valid set of parameters in regard to \( op \). Then by definition of \( \delta p \) we have:

\[
\delta p(\delta p(i, p), p, i) = (\delta p(i, p) \setminus \delta p^r(\delta p(i, p), p, i)), [m = i_m] \cup \delta p^a(\delta p(i, p), p, i)
\]

However, by definition of \( \delta p^r \), knowing \( x \) is the first parameter, since we directly have \( x = \delta p(i_m, p) \) (because the \( x \) corresponds here to \( \delta p(i, p) \) and \( i_m \) corresponds to \( i_m \)), then we also have \( \delta p^r(\delta p(i, p), p, i) = \delta p^a(i, p) \). Thus:

\[
\delta p(\delta p(i, p), p, i) = (\delta p(i, p) \setminus \delta p^a(i, p)), [m = i_m] \cup \delta p^a(\delta p(i, p), p, i)
\]
By definition of $\overrightarrow{op}$, knowing $x$ is the first parameter, since we directly have $x = \overrightarrow{op}(i_{\text{init}}; p)$ (because the $x$ corresponds here to $\overrightarrow{op}(i; p)$ and $i_{\text{init}}$ corresponds to $i$), then we also have $\overrightarrow{op}^{\alpha}(\overrightarrow{op}(i; p), p, i) = \overrightarrow{op}^\mu(i; p)$. Thus:

$$\overrightarrow{op}(\overrightarrow{op}(i; p), p, i) = ((\overrightarrow{op}(i; p) \setminus \overrightarrow{op}^{\alpha}(i; p))[m = i.m]) \cup \overrightarrow{op}^\mu(i; p)$$

By definition of $\overrightarrow{op}$, we have now:

$$\overrightarrow{op}(\overrightarrow{op}(i; p), p, i) = \left( (i \setminus \overrightarrow{op}^\mu(i; p), [m = \text{op}(i.m, p)]) \cup \overrightarrow{op}^{\alpha}(i; p) \right)[m = i.m]$$

Now by definition:

$$\overrightarrow{op}^{\alpha}(i; p) \setminus (i, [m = \text{op}(i.m, p)]) = \overrightarrow{op}^\alpha(i; p)$$

And Then:

$$\overrightarrow{op}^{\alpha}(i; p) \setminus (i \setminus \overrightarrow{op}^\mu(i; p), [m = \text{op}(i.m, p)]) = \overrightarrow{op}^\mu(i; p)$$

Thus, adding and deleting $\overrightarrow{op}^{\alpha}(i; p)$ from $(i \setminus \overrightarrow{op}^\mu(i; p), [m = \text{op}(i.m, p)])$ has no effect. Hence:

$$\overrightarrow{op}(\overrightarrow{op}(i; p), p, i) = (i \setminus \overrightarrow{op}^\mu(i; p), [m = i.m]) \cup \overrightarrow{op}^\mu(i; p)$$

The double applying of metamodel substitution has no effect:

$$\overrightarrow{op}(\overrightarrow{op}(i; p), p, i) = (i \setminus \overrightarrow{op}^\mu(i; p)) \cup \overrightarrow{op}^\mu(i; p)$$

By definition of $\overrightarrow{op}^\mu(i; p)$:

$$\overrightarrow{op}^\mu(i; p) \subseteq i$$

Thus, deleting and adding $\overrightarrow{op}^\mu(i; p)$ from $i$ has no effect. Hence:

$$\overrightarrow{op}(\overrightarrow{op}(i; p), p, i) = i$$

□

4.4 Diagnostics

Migration and recontextualization are defined by means of specific sub-models gathering added and removed model elements. These sets also give way to know whether the corresponding transformation is relevant or not. This kind of knowledge is domain-dependent. Indeed, $\mu\text{Diff}$ allows for instance to delete a concept (i.e. a meta-class) from the metamodel, but only the domain expert knows if this concept is useless or forbidden.

This is a significant difference at the model level. If the deleted concept is useless and if a model to be migrated includes some instances of it, then they are simply and safely removed. But if the deleted concept is forbidden and if a model to be migrated includes some instances of it, then the model is probably unsuitable for the targeted tool.

$\mu\text{Diff}$ allows domain expert to diagnosis migrations, i.e. to distinguish safe migrations from others thanks to their associated sets of added and removed model elements.
Migration diagnostic  Let $op$ be a given $\mu$Diff operator, $i$ be a given input model, and $p$ be a valid set of parameters. The set of classes corresponding to deleted instances is defined as follows:

$$\{ c \in i.m.C \mid \exists i \in \overrightarrow{\partial^p}(i, p).I, \ i.\delta_{inst}(i) = c \}$$

The sets of references corresponding to deleted links is defined as follows:

$$\{ r \in i.m.R \mid \exists (i \mapsto I) \in \overrightarrow{\partial^p}(i, p), \ i.\delta_l(i, r) = I \}$$

The sets of attributes corresponding to deleted values is defined as follows:

$$\{ a \in i.m.A \mid \exists (i \mapsto S) \in \overrightarrow{\partial^p}(i, p), \ i.\delta_v(i, a) = S \}$$

These definitions allow the domain expert to spot instances that have been deleted before the tool’s application. The domain expert can use this information to identify unsafe model migrations.

Moreover, if there is a specific condition to have a valid migrated model, then these sets can be used to understand why this condition is not satisfied.

Black-box rewriting tool diagnostic  Recontextualization makes sense if the migrated model is modified, by a rewriting tool for instance. But in this case, we need to ensure the tool’s action is not challenged by this transformation. More precisely, we first need to observe the tool’s action, and then we need to give way to know whether the transformation counteracts the tool’s action or not.

The tool’s action can be defined in regard to added and removed elements (instances, scalar values and links) at the model level. From the outside of the tool, considering it as a black box, updated elements cannot be distinguished from a pair of added and removed elements.

Let $tool$ be a given rewriting tool applying to a migrated model and a set of parameters. Let $P$ its parameter domain. Thereafter, we note $\overrightarrow{\text{tool}^a}$ (resp. $\overrightarrow{\text{tool}^r}$) the function mapping an input migrated model to the elements that are added (resp. removed) by the tool:

$$\overrightarrow{\text{tool}^a} : I \times P \to I \quad (i, p) \mapsto tool(i, p) \setminus i$$

$$\overrightarrow{\text{tool}^r} : I \times P \to I \quad (i, p) \mapsto i \setminus tool(i, p)$$

Contextualization diagnostic  Let $op$ be a given $\mu$Diff operator, $i_{ini}$ be a given initial model, $i$ be a given migrated model, and $p$ be a valid set of parameters. The set of classes corresponding to deleted instances during the contextualization is defined as follows:

$$\{ c \in i.m.C \mid \exists i \in \overrightarrow{\partial^p'}(i, p, i_{ini}).I, \ i.\delta_{inst}(i) = c \}$$
The sets of references corresponding to deleted links is defined as follows:

\[ \{ r \in i.m.R \mid \exists(i \mapsto I) \in \delta p^r(i, p, i_m).\delta_1, \ i.\delta_1(i, r) = I \} \]

The sets of attributes corresponding to deleted values is defined as follows:

\[ \{ a \in i.m.A \mid \exists(i \mapsto S) \in \delta p^a(i, p, i_m).\delta_2, \ i.\delta_2(i, a) = S \} \]

These definitions allow the domain expert to spot instances that have been added by the tool, and that have been later removed by the contextualization. These instances are typically irrelevant within the initial context, but the domain expert can decide whether their deletion counteracts the tool’s action or not.

As for migration, if there is a specific condition to have a valid model after recontextualization, then these sets can be used to understand why this condition is not satisfied.

### 4.5 By-default model migration

We consider the 22 \( \mu \text{Dif} \) operators of table 19. Now we aim at formally defining the migration and the recontextualization associated to each of them. For a given operator \( op \), in accordance with the principles we stated before, we mainly need to define the functions \( \delta^r p, \delta^a p, \delta^s p \) and \( \delta^a p \). We also need to validate these operations in regard to conformity by the definitions of predicates \( C^r_{op} \) and \( C^a_{op} \).

In many cases, the sets of discarded or added elements are empty. If both are empty, it corresponds to a metamodel refactoring operator which has no effects at the model level. Also in many cases, there are no specific conditions to maintain the validity of transformed models in regard to conformity.

Thus, we introduce the following generic default semantics for a given operator \( op \) and its associated parameter domain \( P \):

\[
\delta^r p : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I} \quad \delta^a p : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I} \quad \delta^s p : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I} \\
(i, p) \mapsto i^0 \quad (i, p) \mapsto i^0, [m = op(i.m, p)]
\]

\[
\delta^r p : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I} \quad \delta^a p : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{I} \\
(i, p, i_{ini}) \mapsto i^0 \quad (i, p, i_{ini}) \mapsto i^0, [m = i_{ini}.m]
\]

\[
C^r_{op} : \mathcal{I} \times \mathcal{P} \rightarrow \mathbb{B} \quad C^a_{op} : \mathcal{I} \times \mathcal{P} \times \mathcal{I} \rightarrow \mathbb{B} \\
(i, p) \mapsto true \quad (i, p, i_{ini}) \mapsto true
\]

Note in this default case, we actually have the following required conditions:

\[
\forall(i, p) \in \mathcal{I} \times \mathcal{P}, \ i^0 \subseteq i
\]

\[
\forall(i, p) \in \mathcal{I} \times \mathcal{P}, \ ((i^0, [m = op(i.m, p)]).m) = op(i.m, p) \land (i^0 \setminus i = i^0)
\]

\[
\forall(i, p, i_{ini}) \in \mathcal{I} \times \mathcal{P} \times \mathcal{I}, \ i^0 \subseteq ((i \setminus \delta^r p(i_{ini}, p)) \cup \delta^a p(i_{ini}, p)) \land (i^0 = \delta^a p(i_{ini}, p))
\]

\[
\forall(i, p, i_{ini}) \in \mathcal{I} \times \mathcal{P} \times \mathcal{I}, \ ((i^0, [m = i_{ini}.m]).m) = i_{ini}.m \land (i^0 \setminus (i, [m = i_{ini}.m]) = i^0 \land i^0 = \delta^r p(i_{ini}, p))
\]
4.6 Model migration by operator in detail

We define now the 22 \( \mu \text{Dif} \) operators of table 19. For each of them we only give the specific definitions of \( \overrightarrow{op}^{p}, \overrightarrow{op}^{a}, \overrightarrow{op}^{p}, \overrightarrow{op}^{a}, \overrightarrow{C_{op}} \) and \( C_{op}^{2} \). More precisely, we only give these definitions when they are different from the generic default semantics we stated before.

Create class

The creation of a new concrete class without super classes and without features has no effects on conforming models. Thus, nothing needs to be deleted or added during the migration.

However, the recontextualization implies to delete instances of this class whether they have been added by a rewriting tool. In this case, the migrated model cannot contains links targeting these instances because the corresponding metamodel does not have references targeting the new class.

\[
\overrightarrow{cc}^{r} : \mathcal{I} \times \mathcal{N} \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n, i_{ini}) \mapsto \left\{ \begin{array}{l}
 i^\emptyset & .\mathcal{I} = \{ i \in \mathcal{I} \mid \delta_{\text{inst}}(i) = n \} \\
 .\mathcal{I}_{\text{inst}} = \{ (i, c) \in \mathcal{I} \times \mathcal{m}.\mathcal{C} \mid c = n \land \delta_{\text{inst}}(i) = c \} 
\end{array} \right.
\]

Deleted elements during the recontextualization corresponds to instances that could not appear in any initial model, and that have not been added during the migration. Finally, we can easily check that we actually have:

\[
\forall (n, i, i_{ini}) \in \mathcal{N} \times \mathcal{I}^{2}, \overrightarrow{cc}^{r}(i, n, i_{ini}) \subseteq (i \setminus \overrightarrow{c}(i_{ini}, p))
\]

Thus, if the migrated model is kept unchanged, \textit{i.e.} if \( i = \overrightarrow{c}(i_{ini}, p) \), then \( i \setminus \overrightarrow{c}(i_{ini}, p) = i^\emptyset \). Then in this case we actually have:

\[
\overrightarrow{cc}^{r}(i, n, i_{ini}) = i^\emptyset = \overrightarrow{cc}^{a}(i, n)
\]

Validity

We only give here proof sketches since the comprehensive proofs are more tedious (because of notations) than inherently difficult.

There is no specific conditions to keep the conformance property over the migration and the recontextualization associated to \( cc \). Indeed, we do not add or remove anything during the migration, and the metamodel is kept unchanged except a new class without features and without instances is added. And during the recontextualization, the only deleted instances correspond to the deleted class in the metamodel.

Create data type

As for a new class, the creation of a new data type has no effects on conforming models. Thus, nothing needs to be deleted or added during the migration.

Moreover, the scalar values are related to a data type by means of value links corresponding to attributes. But the creation of a data type has no effects on attributes (the modification of the attribute type is implemented by operation \( \text{moveAttributeTypeTo} \)). Thus, the recontextualization does not require to delete anything, nor to add anything.
Validity Since nothing is changed at the model level when a data type is created or canceled, we obviously don’t need any specific condition to preserve the conformance property.

Create attribute The creation of a new attribute with default multiplicity 0..1 has no effects on conforming models since this new attribute is not mandatory. Thus, nothing needs to be deleted or added during the migration.

However, the recontextualization implies to delete the value links corresponding to this new attribute whether they have been added by a rewriting tool.

After this deletion, some scalar values may be isolated in the model. Therefore, these values are also deleted.

\[ \mathcal{C}^\rightarrow : \mathcal{I} \times \mathcal{N}^3 \times \mathcal{I} \to \mathcal{I} \]

\[ (i, n_a, n_c, n_d, i_m) \mapsto \begin{cases} i^\emptyset, [\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathcal{m.A} \times \mathcal{P}(\mathcal{S}) \mid a = (n_c, n_a) \land \delta_v(i, a) = S]\} \\ \delta_v = \{s \in \mathcal{S} \mid \forall (i, a) \in \mathcal{I} \times \mathcal{m.A}, s \notin \delta_v(i, a)\} \end{cases} \]

Validity A valid model transformed by \( \mathcal{C}^\rightarrow \) remains valid because nothing is added at the model level and the new attribute at the metamodel level is not mandatory.

A valid model transformed by \( \mathcal{C}^\rightarrow \) remains valid because the only deleted model elements are value links corresponding to an attribute which is removed from the metamodel.

Thus, we don’t need any specific condition to preserve the conformance property over the migration and the recontextualization.

Create reference As for attributes, the creation of a new reference with default multiplicity 0..1 has no effects on conforming models since this new reference is not mandatory. Thus, nothing needs to be deleted or added during the migration.

However, the recontextualization implies to delete the reference links corresponding to this new reference whether they have been added by a rewriting tool.

\[ \mathcal{C}^\rightarrow : \mathcal{I} \times \mathcal{N}^3 \times \mathcal{I} \to \mathcal{I} \]

\[ (i, n_r, n_c, n'_c, i_m) \mapsto \begin{cases} i^\emptyset, [\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathcal{m.R} \times \mathcal{P}(\mathcal{I}) \mid r = (n_c, n_r) \land \delta_l(i, r) = S}\} \\ \delta_l = \{s \in \mathcal{S} \mid \forall (i, r) \in \mathcal{I} \times \mathcal{m.R}, s \notin \delta_l(i, r)\} \end{cases} \]

Validity As for new attributes, a valid model transformed by \( \mathcal{C}^\rightarrow \) remains valid because nothing is added at the model level and the new reference at the metamodel level is not mandatory.

In a same way, a valid model transformed by \( \mathcal{C}^\rightarrow \) remains valid because the only deleted model elements are reference links corresponding to a reference which is removed from the metamodel, and which has no opposite reference by definition.

Thus, we don’t need any specific condition to preserve the conformance property over the migration and the recontextualization.
Delete class

The deletion of an existing class which is not a super class and which is not targeted by any reference implies to delete its instances at the model level, as also its attributes and references. This modification is safe since by definition, the deleted instances are not targeted by any reference link. There is no need to add anything specific during the migration.

Once migrated, a model can be processed by a rewriting tool. In this case, no new elements introduced by this tool are likely to be remove during the recontextualization. Indeed, the operation which is supposed to be undone by the recontextualization is a deletion. Thus, we only need to add the instances and the links that had been discarded during the migration, provided these links are still related to existing instances.

$$\overrightarrow{dc^r} : \mathcal{I} \times \mathcal{N} \rightarrow \mathcal{I}$$

$$(i, n) \mapsto$$

$$\begin{cases}
\{i \in \mathcal{I} \mid \delta_{\text{inst}}(i) = n\} \\
\{i \in \mathcal{I} \times \mathcal{C} \mid c = n \land \delta_{\text{inst}}(i) = c\} \\
\{i \in \mathcal{I} \times \mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\
\delta_{\text{inst}}(i) = n \land \delta_{l}(i, r) = S\} \\
\{i \in \mathcal{I} \times \mathcal{A} \times \mathcal{P}(S) \mid \\
\delta_{\text{inst}}(i) = n \land \delta_{v}(i, a) = S\}
\end{cases}$$

$$\overrightarrow{dc^a} : \mathcal{I} \times \mathcal{N} \times \mathcal{I} \rightarrow \mathcal{I}$$

$$(i, n, i_{\text{ini}}) \mapsto$$

$$\begin{cases}
\{i \in i_{\text{ini}} \mathcal{I} \mid i_{\text{ini}} . \delta_{\text{inst}}(i) = n\} \\
\{i \in i_{\text{ini}} \mathcal{I} \times \mathcal{C} \mid c = n \land i_{\text{ini}} . \delta_{\text{inst}}(i) = c\} \\
\{i \in i_{\text{ini}} \mathcal{I} \times i_{\text{ini}} \mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\
i_{\text{ini}} . \delta_{\text{inst}}(i) = n \land \forall j \in S, j \in (\delta_{l} \cap i_{\text{ini}} . \delta_{l})(i, r)\} \\
\{i \in i_{\text{ini}} \mathcal{I} \times i_{\text{ini}} \mathcal{A} \times \mathcal{P}(S) \mid \\
i_{\text{ini}} . \delta_{\text{inst}}(v) = n \land \forall j \in S, j \in (\delta_{v} \cap i_{\text{ini}} . \delta_{v})(i, a)\}
\end{cases}$$

Validity

A valid model transformed by $\overrightarrow{dc}$ remains valid because all the deleted model elements are isolated scalar values and either instances of the deleted class or links going out of them. By definition, the initial model does not include links targeting the deleted instances.

However, a valid model transformed by $\overrightarrow{dc}$ remains valid after the recontextualization only if the discarded links that cannot be recovered (because of a deleted target) are not mandatory in regard to multiplicity. Hence the unique following predicate giving the sufficient condition under which a valid model remains valid over recontextualization:

$$\overrightarrow{C_{dc}} : \mathcal{I} \times \mathcal{N} \times \mathcal{I} \rightarrow \mathbb{B}$$

$$(i, n, i_{\text{ini}}) \mapsto (i_{\text{ini}} \mathcal{I} \setminus i \mathcal{I}) \cap \cup_{i \in i_{\text{ini}}} \mathcal{I} \left(i_{\text{ini}} . \delta_{l}(i)\right) = \emptyset$$

If all instances targeted by the discarded links are kept, then this is a sufficient condition to preserve validity.

Delete data type

Unlike the deletion of an existing class, the deletion of an existing data type which is not targeted by any attribute does not imply any
modification at the model level. Thus, nothing needs to be deleted or added during the migration or the recontextualization.

**Validity** Since nothing is changed at the model level when a data type is deleted or recovered, we obviously don’t need any specific condition to preserve the conformance property.

**Delete attribute** The deletion of an existing attribute implies to remove all the corresponding attribute links at the model level. There is no need to add anything during the migration. During the recontextualization, the deleted links are recovered, as far as the corresponding instances still exist.

\[
\overrightarrow{da^a} : \mathcal{I} \times \mathcal{N}^2 \to \mathcal{I} \\
(i, n_a, n_c) \mapsto \left\{ i^\emptyset, [\delta_v = \{(i, a, S) \in \mathcal{I} \times m.A \times P(S) \mid a = (n_c, n_a) \land \delta_v(i, (n_c, n_a)) = S}\right]\]

\[
\overleftarrow{da^a} : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \to \mathcal{I} \\
(i, n_a, n_c, i_{ini}) \mapsto \left\{ i^\emptyset, [\delta_v = \{(i, a, S) \in i.I \times i_{ini}.m.A \times P(i_{ini}.S) \mid a = (n_c, n_a) \land i_{ini}.\delta_v(i, (n_c, n_a)) = S}\right]\]

**Validity** The migration by \(da\) of a valid model leads to a new valid model because the only deleted model elements are isolated scalar values and value links corresponding to the attribute which is intended to be removed from the metamodel.

However, we need to state a specific condition for the recontextualization. Indeed, some instances of the class owning the deleted attribute may have been added at the model level. After the recontextualization, these instances won’t have values associated to the recovered attribute. This cannot be valid if the multiplicity of this attribute has a lower bound greater than 0, i.e., it is mandatory. We introduce the predicates \(C_1\) and \(C_2\) to address this case:

\[
C_1 : \mathcal{N}^2 \times i \to B \\
(n_a, n_c, i) \mapsto \exists ((x, y), d) \in \mathcal{M} \times \mathcal{D}, (i.m.\delta_A(n_c, n_a) = ((x, y), d) \land x = 0)
\]

\[
C_2 : \mathcal{N} \times i^2 \to B \\
(n_c, i_{ini}) \mapsto \{i \in i.I \mid i.\delta_{inst}(i) = n_c\} \setminus \{i \in i.I \mid i_{ini}.\delta_{inst}(i) = n_c\} = \emptyset
\]

Recontextualization is valid if the multiplicity of the deleted attribute is not mandatory \((C_1)\). Recontextualization is also valid if the set of new instances of the class from which the attribute has been removed is empty \((C_2)\). Hence the following global condition for the conformance properties:

\[
\overleftarrow{\overline{C}_{da}} : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \to B \\
(i, n_a, n_c, i_{ini}) \mapsto (C_1(n_a, n_c, i) \lor C_2(n_c, i, i_{ini}))
\]
Delete reference  The deletion of an existing reference implies to remove all the corresponding reference links at the model level. There is no need to add anything during the migration.

During the recontextualization, the deleted links are recovered, as far as the corresponding source and target still exist.

\[
\overleftarrow{dr}^r : \mathcal{I} \times \mathcal{N}^2 \rightarrow \mathcal{I} \\
(i, n_r, n_c) \mapsto \{ \delta_l = \{ (i, r, S) \in \mathcal{I} \times \mathcal{M} \rightarrow \mathcal{P}(\mathcal{I}) \mid r = (n_c, n_r) \land \delta_l(i, (n_c, n_r)) = S \} \}
\]

\[
\overrightarrow{dr}^a : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_r, n_c, i_{ini}) \mapsto \{ \delta_l = \{ (i, r, S) \in i_{ini} \times \mathcal{M} \rightarrow \mathcal{P}(i_{ini}) \mid r = (n_c, n_r) \land \forall j \in S \rightarrow j \in i_{ini}.\delta_l(i, r) \} \}
\]

Validity  The migration by \(dr\) of a valid model leads to a new valid model because the only deleted model elements are links corresponding to the reference which is intended to be removed from the metamodel.

However, as for attributes and for the same reasons, we need to state a specific condition for the recontextualization. Indeed, some instances of the class owning the deleted reference may have been added at the model level. After the recontextualization, these instances won’t have values associated to the recovered reference. This cannot be valid if the multiplicity of this reference has a lower bound greater than 0, i.e. it is mandatory.

We introduce the specific predicate \(C'_1\) and we reuse the predicate \(C_2\) to address this case:

\[
C'_1 : \mathcal{N}^2 \times i \rightarrow \mathbb{B} \\
(n_r, n_c, i) \mapsto \exists ((x, y), c) \in \mathcal{M} \times \mathcal{C}, (i \rightarrow \mathcal{M}.\delta_l(n_c, n_r) = ((x, y), c) \land x = 0)
\]

Recontextualization is valid if the multiplicity of the deleted reference is not mandatory \((C'_1)\). Recontextualization is also valid if the set of new instances of the class from which the reference has been removed is empty \((C_2)\).

There is a second condition due to the links that have been discarded by the migration and that cannot be recovered by the recontextualization. In this case, the recontextualization preserves validity only if these links are not mandatory in regard to multiplicity. Hence the unique following predicate giving the sufficient condition under which a valid model remains valid over recontextualization:

\[
\overleftarrow{C_{dr}} : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \rightarrow \mathbb{B} \\
(i, n_r, n_c, i_{ini}) \mapsto \{ C'_1(n_r, n_c, i) \lor C_2(n_c, i, i_{ini}) \} \land (i_{ini}.\mathcal{I} \cap \cup_{i \in i_{ini}}(i_{ini}.\delta_l(i)) = \emptyset
\]

If all instances targeted by the discarded links are kept, then this is a sufficient condition to preserve validity.

Set classifier name  The setting of a new classifier name implies to update the instantiation links at the model level. This update is performed in two times.
First, the links are deleted, and then, new links between the instances and the new class are added. There is no effect on other links (attributes or references).

During the recontextualization, the actions performed by the migration are undone: added links are automatically deleted and discarded links are automatically recovered. However, a processing tool may have introduced new instances of the renamed classifier. In this case, the instantiation links from these specific instances have to be replaced by new instantiation links targeting the old classifier.

Only a part of the links that have been discarded during the migration is recovered during the recontextualization. The instantiation link between an instance and the renamed class is not recovered if this instance has been deleted by a tool.

\[
\begin{align*}
\re{sn_c}{r} : \mathfrak{I} \times \mathfrak{N}^2 & \rightarrow \mathfrak{I} \\
(i, n_c, n'_c) & \mapsto \{ i^0, [\delta_{\text{inst}} = \{(i, c) \in \mathfrak{I} \times \mathfrak{m.C} \mid c = n_c \land \delta_{\text{inst}}(i) = n_c\}] \}
\end{align*}
\]

\[
\begin{align*}
\re{sn_c}{a} : \mathfrak{I} \times \mathfrak{N}^2 & \rightarrow \mathfrak{I} \\
(i, n_c, n'_c) & \mapsto \{ i^0, [\delta_{\text{inst}} = \{(i, c) \in \mathfrak{I} \times \mathfrak{N} \mid c = n'_c \land \delta_{\text{inst}}(i) = n_c\}] \}
\end{align*}
\]

\[
\begin{align*}
\re{sn_{i=1}}{r} : \mathfrak{I} \times \mathfrak{N}^2 \times \mathfrak{I} & \rightarrow \mathfrak{I} \\
(i, n_c, n'_c, i_{\text{ini}}) & \mapsto \{ i^0, [\delta_{\text{inst}} = \{(i, c) \in \mathfrak{I} \times \mathfrak{m.C} \mid c = n'_c \land \delta_{\text{inst}}(i) = n'_c\}] \}
\end{align*}
\]

\[
\begin{align*}
\re{sn_{i=1}}{a} : \mathfrak{I} \times \mathfrak{N}^2 \times \mathfrak{I} & \rightarrow \mathfrak{I} \\
(i, n_c, n'_c, i_{\text{ini}}) & \mapsto \{ i^0, [\delta_{\text{inst}} = \{(i, c) \in \mathfrak{I} \times \mathfrak{N} \mid c = n_c \land \delta_{\text{inst}}(i) = n'_c\}] \}
\end{align*}
\]

**Validity** Since only instantiation links are substituted in accordance to a new classifier name, we don’t need any specific condition to preserve the conformance property.

**Set feature name** The setting of a new feature name implies to update the value or reference links at the model level. This update is performed in two times. First, the links are deleted, and then, new links between the instances and the corresponding value (value link) or instance (reference link) are added. There is no effect on instantiation links.

During the recontextualization, the actions performed by the migration are undone: added links are automatically deleted and discarded links are automatically recovered. However, a processing tool may have introduced new instances of the class owning the renamed feature. In this case, the corresponding links from these specific instances have to be replaced by new links targeting the same elements.

Only a part of the links that have been discarded during the migration is recovered during the recontextualization. The value (or the reference) link between an instance and a scalar (or another instance) is not recovered if this instance has been deleted by a tool.
we define precisely the predicates model unchanged during the migration and the recontextualization. In return, we need to put back a new link. when after the migration, a tool removes a former mandatory link. Then during the migration, without adding or deleting anything. But if a tool removes some links, some new links may lack. This is typically the case when an optional feature becomes mandatory. In this case, if we aim at preserving the validity of the model, we need to randomly add some new links.

For instance, it is the case when a mandatory feature becomes optional, and then we need to give the sufficient conditions under which a valid model remains valid over migration and recontextualization:

\[
\begin{align*}
\overline{s_{nf}}^r & : \mathcal{I} \times \mathcal{N}^3 \rightarrow \mathcal{I} \\
(i, n_c, n_f, n'_f) & \mapsto \left\{ \begin{array}{l}
\overline{\delta}_v = \{ (i, a, S) \in \mathcal{I} \times \mathcal{m} \mathcal{A} \times \mathcal{P}(\mathcal{S}) | \\
a = (n_c, n_f') \land \delta_v(i, (n_c, n_f)) = S \} \\
\overline{\delta}_l = \{ (i, r, S) \in \mathcal{I} \times \mathcal{m} \mathcal{R} \times \mathcal{P}(\mathcal{I}) | \\
r = (n_c, n_f') \land \delta_l(i, (n_c, n_f)) = S \} 
\end{array} \right. \\
\overline{s_{nf}}^a & : \mathcal{I} \times \mathcal{N}^3 \rightarrow \mathcal{I} \\
(i, n_c, n_f, n'_f) & \mapsto \left\{ \begin{array}{l}
\overline{\delta}_v = \{ (i, a, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{S}) | \\
a = (n_c, n_f') \land \delta_v(i, (n_c, n_f)) = S \} \\
\overline{\delta}_l = \{ (i, r, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{I}) | \\
r = (n_c, n_f') \land \delta_l(i, (n_c, n_f)) = S \} 
\end{array} \right. \\
\widetilde{s_{nf}}^r & : \mathcal{I} \times \mathcal{N}^3 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_c, n_f, n'_f, i_m) & \mapsto \left\{ \begin{array}{l}
\overline{\delta}_v = \{ (i, a, S) \in \mathcal{I} \times \mathcal{m} \mathcal{A} \times \mathcal{P}(\mathcal{S}) | \\
a = (n_c, n_f') \land \delta_v(i, (n_c, n_f)) = S \} \\
\overline{\delta}_l = \{ (i, r, S) \in \mathcal{I} \times \mathcal{m} \mathcal{R} \times \mathcal{P}(\mathcal{I}) | \\
r = (n_c, n_f') \land \delta_l(i, (n_c, n_f)) = S \} 
\end{array} \right. \\
\widetilde{s_{nf}}^a & : \mathcal{I} \times \mathcal{N}^3 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_c, n_f, n'_f, i_m) & \mapsto \left\{ \begin{array}{l}
\overline{\delta}_v = \{ (i, a, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{S}) | \\
a = (n_c, n_f') \land \delta_v(i, (n_c, n_f)) = S \} \\
\overline{\delta}_l = \{ (i, r, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{I}) | \\
r = (n_c, n_f') \land \delta_l(i, (n_c, n_f)) = S \} 
\end{array} \right. 
\end{align*}
\]

**Validity** Since only value or reference links are substituted in accordance to a new feature name, we don’t need any specific condition to preserve the conformance property.

**Set lower bound** The setting of a new lower bound for a given feature implies to make a semantic choice. Indeed, if the lower bound is increased, then some links may lack. This is typically the case when an optional feature becomes mandatory. In this case, if we aim at preserving the validity of the model, we need to randomly add some new links.

If the lower bound is decreased, then any valid model remains valid after the migration, without adding or deleting anything. But if a tool removes some links, then we need to randomly add some new links during the recontextualization. For instance, it is the case when a mandatory feature becomes optional, and when after the migration, a tool removes a former mandatory link. Then during the recontextualization, we need to put back a new link.

We decided to avoid these random actions. Instead of that, we keep the model unchanged during the migration and the recontextualization. In return, we define precisely the predicates giving the sufficient conditions under which a valid model remains valid over migration and recontextualization:
The migration preserves validity if there is no lack of mandatory links in the case of a lower bound increase ($\overrightarrow{C_{slb}}$). The recontextualization preserves validity if there is no lack of mandatory links in the case of a lower bound decrease ($\overleftarrow{C_{slb}}$).

**Set upper bound** The setting of a new upper bound for a given feature is a symmetric case of the previous one. Instead of randomly adding, we need to propagate several modifications about attributes and references in regard to the corresponding multiplicity.

$$\overrightarrow{C_{sub}} : \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} \rightarrow \mathbb{B}$$

$$(i, n_c, n_f, x) \mapsto \forall i \in i.I, \left(\left|\delta_v(i, (n_c, n_f))\right| + |\delta_l(i, (n_c, n_f))|\right) \geq x$$

$$\overleftarrow{C_{sub}} : \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} \times \mathfrak{I} \rightarrow \mathbb{B}$$

$$(i, n_c, n_f, x, i_{ini}) \mapsto \forall i \in i.I, \left(\left|\delta_v(i, (n_c, n_f))\right| + |\delta_l(i, (n_c, n_f))|\right) \geq x_{ini}$$

The migration preserves validity if there is no lack of mandatory links in the case of a lower bound increase ($\overrightarrow{C_{sub}}$). The recontextualization preserves validity if there is no lack of mandatory links in the case of a lower bound decrease ($\overleftarrow{C_{sub}}$).

**Set abstract** As for the previous operations, setting a class abstract or concrete implies to make a semantic choice. Indeed, if a concrete class is made abstract, then we need to transform its instances. They could be connected to another class among the ancestors or among the descendants of the modified class, but it implies that this class actually have ancestors or descendants. If so, we also need to propagate several modifications about attributes and references in regard to the corresponding multiplicity.

Instead of that, we keep the model unchanged during the migration and the recontextualization. In return, we define precisely the predicates giving the sufficient conditions under which a valid model remains valid over migration and recontextualization.

$$\overrightarrow{C_{sa}} : \mathfrak{I} \times \mathcal{N} \times \mathbb{B} \rightarrow \mathbb{B}$$

$$(i, n_c, b) \mapsto b \implies \{i \in i.I \mid i.\delta_{inst}(i) = n_c\} = \emptyset$$

$$\overleftarrow{C_{sa}} : \mathfrak{I} \times \mathcal{N} \times \mathbb{B} \times \mathfrak{I} \rightarrow \mathbb{B}$$

$$(i, n_c, b, i_{ini}) \mapsto \neg b \implies \{i \in i.I \mid i.\delta_{inst}(i) = n_c\} = \emptyset$$
The migration preserves validity if there is no instance of a concrete class made abstract. The recontextualization preserves validity if there is no instance of a former abstract class (because it has to be set abstract again).

**Set container** Setting a reference *composite* or not is subject to strong preconditions at the metamodel level. Under these conditions, the corresponding migration and recontextualization have no effects, but they are subject to validity conditions. Indeed, in μEcore, all instances are gathered within a same and unique root package. Reference links over these instances are given by partial functions, regardless they are composite or not.

Once the specific constraints of containment are verified at the metamodel level (e.g., no circular composite references, or no multiplicities upper bound greater than 1), there is one extra requirement at the model level: if a relation is made composite, then the instances targeted by the corresponding links cannot be also targeted by other composite links.

\[
C_{sc}^+ : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{B} \to \mathcal{B} \\
(i, n_c, n_r, b) \mapsto b \iff \exists i \in \bigcup_{j \in i.I} (i.\delta_l(j, (n_c, n_r))) \Rightarrow (\exists (i', (n'_c, n'_r)) \in i.I \times i.m.R, i' \in i.\delta_l(i', (n'_c, n'_r))) \Rightarrow (n'_c, n'_r) \notin i.m.R_C
\]

\[
C_{sc}^- : \mathcal{I} \times \mathcal{N} \times \mathcal{B} \times \mathcal{I} \to \mathcal{B} \\
(i, n_c, n_r, b, i_{init}) \mapsto \neg b \iff \exists i \in \bigcup_{j \in i.I} (i.\delta_l(j, (n_c, n_r))) \Rightarrow (\exists (i', (n'_c, n'_r)) \in i.I \times i.m.R, i' \in i.\delta_l(i', (n'_c, n'_r))) \Rightarrow (n'_c, n'_r) \notin i.m.R_C
\]

The migration preserves validity if the links corresponding to a reference made composite target instances that are not already targeted by another composite link. The recontextualization preserves validity if the links corresponding to a former composite reference target instances that are not already targeted by another composite link (because it has to be set composite again).

**Move feature** Moving a feature from a class to another one (provided the feature does not imply name clashes along inheritance links of the target class) implies again a *semantic choice*. Indeed, if a feature \(f\) belongs to a class \(A\), and if this feature has to be moved to class \(B\), then if \(A\) has instances, we need to remove the \(f\) links from them, but we also need to put them randomly over the set of existing instances of \(B\). If the number of instances of \(B\) is lower than the number of instances of \(A\), some links will be randomly deleted. If the number of instances of \(B\) is greater than the number of instances of \(A\), some links will be randomly added if the moved feature is mandatory.

Instead of performing such random actions, we keep the model unchanged during the migration and the recontextualization. In return, we define precisely the *predicates* giving the sufficient conditions under which a valid model remains valid over migration and recontextualization:
The migration preserves validity if there is no link corresponding the moved feature (from the initial class). The recontextualization preserves validity if there is no link corresponding the moved feature (from the targeted class).

**Move reference target** Moving the target of a reference from a class to another one raises similar questions to those of the previous case. Indeed, to achieve a model transformation in regard to this operator, we need to change randomly the current targets of reference links to new targets.

As in the previous case, we avoid these random actions and we keep the model unchanged during the migration and the recontextualization. As in the previous case, we define precisely the *predicates* giving the sufficient conditions under which a valid model remains valid over migration and recontextualization:

\[
\overrightarrow{C_{mft}} : \mathcal{I} \times \mathcal{N}^3 \rightarrow \mathcal{B} \\
(i, n_c, n_f, n'_c) \mapsto \left\{ \bigcup_{j \in \mathcal{I}} \left( i, (\delta_v \cup \delta_l)(j, (n_c, n_f)) \right) \right\} = \emptyset
\]

\[
\overrightarrow{C_{mrtt}} : \mathcal{I} \times \mathcal{N}^3 \mathcal{B} \times \mathcal{I} \rightarrow \mathcal{B} \\
(i, n_c, n_f, n'_c, i_{ini}) \mapsto \left\{ \bigcup_{j \in \mathcal{I}} \left( i, (\delta_v \cup \delta_l)(j, (n'_c, n_f)) \right) \right\} = \emptyset
\]

The migration preserves validity if there is no link corresponding the modified reference. The recontextualization preserves validity if there is no new link corresponding the same moved modified reference.

**Move attribute type** Moving the type of an attribute from a data type to another one is a much simpler case than the previous one since \(\mu\text{Dif} \) does not take data types into account. In this approach, scalar values (at the model level) are not bound to a data type (at the metamodel level). Thus, there is no need to change anything at the model level when a data type is updated.

Validity Since nothing is changed at the model level when a data type is updated, we obviously don’t need any specific condition to preserve the *conformance* property.

**Add super class** Adding a new super class to a class does not imply any modification at the model level during the migration, provided the new class or
one of its ancestors has no mandatory feature. If this condition is not satisfied, then we do not add random feature links and thus, the migration does not preserve the conformance property.

During the recontextualization, the links corresponding to the features defined by the new super class have to be removed.

\[ \delta_{\text{asc}}^r : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \rightarrow \mathcal{I} \]

\[ (i, n_c, n_c', \text{ini}) \mapsto \left\{ \begin{array}{l}
i^0. [\delta_v = \{(i, (c, n_a), S) \in \mathcal{I} \times \mathcal{M} \times \mathcal{P}(S) | \\
\delta_{\text{inst}}(i) = n_c \land \delta_c(i, (c, n_a)) = S \land c \in \mathcal{M} \alpha'_f(n_c')\\
.\delta_t = \{(i, (c, n_r), S) \in \mathcal{I} \times \mathcal{M} \times \mathcal{P}(S) | \\
\delta_{\text{inst}}(i) = n_c \land \delta_t(i, (c, n_r)) = S \land c \in \mathcal{M} \alpha'_f(n_c')\} \end{array} \right. \]

Validity The recontextualization always preserves the conformance property since the only discarded elements corresponds to features inherited from the class which is removed by this transformation.

But as mentioned before, the migration preserve the conformance property only if the new super class or one of its ancestors has no mandatory feature. Hence the unique following predicate giving the sufficient condition under which a valid model remains valid over migration:

\[ \overrightarrow{C_{\text{asc}}} : \mathcal{I} \times \mathcal{N}^2 \rightarrow \mathbb{B} \]

\[ (i, n_c, n_c') \mapsto \forall c \in \text{i.m} \alpha'_f(n_c'), \forall((n_f, n), (x, y)) \in \mathcal{N}^2 \times \mathcal{M} \\
((x, y), n) = \text{i.m} (\delta_A \cup \delta_R)(c, n_f) \implies x = 0 \]

Remove super class Removing an existing super class implies to remove links corresponding to the features inherited from the deleted super class. There is no new specific element to add during the migration.

During the recontextualization, the links that have been discarded by the migration are recovered, as far as they are related to existing instances. If the source or the target of the link has been deleted, then the link is not recovered.

\[ \overrightarrow{r_{\text{asc}}} : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \rightarrow \mathcal{I} \]

\[ (i, n_c, n_c') \mapsto \left\{ \begin{array}{l}
i^0. [\delta_v = \{(i, (c, n_a), S) \in \mathcal{I} \times \mathcal{M} \times \mathcal{P}(S) | \\
\delta_{\text{inst}}(i) = n_c \land \delta_c(i, (c, n_a)) = S \land c \in \text{i.m} \alpha'_f(n_c')\\
.\delta_t = \{(i, (c, n_r), S) \in \mathcal{I} \times \mathcal{M} \times \mathcal{P}(S) | \\
\delta_{\text{inst}}(i) = n_c \land \delta_t(i, (c, n_r)) = S \land c \in \text{i.m} \alpha'_f(n_c')\} \end{array} \right. \]

\[ \overrightarrow{f_{\text{asc}}} : \mathcal{I} \times \mathcal{N}^2 \rightarrow \mathcal{I} \]

\[ (i, n_c, n_c', \text{ini}) \mapsto \left\{ \begin{array}{l}
i^0. [\delta_v = \{(i, (c, n_a), S) \in \text{i.ini} \delta_v | i \in \text{i.I} \\
\land \text{i.ini} \delta_{\text{inst}}(i) = n_c \land c \in \text{i.ini.m} \alpha'_f(n_c')\\
.\delta_t = \{(i, (c, n_r), S) \in \text{i.ini.I} \times \text{i.ini.m} \mathcal{R} \times \mathcal{P}(\text{i.ini.I}) | \\
i \in \text{i.I} \land \text{i.ini} \delta_{\text{inst}}(i) = n_c \land c \in \text{i.ini.m} \alpha'_f(n_c')\\
\land \forall i' \in S, (i' \in \text{i.ini}, \delta_t(i, (c, n_r)) \land i' \in \text{i.I})\} \end{array} \right. \]
Validity The migration always preserves the conformance property since the only discarded elements corresponds to features inherited from the class which is removed by this transformation.

But the recontextualization preserve the conformance property only if:

- the deleted super class or one of its ancestors has no mandatory feature
- or there is no new instance of the modified class

Indeed, if a tool has introduced new instances of the modified class before the recontextualization, then the specific links inherited from the discarded super class cannot be recovered. This is a problem only if these links corresponds to mandatory features. Hence the unique following predicate giving the sufficient condition under which a valid model remains valid over recontextualization:

\[
C_{rsc} : \mathcal{I} \times \mathcal{N}^2 \times \mathcal{I} \rightarrow \mathcal{B}
\]

\[
(i, n_c, n'_c, i_{ini}) \mapsto \big( \forall c \in i_{ini}.m, \alpha'_i(n'_c), \forall((n_f, n), (x, y)) \in \mathcal{N}^2 \times \mathcal{M} (\exists x = 0)
\]

Move super class The replacement of an existing link between a class and a super class by a link between the same class and another one could be seen as the composition of the two previous operator (super class deletion followed by new super class addition). In this approach, we do not take advantage of an important specific information: the set of common features. Indeed, the common features (i.e. same name and same target) can be kept during the transformation.

For that purpose, we first formally define this set of common features. We note \(\cap_a\) the common attributes between two metamodels and we note \(\cap_r\) the common references between two classes.

The common attributes are found along the inheritance paths (starting from the two provided classes). Two attributes match if they have the same name, the same data type, and the same multiplicity.

As for attributes, the common references are found along the inheritance paths (starting from the two provided classes). Two references match if they have the same name, the same multiplicity, and if the first targeted class appears among the ancestors of the second targeted class.

\[
\cap_a : \mathcal{M} \times \mathcal{C}^2 \rightarrow \mathcal{A} \times \mathcal{M} \times \mathcal{D}
\]

\[
(m, (c_1, c_2)) \mapsto \{(c, n), m, d) \in m.\delta_A | c \in \alpha'_i(c_1) \land \exists c' \in \mathcal{C} (m.\delta_A(c', n) = (m, d) \land c' \in \alpha'_i(c_2))
\]

\[
\cap_r : \mathcal{M} \times \mathcal{C}^2 \rightarrow \mathcal{R} \times \mathcal{M} \times \mathcal{C}
\]

\[
(m, (c_1, c_2)) \mapsto \{(c, n), m, c_i) \in m.\delta_R | c \in \alpha'_i(c_1) \land \exists c' \in \mathcal{C}^2, (m.\delta_R(c', n) = (m, c') \land c' \in \alpha'_i(c_2)) \land c_i \in \alpha'_i(c_2))
\]
Using these definitions, we can state that the migration corresponding to the replacement of an existing link between a class and a super class by a link between the same class and another one implies to remove links corresponding to the features inherited from the deleted super class, provided they are not common with the new super class.

The recontextualization implies to remove the links corresponding to the features defined by the new super class, provided they are not common with the new super class.

Finally, during the recontextualization, the links that have been discarded by the migration are recovered, as far as they are related to existing instances. If the source or the target of the link has been deleted, then the link is not recovered.

\[
\begin{align*}
\bar{\text{mst}}^r &: \mathcal{J} \times \mathcal{N}^3 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_c, n'_c, n''_c) &\mapsto \begin{cases} 
\text{i}^\emptyset[\delta_v = ((i, c, n_a), S) \in i.\delta_v \mid i.\delta_{\text{inst}}(i) = n_c \\
\land \forall (m, n), (c, n_a, m, n) \notin \cap_d (i.m, n'_c, n''_c) \\
\land c \in m.a_i'(n'_c)] \\
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
\bar{\text{mst}}^r &: \mathcal{J} \times \mathcal{N}^3 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_c, n'_c, n''_c, i_{\text{ini}}) &\mapsto \begin{cases} 
\text{i}^\emptyset[\delta_v = ((i, c, n_a), S) \in i.\delta_v \mid i.\delta_{\text{inst}}(i) = n_c \\
\land \forall (m, n), (c, n_a, m, n) \notin \cap_d (i.m, n''_c, n'_c) \\
\land c \in m.a_i'(n''_c)] \\
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
\bar{\text{mst}}^a &: \mathcal{J} \times \mathcal{N}^3 \times \mathcal{I} \rightarrow \mathcal{I} \\
(i, n_c, n'_c, n''_c, i_{\text{ini}}) &\mapsto \begin{cases} 
\text{i}^\emptyset[\delta_v = ((i, c, n_a), S) \in i_{\text{ini}}.\delta_v \mid i \in i.\mathcal{I} \\
\land i_{\text{ini}}.\delta_{\text{inst}}(i) = n_c \land c \in i_{\text{ini}}.m.a_i'(n'_c) \\
\land \forall (m, n), (c, n_a, m, n) \notin \cap_d (i.m, n''_c, n'_c)] \\
\end{cases} \\
\end{align*}
\]

Validity The migration preserve the conformance property only if the new super class or one of its ancestors has no mandatory feature among its specific features (i.e. regardless of the common features).

The recontextualization preserve the conformance property only if the replaced super class or one of its ancestors has no mandatory feature among its specific features (i.e. regardless of the common features), or there is no new instance of the modified class. Indeed, if a tool has introduced new instances of the modified class before the recontextualization, then the specific links inherited
from the initial super class cannot be recovered. This is a problem only if these links correspond to mandatory features.

Hence the following predicates giving the sufficient conditions under which a valid model remains valid over migration and recontextualization:

\[
\overline{C_{mset}} : \mathcal{J} \times \mathcal{N} \rightarrow \mathcal{B}
\]

\[
(i, n, n', n'') \rightarrow \forall c \in i.m.\alpha'_f(n''), \forall ((n_f, n), (x, y)) \in \mathcal{N} \times \mathcal{M}
\]

\[
\left( ((c, n_f), (x, y), n) \notin (\cap_{\alpha} \cup \cap_{\gamma})(i.m, n', n'') \right)
\]

\[
\wedge ((x, y), n) = i.m.\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}}(c, n_f) \implies x = 0
\]

\[
\overline{C_{mset}} : \mathcal{J} \times \mathcal{N} \times \mathcal{J} \rightarrow \mathcal{B}
\]

\[
(i, n, n', n'', i_{\text{ini}}) \rightarrow \left( \forall c \in i_{\text{ini}}.m.\alpha'_f(n''), \forall ((n_f, n), (x, y)) \in \mathcal{N} \times \mathcal{M}
\]

\[
\left( ((c, n_f), (x, y), n) \notin (\cap_{\alpha} \cup \cap_{\gamma})(i.m, n'', n'_f) \right)
\]

\[
\wedge ((x, y), n) = i.m.\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}}(c, n_f) \implies x = 0
\]

\[
\forall \left( \{ i \in i.I | i.\delta_{\text{inst}}(i) = n_c \} \right)
\]

\[
\setminus \{ i \in i_{\text{ini}}.I | i_{\text{ini}}.\delta_{\text{inst}}(i) = n_c \} = \emptyset
\]

**Move opposite** The setting of a new opposite reference to a reference (whenever it already has an opposite reference or not) is not supposed to have any impact at the model level since opposite corresponds to a meta data associated to a pair of existing references. Thus, there is nothing to add or remove during the migration and the recontextualization.

**Remove opposite** For the same reasons as in the previous case, removing an opposite reference is not supposed to have any impact at the model level. Thus, there is nothing to add or remove during the migration and the recontextualization.

\[
[.]_{\text{mig}} : \mathcal{L}(\text{mig}) \rightarrow \mathcal{J}
\]

\[
s \mapsto \begin{cases} 
\{ \text{mod} \} & : \text{s matches with (\{ mod \} op)} \\
\overline{[op]}_{\text{op}}(\overline{[\text{mod}]}_{\text{mod}} \cdot \overline{[\text{op}]}_{\text{param}}) & : \text{s matches with (s' op) where s' = (\{ mod \} (op')^+)}
\end{cases}
\]

Fig. 31. Valuation of \(\mu\text{Diff}\) specifications including model migration

4.7 Specifications

We note \(\mathcal{L}(\text{mig})\) the sets of words yielded from \(\text{mig}\) in figure 30. Basically, \(\mathcal{L}(\text{mig})\) contains a specification made of one model \(\text{mod}\) (including its metamodel) plus an ordered non-empty set of operators applied to \(\text{mod}\).
As for the semantics of metamodel evolution, we note \([op]_{param}\) the set of specific parameters of the operator \(op\), in accordance to figure 19.

We introduce in figure 31 a valuation function noted \([\cdot]_{mig}\). It applies to \(L(mig)\). It maps a model (including its metamodel) to an output migrated model (including its refactored metamodel). It is obtained by a recursive application of the functional denotations corresponding to each operator in accordance to figure 19.

References

List of Figures

1  Ecore metamodel ............................................. 2
2  μEcore metamodel ............................................. 3
3  Textual syntax of μEcore .................................... 3
4  Textual and graphical form of a μEcore metamodel ......... 3
5  μEcore semantics: name spaces ............................... 4
6  μEcore semantics: multiplicity ............................... 4
7  μEcore semantics: partial functions ......................... 5
8  μEcore semantics: metamodels ............................... 5
9  μEcore semantics: valuation .................................. 6
10 Semantics of a μEcore metamodel ........................... 6
11 Graph view of the semantics of a μEcore metamodel ..... 6
12 Textual syntax of μDif ....................................... 7
13 μDif creation ................................................ 8
14 μDif deletion ................................................ 9
15 μDif update .................................................. 9
16 μDif value update .......................................... 10
17 μDif containment update .................................. 10
18 μDif link update ........................................... 11
19 Valuation of μDif operators .................................. 12
20 Valuation of μDif specifications ............................. 20
21 Textual syntax of μEcore models ............................ 20
22 Textual and graphical form of a μEcore model .......... 21
23 μEcore models semantics: name spaces .................... 22
24 μEcore model semantics: partial functions ................ 22
25 μEcore model semantics ..................................... 22
26 μEcore model semantics: valuation ......................... 23
27 Semantics of a μEcore model ................................. 23
28 Graph view of the semantics of a μEcore model .......... 24
29 Conformance property of μEcore models .................... 25
30 Textual syntax of μDif including model migration ......... 26
31 Valuation of μDif specifications including model migration .... 47
# Table of Contents

A kernel transformation language for metamodel evolution and reversible model co-evolution ........................................ 1  
*Mickaël Kerboeuf, Paola Vallejo, and Jean-Philippe Babau*

1 **µEcore** .......................................................... 1
   1.1 Textual syntax of **µEcore** .................................. 1
   1.2 Denotational semantics of **µEcore** ......................... 4
      Semantic domain .............................................. 4
      Valuation function ........................................ 5
   1.3 Example ........................................................ 5

2 Metamodel evolution with **µDif** ................................ 7
   2.1 Overview of **µDif** ........................................... 7
   2.2 **µDif** creation ............................................... 8
   2.3 **µDif** deletion ............................................... 8
   2.4 **µDif** update ................................................. 9
      Value update ................................................. 9
      Containment update ....................................... 10
      Link update ................................................ 10
   2.5 **µDif** semantics ............................................ 12
   2.6 Notations ....................................................... 13
      Metamodel component ...................................... 13
      Union of metamodel component ............................ 13
      Substitution of metamodel component .................... 13
      General substitution ...................................... 13
      Classifier substitution ................................... 14
      Direct ancestors ........................................... 14
      All ancestors ............................................... 14
   2.7 Operators ..................................................... 14
      Create class ............................................... 14
      Create data type .......................................... 14
      Create attribute .......................................... 15
      Create reference ......................................... 15
      Delete class ............................................... 15
      Delete data type .......................................... 15
      Delete attribute .......................................... 16
      Delete reference ......................................... 16
      Set classifier name ....................................... 16
      Set feature name .......................................... 16
      Set lower bound ........................................... 16
      Set upper bound .......................................... 17
      Set abstract ............................................... 17
      Set container .............................................. 17
3 μEcore models
3.1 Syntax extension
3.2 Semantics of a μEcore model
3.3 Semantic domain
3.4 Valuation function
3.5 Example
3.6 Conformity
4 Model co-evolution with μDif
4.1 Syntax extension
4.2 Notations
4.3 Principles of reversible model migration
4.4 Diagnostics
4.5 By-default model migration
4.6 Model migration by operator in detail
Move feature
Move reference target
Move attribute type
Add super class
Remove super class
Move super class
Move opposite
Remove opposite
2.8 Specifications
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set lower bound</td>
<td>40</td>
</tr>
<tr>
<td>Set upper bound</td>
<td>41</td>
</tr>
<tr>
<td>Set abstract</td>
<td>41</td>
</tr>
<tr>
<td>Set container</td>
<td>42</td>
</tr>
<tr>
<td>Move feature</td>
<td>42</td>
</tr>
<tr>
<td>Move reference target</td>
<td>43</td>
</tr>
<tr>
<td>Move attribute type</td>
<td>43</td>
</tr>
<tr>
<td>Add super class</td>
<td>43</td>
</tr>
<tr>
<td>Remove super class</td>
<td>44</td>
</tr>
<tr>
<td>Move super class</td>
<td>45</td>
</tr>
<tr>
<td>Move opposite</td>
<td>47</td>
</tr>
<tr>
<td>Remove opposite</td>
<td>47</td>
</tr>
</tbody>
</table>

4.7 Specifications ............................................. 47