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

Mickaël Kerboeuf, Paola Vallejo, Jean-Philippe Babau

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 μDif, a kernel transformation language for metamodel evolution and reversible model co-evolution. To begin with, a kernel subset of Ecore is highlighted and formally defined thanks to a suitable denotational semantics. Then μDif 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 μDif operator is also provided with a dedicated predicate which explains the sufficient conditions for a model to remain valid after these transformations.

1 μEcore

Figure 1 depicts the metamodel of 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 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 Ecore concepts like *uniqueness* or *order* for attributes are discarded. In the end, we obtain the simplified version of Ecore we called μEcore, and whose metamodel is depicted by figure 2.

1.1 Textual syntax of μEcore

In order to make easier the formal definition of μDif, we introduce in figure 3 a feather light textual syntax for metamodels conforming to the μEcore metametamodel 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

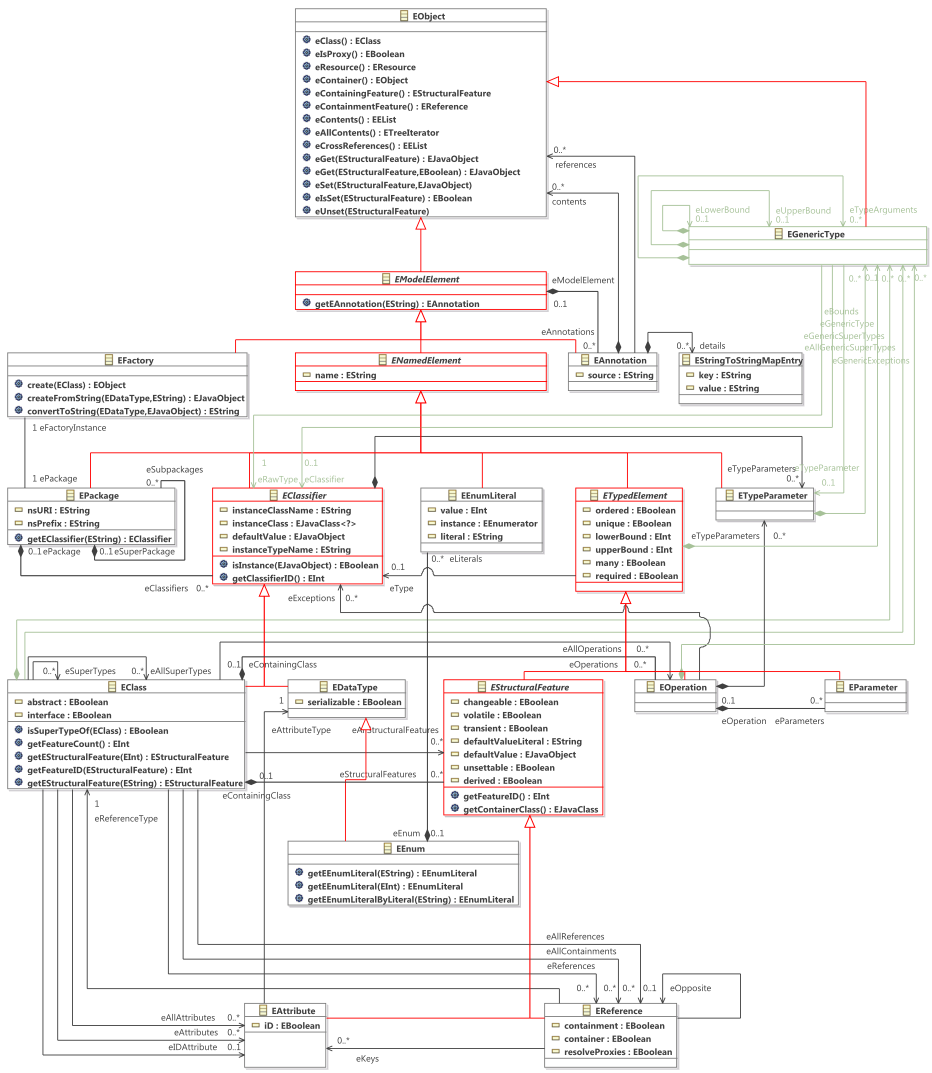

Fig. 1. Ecore metamodel

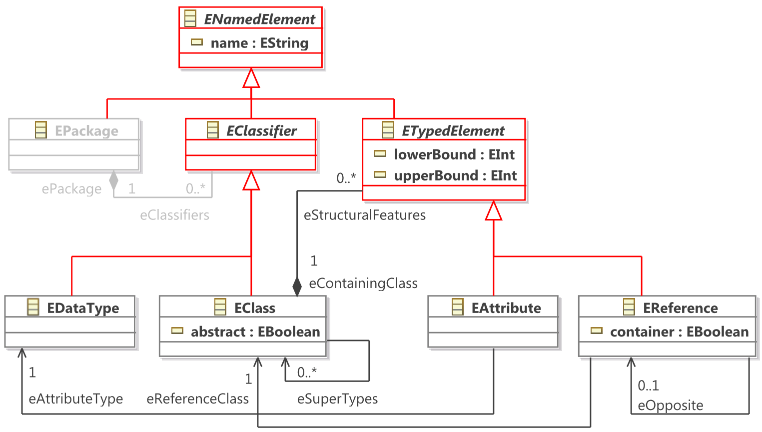

Fig. 2. µEcore metamodel

$$
\begin{array}{rll}
n, n_{\mathsf{class}}, n_{\mathsf{type}}, n_{\mathsf{attrib}}, n_{\mathsf{ref}} \in \mathcal{N} & & \text{(name)} \\
m \in \mathcal{M} = \{(x,y) \in \mathbb{N} \times (\mathbb{N}^* \cup \{\infty\}) | x < y\} & & \text{(multiplicity)} \\
 & & \\
mm ::= (\ c \mid d\ )^* & & \text{(metamodel)} \\
c ::= \langle\ (\mathsf{a})^?\ n\ (\ /\ n_{\mathsf{class}}\ )^*\ (\ a \mid r\ )^*\ \rangle_{\mathsf{c}} & & \text{(class)} \\
d ::= \langle\ n\ \rangle_{\mathsf{d}} & & \text{(data type)} \\
a ::= [\ n\ ,\ m\ ,\ n_{\mathsf{type}}\ ]_{\mathsf{a}} & & \text{(attribute)} \\
r ::= [\ (\mathsf{c})^?\ n\ ,\ m\ ,\ n_{\mathsf{class}}\ (\leftarrow n_{\mathsf{ref}})^?\ ]_{\mathsf{r}} & & \text{(reference)}
\end{array}
$$

Fig. 3. Textual syntax of µEcore

$$
\begin{array}{l}
\langle\ \mathsf{Int}\ \rangle_{\mathsf{d}}\ \ \langle\ \mathsf{Bool}\ \rangle_{\mathsf{d}} \\
\langle\ \mathsf{a}\ \mathsf{A}\ [\mathsf{i}, (0,1), \mathsf{Int}]_{\mathsf{a}}\ [\mathsf{b}, (0,\infty), \mathsf{Bool}]_{\mathsf{a}}\ [\mathsf{y}, (0,1), \mathsf{Y}\ \leftarrow \mathsf{a}]_{\mathsf{r}}\ \rangle_{\mathsf{c}} \\
\langle\ \mathsf{X}\ /\ \mathsf{A}\ [\mathsf{j}, (1,1), \mathsf{Int}]_{\mathsf{a}}\ \rangle_{\mathsf{c}} \\
\langle\ \mathsf{Y}\ /\ \mathsf{A}\ [\mathsf{c}\ \mathsf{a}, (1,2), \mathsf{A}\ \leftarrow \mathsf{y}]_{\mathsf{r}}\ [\mathsf{x}, (0,\infty), \mathsf{X}]_{\mathsf{r}}\ \rangle_{\mathsf{c}}
\end{array}
$$

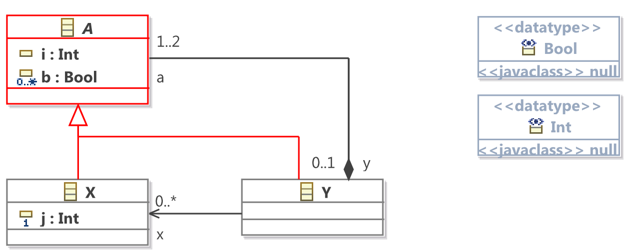

Fig. 4. Textual and graphical form of a µEcore metamodel

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 ($\leftarrow$).

As an illustration, figure 4 shows a metamodel conforming to µEcore together with its equivalent textual specification.

1.2 Denotational semantics of µEcore

The formal semantics of µEcore 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 corresponds to *alphabets*, *i.e.* finite non empty sets of symbols.

$$
\begin{array}{lll}
\mathcal{N} & : \text{named elements} & \\
\mathcal{C} \subseteq \mathcal{N} & : \text{classes} & \mathcal{C}_{\mathcal{A}} \subseteq \mathcal{C} \ : \text{abstract classes} \\
\mathcal{D} \subseteq \mathcal{N} \setminus \mathcal{C} & : \text{data types} & \\
\mathcal{R} \subseteq \mathcal{C} \times \mathcal{N} & : \text{references} & \mathcal{R}_{\mathcal{C}} \subseteq \mathcal{R} : \text{composite references} \\
\mathcal{A} \subseteq (\mathcal{C} \times \mathcal{N}) \setminus \mathcal{R} & : \text{attributes} &
\end{array}
$$

Fig. 5. µEcore semantics: name spaces

Figure 6 defines *multiplicity*, which is by definition a set of pairs composed of natural numbers extended with the special mark ∞[3]. This definition excludes the irrelevant multiplicities $(0, 0)$ and (∞, ∞).

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

Fig. 6. µEcore 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 $\mathfrak{f} \in \mathfrak{F}$ (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.

[3] by definition, $\forall n \in \mathbb{N}, n < \infty$

$$
\begin{array}{lll}
\delta_I & : \mathcal{C} \rightharpoonup \mathcal{C} & \text{inheritance link} \\
\delta_A & : \mathcal{A} \rightharpoonup \mathcal{M} \times \mathcal{D} & \text{attribute link} \\
\delta_R & : \mathcal{R} \rightharpoonup \mathcal{M} \times \mathcal{C} & \text{reference link} \\
\delta_{opp} & : \mathcal{R} \rightharpoonup \mathcal{R} & \text{opposite link}
\end{array}
$$

Fig. 7. μEcore semantics: partial functions

$$
\begin{array}{lll}
\mathfrak{M} \triangleq \mathfrak{N} \times \mathfrak{F} & & \text{metamodels} \\
\mathfrak{N} \triangleq \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C_A}) \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{R}) \times \mathcal{P}(\mathcal{R_C}) \times \mathcal{P}(\mathcal{A}) & & \text{name spaces} \\
\mathfrak{F} \triangleq \mathcal{P}(\delta_I) \times \mathcal{P}(\delta_A) \times \mathcal{P}(\delta_R) \times \mathcal{P}(\delta_{opp}) & & \text{partial functions}
\end{array}
$$

Fig. 8. μEcore semantics: metamodels

Valuation function

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

$$\mathsf{pattern} \sqsubset mm$$

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: $\mathsf{pattern} \in \mathcal{L}(c) \cup \mathcal{L}(d)$.

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 $(0, 1)$:

$$\langle\ \mathsf{a}\ n_1\ [n_2, (0,1), n_3]_\mathsf{a}\ \rangle_\mathsf{c} \quad \sqsubset \quad mm$$

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

$$mm \triangleq \big(\ \langle\ n_3\ \rangle_\mathsf{d}\ \langle\ \mathsf{a}\ n_1\ /\ n_4\ [n_2, (0,1), n_3]_\mathsf{a}\ [n_r, (0,\infty), n_4]_\mathsf{r}\ \rangle_\mathsf{c}\ \langle\ \mathsf{a}\ n_4\ \rangle_\mathsf{c}\ \big)$$

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 μEcore to the semantic domain $\mathfrak{M}$ defined by figure 8.

1.3 Example

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

Figure 10 shows $\mathfrak{m}$ in details, and figure 11 shows a *graph-based* representation of it. In this representation, the different name spaces $\mathfrak{n}$ corresponding to classes, data types, attributes and references are depicted by four kinds of dedicated *vertices*.

$$
\begin{aligned}
[\![\cdot]\!]_{mm} \;:\; & \mathcal{L}(mm) \to \mathfrak{M} \\
& x \mapsto [\![x]\!]_{mm} = \big((\mathcal{C}, \mathcal{C}_{\mathcal{A}}, \mathcal{D}, \mathcal{R}, \mathcal{R}_{\mathcal{C}}, \mathcal{A}), (\delta_I, \delta_A, \delta_R, \delta_{opp}) \big) \quad \text{where:}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C} &= \{n \in \mathcal{N} \mid \langle\; n \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\mathcal{C}_{\mathcal{A}} &= \{n \in \mathcal{N} \mid \langle\; \mathsf{a}\; n \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\mathcal{D} &= \{n \in \mathcal{N} \mid \langle\; n \;\rangle_{\mathsf{d}} \sqsubset x\} \\
\mathcal{R} &= \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, \exists n_3 \in \mathcal{N}, \langle\; n_1\; [n_2, m, n_3]_{\mathsf{r}} \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\mathcal{R}_{\mathcal{C}} &= \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, \exists n_3 \in \mathcal{N}, \langle\; n_1\; [\mathsf{c}\; n_2, m, n_3]_{\mathsf{r}} \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\mathcal{A} &= \{(n_1, n_2) \in \mathcal{N}^2 \mid \exists m \in \mathcal{M}, \exists n_3 \in \mathcal{N}, \langle\; n_1\; [n_2, m, n_3]_{\mathsf{a}} \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\\
\delta_I &= \{(n_1, n_2) \in \mathcal{C}^2 \mid \langle\; n_1 \;/\; n_2 \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\delta_A &= \{((n_1, n_2), (m, n_3)) \in \mathcal{A} \times (\mathcal{M} \times \mathcal{D}) \mid \langle\; n_1\; [n_2, m, n_3]_{\mathsf{a}} \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\delta_R &= \{((n_1, n_2), (m, n_3)) \in \mathcal{R} \times (\mathcal{M} \times \mathcal{C}) \mid \langle\; n_1\; [n_2, m, n_3]_{\mathsf{r}} \;\rangle_{\mathsf{c}} \sqsubset x\} \\
\delta_{opp} &= \{((n_1, n_2), (n_3, n_o)) \in \mathcal{R}^2 \mid \exists m \in \mathcal{M}, \langle\; n_1\; [n_2, m, n_3 \;\leftarrow n_o\;]_{\mathsf{r}} \;\rangle_{\mathsf{c}} \sqsubset x\}
\end{aligned}
$$

Fig. 9. µEcore semantics: valuation

$$
[\![x]\!]_{mm} = \mathfrak{m} = (\mathfrak{n}, \mathfrak{f}) = \big((\mathcal{C}, \mathcal{C}_{\mathcal{A}}, \mathcal{D}, \mathcal{R}, \mathcal{R}_{\mathcal{C}}, \mathcal{A}) \;,\; (\delta_I, \delta_A, \delta_R, \delta_{opp}) \big) \quad \text{where:}
$$

$$
\begin{aligned}
\mathcal{C} &= \{A, X, Y\} \quad ; \quad \mathcal{C}_{\mathcal{A}} = \{A\} \\
\mathcal{D} &= \{\mathsf{Bool}, \mathsf{Int}\} \\
\mathcal{R} &= \{(Y, a), (Y, x), (A, y)\} \quad ; \quad \mathcal{R}_{\mathcal{C}} = \{(Y, a)\} \\
\mathcal{A} &= \{(A, i), (A, b), (X, j)\} \\
\\
\delta_I &= \{X \mapsto A, Y \mapsto A\} \\
\delta_A &= \{(A, i) \mapsto ((0, 1), \mathsf{Int}), (A, b) \mapsto ((0, \infty), \mathsf{Bool}), (X, j) \mapsto ((1, 1), \mathsf{Int})\} \\
\delta_R &= \{(Y, a) \mapsto ((1, 2), A), (Y, x) \mapsto ((0, \infty), X), (A, y) \mapsto ((0, 1), Y)\} \\
\delta_{opp} &= \{(A, y) \mapsto (Y, a), (Y, a) \mapsto (A, y)\}
\end{aligned}
$$

Fig. 10. Semantics of a µEcore metamodel

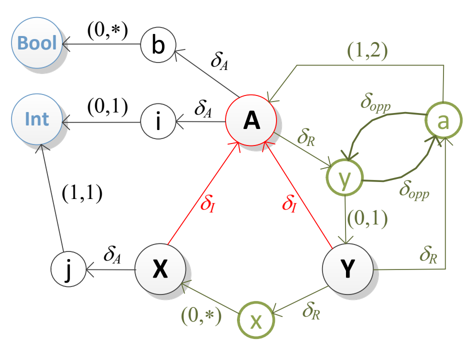

Fig. 11. Graph view of the semantics of a µEcore metamodel

Edges represent (and are labeled by) the partial functions of $\mathfrak{f}$. For instance, $(n_1 \mapsto n_2) \in \delta_I$ is represented by an edge from a *class* vertex n_1 to a *class* vertex n_2. This edge is labeled by δ_I.

Partial functions δ_A and δ_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 δ_A. The second one is labeled by m.

Partial functions δ_{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 δ_{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 δ_R.

2 Metamodel evolution with µDif

µDif is a refactoring language whose scope encompass metamodels conforming to µEcore. It is intended to perform reversible model migration together with metamodel refactoring. In this section, we focus on metamodel refactoring.

2.1 Overview of µDif

µDif is basically a metamodel refactoring language gathering CRUD operations. A µDif specification is an ordered sequence of refactoring operators applied to an input µEcore metamodel. Therefore, each µDif operator has an implicit parameter corresponding to a µEcore metamodel. We name *context* and note mm this metamodel associated to each µDif operator.

$$\begin{array}{rll} spec ::= & \{\ mm\ \}\ (op)^+ & \text{(specification)} \\ op ::= & cr \mid u \mid d & \text{(operator)} \end{array}$$

Fig. 12. Textual syntax of µDif

Figure 12 is a partial view of the syntax of µDif. It extends the textual syntax of µEcore 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 µDif in regard to the semantic domains we introduced for µEcore.

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.

$$
\begin{array}{rll}
cr ::= & \mathsf{createClass}(n) & \text{(creation)} \\
& \mid \mathsf{createDataType}(n) & \\
& \mid \mathsf{createAttribute}(n, n_{\mathsf{class}}, n_{\mathsf{type}}) & \\
& \mid \mathsf{createReference}(n, n_{\mathsf{class}}, n_{\mathsf{targetedClass}}) &
\end{array}
$$

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:

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

$$\begin{aligned} d ::= \ & \mathsf{deleteClass}(n_{\mathsf{class}}) \qquad\qquad \text{(deletion)} \\ & \mid \mathsf{deleteDataType}(n_{\mathsf{type}}) \\ & \mid \mathsf{deleteAttribute}(n_{\mathsf{attrib}}, n_{\mathsf{class}}) \\ & \mid \mathsf{deleteReference}(n_{\mathsf{ref}}, n_{\mathsf{class}}) \end{aligned}$$

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($n_{\mathsf{attrib}}, n_{\mathsf{class}}$) deletes the attribute named n_{attrib} in the class of mm named n_{class}; no precondition
- deleteReference($n_{\mathsf{ref}}, n_{\mathsf{class}}$) 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.

$$u ::= vu \mid cu \mid lu \quad \text{(update)}$$

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(($n \mid n_{\mathsf{class}}.n$) , 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}.

$$\begin{aligned} vu ::= \;& \mathsf{setName}(\ (n \mid n_{\mathsf{class}}.n\ )\ ,\ n_{\mathsf{new}}\ ) \quad \text{(value update)} \\ & \mid \mathsf{setLowerBound}(n_{\mathsf{class}}.n, i) \\ & \mid \mathsf{setUpperBound}(n_{\mathsf{class}}.n, i) \\ & \mid \mathsf{setAbstract}(n_{\mathsf{class}}, b) \\ & \mid \mathsf{setContainer}(n_{\mathsf{class}}.n, b) \end{aligned}$$

Fig. 16. µDif value update

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

Containment update Containment update is related to targets of *compositions* in the metametamodel. In the metamodel of µEcore (fig. 2), it corresponds to a unique element, namely 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.

$$cu ::= \mathsf{moveFeatureTo}(n_{\mathsf{class}}.n, n_{\mathsf{oClass}}) \quad \text{(containment update)}$$

Fig. 17. µDif containment update

Figure 17 introduces the syntax of this unique *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:

- moveFeatureTo($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}$) moves the feature n (attribute or reference) of class n_{class} to another class of mm named n_{oClass}; precondition: n does not already appear among the features associated to n_{oClass} 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 *creation* and *deletion operators* (see figures 13 and 14).

Link update Link update is related to targets of *non-composite references* in the metametamodel. In the metamodel of µEcore (fig. 2), it corresponds to eAttributeType, 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.

$$
\begin{array}{rll}
lu ::= & mlu \mid clu \mid olu & \text{(link update)} \\[1ex]
mlu ::= & \mathsf{moveReferenceTargetTo}(n_{\mathsf{class}}.n, n_{\mathsf{oClass}}) & \text{(mandatory link update)} \\
& \mid \mathsf{moveAttributeTypeTo}(n_{\mathsf{class}}.n, n_{\mathsf{type}}) & \\[1ex]
clu ::= & \mathsf{addSuperType}(n_{\mathsf{class}}, n_{\mathsf{oClass}}) & \text{(collection link update)} \\
& \mid \mathsf{removeSuperType}(n_{\mathsf{class}}, n_{\mathsf{sClass}}) & \\
& \mid \mathsf{moveSuperTypeTo}(n_{\mathsf{class}}, n_{\mathsf{sClass}}, n_{\mathsf{oClass}}) & \\[1ex]
olu ::= & \mathsf{moveOppositeTo}(n_{\mathsf{class}}.n, n_{\mathsf{oClass}}.n_{\mathsf{ref}}) & \text{(optional link update)} \\
& \mid \mathsf{removeOpposite}(n_{\mathsf{class}}.n) &
\end{array}
$$

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($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}$) 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($n_{\mathsf{class}}.n, n_{\mathsf{type}}$) 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($n_{\mathsf{class}}, n_{\mathsf{oClass}}$) adds n_{oClass} to the set of classes inherited by n_{class}; precondition: n_{class} is not an ancestor of n_{oClass}.
- removeSuperType($n_{\mathsf{class}}, n_{\mathsf{sClass}}$) removes n_{sClass} from the set of classes inherited by n_{class}; no precondition.

– moveSuperTypeTo($n_{\mathsf{class}}, n_{\mathsf{sClass}}, n_{\mathsf{oClass}}$) 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($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}.n_{\mathsf{ref}}$) moves the opposite target of n to the reference n_{ref} of class n_{oClass}; precondition: n_{ref} does not already have an opposite reference.
– removeOpposite($n_{\mathsf{class}}.n$) removes the opposite target of n; no precondition.

µDif operators x				denotation $[\![x]\!]_{op}$
create			createClass(n)	$cc : \mathfrak{M} \times \mathcal{N} \to \mathfrak{M}$
			createDataType(n)	$cdt : \mathfrak{M} \times \mathcal{N} \to \mathfrak{M}$
			createAttribute($n, n_{\mathsf{class}}, n_{\mathsf{type}}$)	$ca : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
			createReference($n, n_{\mathsf{class}}, n_{\mathsf{targetedClass}}$)	$cr : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
delete			deleteClass(n_{class})	$dc : \mathfrak{M} \times \mathcal{N} \to \mathfrak{M}$
			deleteDataType(n_{type})	$ddt : \mathfrak{M} \times \mathcal{N} \to \mathfrak{M}$
			deleteAttribute($n_{\mathsf{attrib}}, n_{\mathsf{class}}$)	$da : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$
			deleteReference($n_{\mathsf{ref}}, n_{\mathsf{class}}$)	$dr : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$
update	value		setName(n, n_{new})	$sn_c : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$
			setName($n_{\mathsf{class}}.n, n_{\mathsf{new}}$)	$sn_f : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
			setLowerBound($n_{\mathsf{class}}.n, i$)	$slb : \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{N} \to \mathfrak{M}$
			setUpperBound($n_{\mathsf{class}}.n, i$)	$sub : \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{N} \to \mathfrak{M}$
			setAbstract(n_{class}, b)	$sa : \mathfrak{M} \times \mathcal{N} \times \mathbb{B} \to \mathfrak{M}$
			setContainer($n_{\mathsf{class}}.n, b$)	$sc : \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{B} \to \mathfrak{M}$
	containment		moveFeatureTo($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}$)	$mft : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
	link	(1,1)	moveReferenceTargetTo($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}$)	$mrtt : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
			moveAttributeTypeTo($n_{\mathsf{class}}.n, n_{\mathsf{type}}$)	$matt : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
		(0,∞)	addSuperType($n_{\mathsf{class}}, n_{\mathsf{oClass}}$)	$asc : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$
			removeSuperType($n_{\mathsf{class}}, n_{\mathsf{sClass}}$)	$rsc : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$
			moveSuperTypeTo($n_{\mathsf{class}}, n_{\mathsf{sClass}}, n_{\mathsf{oClass}}$)	$msct : \mathfrak{M} \times \mathcal{N}^3 \to \mathfrak{M}$
		(0,1)	moveOppositeTo($n_{\mathsf{class}}.n, n_{\mathsf{oClass}}.n_{\mathsf{ref}}$)	$mot : \mathfrak{M} \times \mathcal{N}^4 \to \mathfrak{M}$
			removeOpposite($n_{\mathsf{class}}.n$)	$ro : \mathfrak{M} \times \mathcal{N}^2 \to \mathfrak{M}$

Fig. 19. Valuation of µDif operators

2.5 µDif semantics

The denotational semantics of µDif is based upon the semantics domain of µEcore noted $\mathfrak{M}$ and introduced in figure 8.

We note $\mathcal{L}(op)$ the sets of words yielded from op in figure 12. Basically, $\mathcal{L}(op)$ contains 22 refactoring operators.

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

Informally said, a μDif operator is described by a function from metamodels (plus specific parameters) to metamodels. Figure 19 gathers the μDif 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 $\mathfrak{m} \in \mathfrak{M}$. It corresponds to the *input* metamodel. By definition, $\mathfrak{m}$ corresponds to the following pair:

$$\mathfrak{m} \triangleq \big((\mathcal{C}, \mathcal{C}_{\mathcal{A}}, \mathcal{D}, \mathcal{R}, \mathcal{R}_{\mathcal{C}}, \mathcal{A}), (\delta_I, \delta_A, \delta_R, \delta_{opp}) \big)$$

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

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

$$\mathfrak{m}.(x \cup y) \triangleq \mathfrak{m}.x \cup \mathfrak{m}.y$$

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

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

- $\forall (a, b) \in \mathcal{N}^2, \mathfrak{m}[a/b] \triangleq \mathfrak{m}$ where each occurrence of b has been replaced by a (regardless the kind of elements that are named b).
- $\forall (a, b, n_1, n_2) \in \mathcal{N}^4, \mathfrak{m}[(a, n_1)/(b, n_2)] \triangleq \mathfrak{m}$ 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 $\mathfrak{m}$).
- $\forall (a, b, a', b') \in \mathcal{N}^4, \mathfrak{m}[(a' \mapsto b')/(a \mapsto b)] \triangleq \mathfrak{m}$ 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 $\mathfrak{m}$).

Classifier substitution Knowing $(a, b) \in \mathcal{N}^2$, we note $\mathfrak{m}[a/b]_c$ the *metamodel* $\mathfrak{m}$ 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 $\mathfrak{m}$). This notation is a restriction of the previous one. It allows to specifically target classifiers. Example:

– $\forall(a, b) \in \mathcal{N}^2, \mathfrak{m}[a/b]_c \triangleq \mathfrak{m}$ where each occurrence of b has been replaced by a. *Features* named b 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*:

$$\begin{aligned} \Delta_I \quad : \quad & \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C}) \\ & c \mapsto \{c' \in \mathcal{C} \mid \delta_I(c) = c'\} \end{aligned}$$

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:

$$\begin{aligned} \alpha_I \quad : \quad & \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C}) \\ & c \mapsto \bigcup_{c' \in \Delta_I(c)} \left(\{c'\} \cup \alpha_I(c')\right) \end{aligned}$$

We note $\alpha'_I(c)$ the set of all ancestor 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.

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

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

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

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.

$$
\begin{aligned}
ca \quad : \quad & \mathfrak{M} \times \mathcal{N}^3 \quad \to \mathfrak{M} \\
& (\mathfrak{m}, n_a, n_c, n_d) \mapsto \\
& \begin{cases}
n_c \notin \mathfrak{m}.\mathcal{C} \vee n_d \notin \mathfrak{m}.\mathcal{D} \vee \exists n \in \alpha'_I(n_c), (n, n_a) \in \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) & : \quad \mathfrak{m} \\
n_c \in \mathfrak{m}.\mathcal{C} \wedge n_d \in \mathfrak{m}.\mathcal{D} \wedge \forall n \in \alpha'_I(n_c), (n, n_a) \notin \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) & : \\
\qquad \mathfrak{m}.[\mathcal{A} = \mathcal{A} \cup \{(n_c, n_a)\}] & \\
\qquad \quad .[\delta_{\mathcal{A}} = \delta_{\mathcal{A}} \cup \{(n_c, n_a) \mapsto ((0,1), n_d)\}] &
\end{cases}
\end{aligned}
$$

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.

$$
\begin{aligned}
cr \quad : \quad & \mathfrak{M} \times \mathcal{N}^3 \quad \to \mathfrak{M} \\
& (\mathfrak{m}, n_r, n_c, n'_c) \mapsto \\
& \begin{cases}
n_c \notin \mathfrak{m}.\mathcal{C} \vee n'_c \notin \mathfrak{m}.\mathcal{C} \vee \exists n \in \alpha'_I(n_c), (n, n_r) \in \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) & : \quad \mathfrak{m} \\
n_c \in \mathfrak{m}.\mathcal{C} \wedge n'_c \in \mathfrak{m}.\mathcal{C} \wedge \forall n \in \alpha'_I(n_c), (n, n_r) \notin \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) & : \\
\qquad \mathfrak{m}.[\mathcal{R} = \mathcal{R} \cup \{(n_c, n_r)\}] & \\
\qquad \quad .[\delta_{\mathcal{R}} = \delta_{\mathcal{R}} \cup \{(n_c, n_r) \mapsto ((0,1), n'_c)\}] &
\end{cases}
\end{aligned}
$$

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.

$$
\begin{aligned}
dc \quad & : \mathfrak{M} \times \mathcal{N} \to \mathfrak{M} \\
(\mathfrak{m}, n) \quad & \mapsto \begin{cases}
\exists n' \in \mathcal{C}, \delta_I(n') = n \vee \exists (r, m) \in \mathcal{R} \times \mathcal{M}, \delta_R(r) = (m, n) : \mathfrak{m} \\
\forall n' \in \mathcal{C}, \delta_I(n') \neq n \wedge \forall (r, m) \in \mathcal{R} \times \mathcal{M}, \delta_R(r) \neq (m, n) : \\
\qquad \mathfrak{m}.[\mathcal{C} = \mathcal{C} \backslash \{n\}] \\
\qquad \quad .[\mathcal{C}_{\mathcal{A}} = \mathcal{C}_{\mathcal{A}} \backslash \{n\}] \\
\qquad \quad .[\mathcal{R} = \mathcal{R} \backslash \{(n, n_r) \mid n_r \in \mathcal{N}\}] \\
\qquad \quad .[\mathcal{R}_{\mathcal{C}} = \mathcal{R}_{\mathcal{C}} \backslash \{(n, n_r) \mid n_r \in \mathcal{N}\}] \\
\qquad \quad .[\mathcal{A} = \mathcal{A} \backslash \{(n, n_a) \mid n_a \in \mathcal{N}\}] \\
\qquad \quad .[\delta_I = \delta_I \backslash \{n \mapsto n' \mid n' \in \mathcal{C}\}] \\
\qquad \quad .[\delta_A = \delta_A \backslash \{(n, n_a) \mapsto a \mid (n_a, a) \in \mathcal{N} \times (\mathcal{M} \times \mathcal{D})\}] \\
\qquad \quad .[\delta_R = \delta_R \backslash \{(n, n_r) \mapsto r \mid (n_r, r) \in \mathcal{N} \times (\mathcal{M} \times \mathcal{C})\}] \\
\qquad \quad .[\delta_{opp} = \delta_{opp} \backslash \{(n, n_r) \mapsto x \mid (n_r, x) \in \mathcal{N} \times \mathcal{R}\}] \\
\qquad \quad .[\delta_{opp} = \delta_{opp} \backslash \{x \mapsto (n, n_r) \mid (n_r, x) \in \mathcal{N} \times \mathcal{R}\}]
\end{cases}
\end{aligned}
$$

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

$$
\begin{aligned}
ddt \quad : \quad & \mathfrak{M} \times \mathcal{N} \to \mathfrak{M} \\
& (\mathfrak{m}, n) \mapsto \begin{cases}
\exists (a, m) \in \mathcal{A} \times \mathcal{M}, \delta_A(a) = (m, n) & : \quad \mathfrak{m} \\
\forall (a, m) \in \mathcal{A} \times \mathcal{M}, \delta_A(a) \neq (m, n) & : \quad \mathfrak{m}.[\mathcal{D} = \mathcal{D} \backslash \{n\}]
\end{cases}
\end{aligned}
$$

Delete attribute Deletion of an existing attribute: the attribute and its corresponding structural link are removed.

$$\begin{array}{rcl} da & : & \mathfrak{M} \times \mathcal{N}^2 \;\to \mathfrak{M} \\ & & (\mathfrak{m}, n_a, n_c) \mapsto \begin{cases} \mathfrak{m}.[\mathcal{A} = \mathcal{A} \backslash \{(n_c, n_a)\}] \\ \quad .[\delta_A = \delta_A \backslash \{(n_c, n_a) \mapsto a \mid a \in (\mathcal{M} \times \mathcal{D})\}] \end{cases} \end{array}$$

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

$$\begin{array}{rcl} dr & : & \mathfrak{M} \times \mathcal{N}^2 \;\to \mathfrak{M} \\ & & (\mathfrak{m}, n_r, n_c) \mapsto \begin{cases} \exists r' \in \mathcal{R}, \delta_{opp}(r') = (n_c, n_r) \;\;:\;\; \mathfrak{m} \\ \forall r' \in \mathcal{R}, \delta_{opp}(r') \neq (n_c, n_r) \;\;: \\ \qquad \mathfrak{m}.[\mathcal{R} = \mathcal{R} \backslash \{(n_c, n_r)\}] \\ \qquad \quad .[\mathcal{R}_{\mathcal{C}} = \mathcal{R}_{\mathcal{C}} \backslash \{(n_c, n_r)\}] \\ \qquad \quad .[\delta_R = \delta_R \backslash \{(n_c, n_r) \mapsto r \mid r \in (\mathcal{M} \times \mathcal{C})\}] \end{cases} \end{array}$$

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.

$$\begin{array}{rcl} sn_c & : & \mathfrak{M} \times \mathcal{N}^2 \;\to \mathfrak{M} \\ & & (\mathfrak{m}, n_c, n'_c) \mapsto \begin{cases} n_c \notin \mathfrak{m}.(\mathcal{C} \cup \mathcal{D}) \vee n'_c \in \mathfrak{m}.(\mathcal{C} \cup \mathcal{D}) \;\;:\;\; \mathfrak{m} \\ n_c \in \mathfrak{m}.(\mathcal{C} \cup \mathcal{D}) \wedge n'_c \notin \mathfrak{m}.(\mathcal{C} \cup \mathcal{D}) \;\;:\;\; \mathfrak{m}[n'_c/n_c]_c \end{cases} \end{array}$$

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.

$$\begin{array}{l} sn_f : \quad \mathfrak{M} \times \mathcal{N}^3 \quad \to \mathfrak{M} \\ \quad (\mathfrak{m}, n_c, n_f, n'_f) \mapsto \\ \qquad \begin{cases} (n_c, n_f) \notin \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \vee \exists n \in \alpha'_I(n_c), (n, n'_f) \in \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \;\;:\;\; \mathfrak{m} \\ (n_c, n_f) \in \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \wedge \forall n \in \alpha'_I(n_c), (n, n'_f) \notin \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \;\;: \\ \hfill \mathfrak{m}[(n_c, n'_f)/(n_c, n_f)] \end{cases} \end{array}$$

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.

$$\begin{array}{l} slb \;\;:\;\; \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{N} \to \mathfrak{M} \\ \qquad (\mathfrak{m}, n_c, n_f, i) \;\mapsto \\ \qquad \quad \begin{cases} (n_c, n_f) \notin \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \mathfrak{m} \\ (n_c, n_f) \in \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \\ \quad \text{let } \delta_{\mathcal{F}} = \mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}}) \text{ and } ((x, y), n) = \delta_{\mathcal{F}}(n_c, n_f) \text{ in:} \\ \qquad \begin{cases} y < i : \mathfrak{m} \\ y \geq i : \mathfrak{m}[(n_c, n_f) \mapsto ((i, y), n) \;/\; (n_c, n_f) \mapsto ((x, y), n)] \end{cases} \end{cases} \end{array}$$

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.

$$
\begin{array}{l}
sub \quad : \quad \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{N}^* \cup \{\infty\} \to \mathfrak{M} \\
\qquad\qquad (\mathfrak{m}, n_c, n_f, i) \qquad \mapsto \\
\qquad\qquad \begin{cases}
(n_c, n_f) \notin \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \mathfrak{m} \\
(n_c, n_f) \in \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \\
\quad \text{let } \delta_{\mathcal{F}} = \mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}}) \text{ and } ((x, y), n) = \delta_{\mathcal{F}}(n_c, n_f) \text{ in:} \\
\qquad \begin{cases} x > i : \mathfrak{m} \\ x \leq i : \mathfrak{m}[(n_c, n_f) \mapsto ((x, i), n) \ / \ (n_c, n_f) \mapsto ((x, y), n)] \end{cases}
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{l}
sa \quad : \quad \mathfrak{M} \times \mathcal{N} \times \mathbb{B} \to \mathfrak{M} \\
\qquad (\mathfrak{m}, n_c, b) \quad \mapsto \begin{cases}
n_c \notin \mathfrak{m}.\mathcal{C} & : \quad \mathfrak{m} \\
n_c \in \mathcal{C} \wedge b & : \quad \mathfrak{m}.[\mathcal{C}_{\mathcal{A}} = \mathcal{C}_{\mathcal{A}} \cup \{n_c\}] \\
n_c \in \mathcal{C} \wedge \neg b & : \quad \mathfrak{m}.[\mathcal{C}_{\mathcal{A}} = \mathcal{C}_{\mathcal{A}} \backslash \{n_c\}]
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{l}
sc : \mathfrak{M} \times \mathcal{N}^2 \times \mathbb{B} \to \mathfrak{M} \\
\quad (\mathfrak{m}, n_c, n_r, b) \mapsto \begin{cases}
(n_c, n_r) \notin \mathfrak{m}.\mathcal{R} & : \mathfrak{m} \\
(n_c, n_r) \in \mathfrak{m}.\mathcal{R} \wedge b \wedge \exists r \in \mathfrak{m}.\mathcal{R}_{\mathcal{C}}, \delta_{opp}(r) = (n_c, n_r) & : \mathfrak{m} \\
(n_c, n_r) \in \mathfrak{m}.\mathcal{R} \wedge b \wedge \forall r \in \mathfrak{m}.\mathcal{R}_{\mathcal{C}}, \delta_{opp}(r) \neq (n_c, n_r) & : \\
\qquad \mathfrak{m}.[\mathcal{R}_{\mathcal{C}} = \mathcal{R}_{\mathcal{C}} \cup \{(n_c, n_r)\}] & \\
(n_c, n_r) \in \mathfrak{m}.\mathcal{R} \wedge \neg b \quad : \quad \mathfrak{m}.[\mathcal{R}_{\mathcal{C}} = \mathcal{R}_{\mathcal{C}} \backslash \{(n_c, n_r)\}] &
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{l}
mft : \quad \mathfrak{M} \times \mathcal{N}^3 \quad \to \mathfrak{M} \\
\quad (\mathfrak{m}, n_c, n_f, n'_c) \mapsto \\
\quad \begin{cases}
(n_c, n_f) \notin \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \vee n'_c \notin \mathfrak{m}.\mathcal{C} \vee \exists n \in \alpha'_I(n'_c), (n, n_f) \in \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \mathfrak{m} \\
(n_c, n_f) \in \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) \wedge n'_c \in \mathfrak{m}.\mathcal{C} \wedge \forall n \in \alpha'_I(n'_c), (n, n_f) \notin \mathfrak{m}.(\mathcal{A} \cup \mathcal{R}) : \\
\qquad \mathfrak{m}[(n'_c, n_f)/(n_c, n_f)]
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{lll}
mrtt & : & \mathfrak{M} \times \mathcal{N}^3 \quad \rightarrow \mathfrak{M} \\
& & (\mathfrak{m}, n_c, n_r, n'_c) \mapsto \\
& & \quad \begin{cases}
(n_c, n_r) \notin \mathfrak{m}.\mathcal{R} \vee n'_c \notin \mathfrak{m}.\mathcal{C} \quad : \quad \mathfrak{m} \\
(n_c, n_r) \in \mathfrak{m}.\mathcal{R} \wedge n'_c \in \mathfrak{m}.\mathcal{C} \quad : \\
\quad \text{let } (m, n) = \mathfrak{m}.\delta_{\mathcal{R}}(n_c, n_r) \text{ in:} \\
\quad \begin{cases}
\exists n_{ro} \in \mathcal{N}, \delta_{opp}(n, n_{ro}) = (n_c, n_r) \quad : \\
\quad \begin{cases}
\exists n' \in \alpha'_I(n'_c), (n', n_{ro}) \in \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) : \mathfrak{m} \\
\forall n' \in \alpha'_I(n'_c), (n', n_{ro}) \notin \mathfrak{m}.(\mathcal{R} \cup \mathcal{A}) : \\
\quad \mathfrak{m}[(n'_c, n_{ro}) \ / \ (n, n_{ro})] \\
\quad\quad [(n_c, n_r) \mapsto (m, n'_c) \ / \ (n_c, n_r) \mapsto (m, n)]
\end{cases} \\
\forall n_{ro} \in \mathcal{N}, ((n, n_{ro}) \mapsto (n_c, n_r)) \notin \delta_{opp} \quad : \\
\quad \mathfrak{m}[(n_c, n_r) \mapsto (m, n'_c) \ / \ (n_c, n_r) \mapsto (m, n)]
\end{cases}
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{lll}
matt & : & \mathfrak{M} \times \mathcal{N}^3 \quad \rightarrow \mathfrak{M} \\
& & (\mathfrak{m}, n_c, n_a, n_d) \mapsto \\
& & \quad \begin{cases}
(n_c, n_a) \notin \mathfrak{m}.\mathcal{A} \vee n_d \notin \mathfrak{m}.\mathcal{D} \quad : \quad \mathfrak{m} \\
(n_c, n_a) \in \mathfrak{m}.\mathcal{A} \wedge n_d \in \mathfrak{m}.\mathcal{D} \quad : \\
\quad \text{let } (m, n) = \mathfrak{m}.\delta_{\mathcal{A}}(n_c, n_a) \text{ in:} \\
\quad\quad \mathfrak{m}[(n_c, n_a) \mapsto (m, n_d) \ / \ (n_c, n_a) \mapsto (m, n)]
\end{cases}
\end{array}
$$

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.

$$
\begin{array}{lll}
asc & : & \mathfrak{M} \times \mathcal{N}^2 \ \rightarrow \mathfrak{M} \\
& & (\mathfrak{m}, n_c, n'_c) \mapsto \begin{cases}
n_c \notin \mathfrak{m}.\mathcal{C} \vee n'_c \notin \mathfrak{m}.\mathcal{C} \vee n_c \in \alpha'_I(n'_c) \quad : \quad \mathfrak{m} \\
n_c \in \mathfrak{m}.\mathcal{C} \wedge n'_c \in \mathfrak{m}.\mathcal{C} \wedge n_c \notin \alpha'_I(n'_c) \quad : \\
\qquad \mathfrak{m}.[\delta_I = \delta_I \cup \{n_c \mapsto n'_c\}]
\end{cases}
\end{array}
$$

Remove super class Removing an existing super class from a class: the corresponding inheritance link is removed from the set of super classes.

$$
\begin{array}{lll}
rsc & : & \mathfrak{M} \times \mathcal{N}^2 \ \rightarrow \mathfrak{M} \\
& & (\mathfrak{m}, n_c, n'_c) \mapsto \mathfrak{m}.[\delta_I = \delta_I \backslash \{n_c \mapsto n'_c\}]
\end{array}
$$

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.

$$\begin{array}{rcll} msct & : & \mathfrak{M} \times \mathcal{N}^3 & \rightarrow \mathfrak{M} \\ & & (\mathfrak{m}, n_c, n'_c, n''_c) \mapsto & \begin{cases} n''_c \notin \mathfrak{m}.\mathcal{C} \vee n_c \in \alpha'_I(n''_c) & : \quad \mathfrak{m} \\ n''_c \in \mathfrak{m}.\mathcal{C} \wedge n_c \notin \alpha'_I(n''_c) & : \\ \qquad \mathfrak{m}[n_c \mapsto n''_c \ / \ n_c \mapsto n'_c] & \end{cases} \end{array}$$

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.

$$\begin{array}{l} mot : \quad \mathfrak{M} \times \mathcal{N}^4 \quad \rightarrow \mathfrak{M} \\ \quad (\mathfrak{m}, n_c, n_r, n'_c, n'_r) \mapsto \\ \quad \begin{cases} \exists r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) = (n'_c, n'_r) & : \quad \mathfrak{m} \\ (n_c, n_r) \notin \mathfrak{m}.\mathcal{R} \vee (n'_c, n'_r) \notin \mathfrak{m}.\mathcal{R} & : \quad \mathfrak{m} \\ (n_c, n_r) \in \mathfrak{m}.\mathcal{R} \wedge (n'_c, n'_r) \in \mathfrak{m}.\mathcal{R} & \\ \wedge \forall r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) \neq (n'_c, n'_r) & : \\ \quad \begin{cases} \exists r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) = (n_c, n_r) : \\ \quad \mathfrak{m}.[\delta_{opp} = \delta_{opp} \backslash \{r \mapsto (n_c, n_r); (n_c, n_r) \mapsto r\}] \\ \quad\quad .[\delta_{opp} = \delta_{opp} \cup \{(n'_c, n'_r) \mapsto (n_c, n_r); (n_c, n_r) \mapsto (n'_c, n'_r)\}] \\ \forall r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) \neq (n_c, n_r) : \\ \quad \mathfrak{m}.[\delta_{opp} = \delta_{opp} \cup \{(n'_c, n'_r) \mapsto (n_c, n_r); (n_c, n_r) \mapsto (n'_c, n'_r)\}] \end{cases} & \end{cases} \end{array}$$

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

$$\begin{array}{rcll} ro & : & \mathfrak{M} \times \mathcal{N}^2 & \rightarrow \mathfrak{M} \\ & & (\mathfrak{m}, n_c, n_r) \mapsto & \begin{cases} \forall r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) \neq (n_c, n_r) & : \quad \mathfrak{m} \\ \exists r \in \mathfrak{m}.\mathcal{R}, \delta_{opp}(r) = (n_c, n_r) & : \\ \quad \mathfrak{m}.[\delta_{opp} = \delta_{opp} \backslash \{(n_c, n_r) \mapsto r; r \mapsto (n_c, n_r)\}] & \end{cases} \end{array}$$

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:

$$[\![\mathsf{createReference}(n, n_{\mathsf{class}}, n_{\mathsf{targetedClass}})]\!]_{param} \quad \triangleq \quad (n, n_{\mathsf{class}}, n_{\mathsf{targetedClass}}) \in \mathcal{N}^3$$

$$
\begin{array}{lll}
[\![\cdot]\!]_{spec} & : & \mathcal{L}(spec) \rightarrow \mathfrak{M} \\
s & \mapsto & \begin{cases} s \text{ matches with } (\{\ mm\ \}\ op) & : \\ \qquad [\![op]\!]_{op}\big([\![mm]\!]_{mm}\ ,\ [\![op]\!]_{param}\big) & \\ s \text{ matches with } (s'\ op) \text{ where } s' = (\{\ mm\ \}\ (op')^{+}\ ) & : \\ \qquad [\![op]\!]_{op}\big([\![s']\!]_{spec}\ ,\ [\![op]\!]_{param}\big) & \end{cases}
\end{array}
$$

Fig. 20. Valuation of μDif specifications

We introduce in figure 20 a valuation function noted $[\![\cdot]\!]_{spec}$. It applies to $\mathcal{L}(spec)$. 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 μEcore models

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

3.1 Syntax extension

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

Fig. 21. Textual syntax of μEcore models

Figure 21 presents the textual syntax of μEcore models. It extends the syntax of μEcore metamodels introduced in figure 3. A *model* is defined by a given metamodel *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 *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 µEcore metamodel together with its equivalent textual specification. The µEcore metamodel of this figure is taken from figure 4.

$$
\begin{array}{l}
\mathsf{from}\ \langle\ \mathsf{Int}\ \rangle_{\mathsf{d}}\ \ \langle\ \mathsf{Bool}\ \rangle_{\mathsf{d}} \\
\qquad \langle\ \mathsf{a\ A}\ [\mathsf{i},(0,1),\mathsf{Int}]_{\mathsf{a}}\ [\mathsf{b},(0,\infty),\mathsf{Bool}]_{\mathsf{a}}\ [\mathsf{y},(0,1),\mathsf{Y}\ \leftarrow \mathsf{a}]_{\mathsf{r}}\ \rangle_{\mathsf{c}} \\
\qquad \langle\ \mathsf{X}\ /\ \mathsf{A}\ [\mathsf{j},(1,1),\mathsf{Int}]_{\mathsf{a}}\ \rangle_{\mathsf{c}} \\
\qquad \langle\ \mathsf{Y}\ /\ \mathsf{A}\ [\mathsf{c\ a},(1,2),\mathsf{A}\ \leftarrow \mathsf{y}]_{\mathsf{r}}\ [\mathsf{x},(0,\infty),\mathsf{X}]_{\mathsf{r}}\ \rangle_{\mathsf{c}}\quad : \\
\langle\ \mathsf{iy1} : \mathsf{Y}\ [\mathsf{i} : 3]\ [\mathsf{b} : \mathit{true}]\ [\mathsf{b} : \mathit{false}]\ [\mathsf{a} : \mathsf{iy2}]\ [\mathsf{x} : \mathsf{ix}]\ \rangle \\
\langle\ \mathsf{iy2} : \mathsf{Y}\ [\mathsf{a} : \mathsf{ix}]\ [\mathsf{y} : \mathsf{iy1}]\ \rangle \\
\langle\ \mathsf{ix} : \mathsf{X}\ [\mathsf{i} : 4]\ [\mathsf{j} : 4]\ [\mathsf{y} : \mathsf{iy2}]\ \rangle
\end{array}
$$

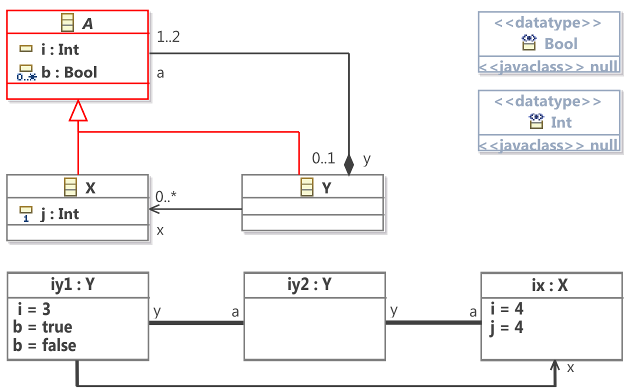

Fig. 22. Textual and graphical form of a µEcore model

3.2 Semantics of a µEcore model

The formal semantics of µEcore 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 µEcore metamodels.

3.3 Semantic domain

Figure 23 defines the *name spaces* of the semantic domain. They are compatible with name spaces $\mathcal{N}$, $\mathcal{C}$ and $\mathcal{D}$ 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

$$
\begin{array}{lcl}
\mathcal{S} & : & \text{scalar values} \\
\mathcal{I} \subseteq \mathcal{N} \setminus (\mathcal{C} \cup \mathcal{D}) & : & \text{instances}
\end{array}
$$

Fig. 23. μEcore 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 δ_v and δ_l 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.

$$
\begin{array}{lll}
\delta_{inst} : \quad \mathcal{I} & \rightharpoonup \mathcal{C} \setminus \mathcal{C}_{\mathcal{A}} & \text{instanciation} \\
\delta_v \quad : \mathcal{I} \times \mathcal{A} & \rightharpoonup \mathcal{P}(\mathcal{S}) & \text{values} \\
\delta_l \quad : \mathcal{I} \times \mathcal{R} & \rightharpoonup \mathcal{P}(\mathcal{I}) & \text{links}
\end{array}
$$

Fig. 24. μEcore model semantics: partial functions

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

$$
\begin{array}{lll}
\mathfrak{I} & \triangleq \mathfrak{N}_i \times \mathfrak{F}_i \times \mathfrak{M} & \text{models} \\
\mathfrak{N}_i & \triangleq \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) & \text{models' name spaces} \\
\mathfrak{F}_i & \triangleq \mathcal{P}(\delta_{inst}) \times \mathcal{P}(\delta_v) \times \mathcal{P}(\delta_l) & \text{models' partial functions}
\end{array}
$$

Fig. 25. μEcore 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 $\mathfrak{i} \in \mathfrak{I}$ corresponds to the following triplet:

$$\mathfrak{i} \triangleq \Big(\; (\mathcal{S}, \mathcal{I}) \; , \; (\delta_{inst}, \delta_v, \delta_l) \; , \; \mathfrak{m} \; \Big)$$

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

3.4 Valuation function

Notation We note $mm(x)$ the metamodel part of a given model specification $x = \boxed{\textsf{from meta} \;:\; (i^+)}$. By definition in this case, $mm(x) \triangleq \textsf{meta}$.

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:

$$\textsf{pattern} \sqsubset mod$$

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 $\mathcal{L}(i)$ the sets of words yielded from i in figure 21, then by definition: $\textsf{pattern} \in \mathcal{L}(i)$.

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

$$\begin{aligned} [\![\cdot]\!]_{mod} \; : \; & \mathcal{L}(mod) \rightarrow \mathfrak{I} \\ & x \mapsto [\![x]\!]_{mod} = \big(\; (\mathcal{S}, \mathcal{I}) \; , \; (\delta_{inst}, \delta_v, \delta_l) \; , \; \mathfrak{m} \; \big) \quad \text{where:} \end{aligned}$$

$$\mathfrak{m} = [\![mm(x)]\!]_{mm}$$

$$\begin{aligned} \mathcal{S} &= \{ s \in \mathcal{S} \mid \exists (n, n', n'') \in \mathcal{N}^3 \langle \; n : n' \; [n'' : s] \; \rangle \sqsubset x \} \\ \mathcal{I} &= \{ n \in \mathcal{N} \mid \exists n' \in \mathcal{N}, \langle \; n : n' \; \rangle \sqsubset x \} \end{aligned}$$

$$\begin{aligned} \delta_{inst} &= \{ (n, n') \in \mathcal{I} \times \mathcal{C} \mid \langle \; n : n' \; \rangle \sqsubset x \} \\ \delta_v &= \{ ((n, (n', n'')), X) \in (\mathcal{I} \times \mathcal{A}) \times \mathcal{P}(\mathcal{S}) \mid \exists s \in X, \langle \; n : n' \; [n'' : s] \; \rangle \sqsubset x \} \\ \delta_l &= \{ ((n, (n', n'')), X) \in (\mathcal{I} \times \mathcal{A}) \times \mathcal{P}(\mathcal{I}) \mid \exists n_r \in X, \langle \; n : n' \; [n'' : n_r] \; \rangle \sqsubset x \} \end{aligned}$$

Fig. 26. µEcore model semantics: valuation

$$[\![x]\!]_{mod} = \mathfrak{i} = (\mathfrak{n}_i, \mathfrak{f}_i, \mathfrak{m}) = \big(\;\; (\mathcal{S}, \mathcal{I}) \; , \; (\delta_{inst}, \delta_v, \delta_l) \; , \; \mathfrak{m} \; \big) \quad \text{where:}$$

$$\mathfrak{m} = [\![mm(x)]\!]_{mm} \qquad \text{(see figure 10 for details)}$$

$$\begin{aligned} \mathcal{S} &= \{3, 4, true, false\} \\ \mathcal{I} &= \{iy_1, iy_2, ix\} \end{aligned}$$

$$\begin{aligned} \delta_{inst} &= \{ iy_1 \mapsto Y, iy_2 \mapsto Y, ix \mapsto X \} \\ \delta_v &= \{ (iy_1, (A, i)) \mapsto \{3\}, (iy_1, (A, b)) \mapsto \{true, false\}, \\ & \quad\; (ix, (A, i)) \mapsto \{4\}, (ix, (X, j)) \mapsto \{4\} \} \\ \delta_l &= \{ (iy_1, (Y, a)) \mapsto \{iy_2\}, (iy_1, (Y, x)) \mapsto \{ix\}, \\ & \quad\; (iy_2, (Y, a)) \mapsto \{ix\}, (iy_2, (A, y)) \mapsto \{iy_1\}, (ix, (A, y)) \mapsto \{iy_2\} \} \end{aligned}$$

Fig. 27. Semantics of a µEcore model

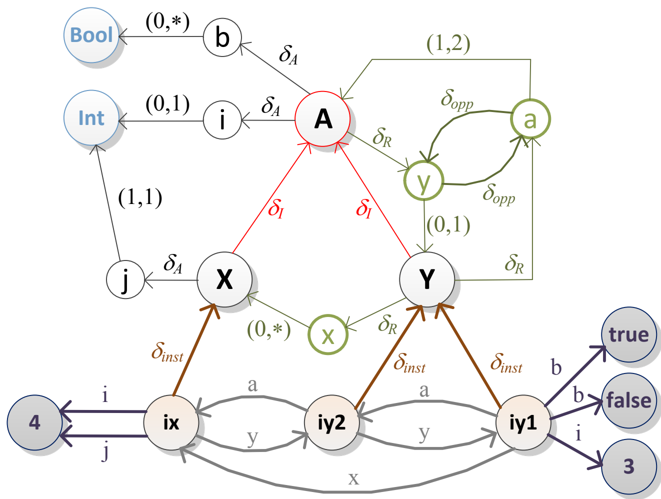

Fig. 28. Graph view of the semantics of a μEcore model

3.5 Example

Let x be the model of figure 22. Its denotation is given by $[\![x]\!]_{mod} = \mathfrak{i} = (\mathfrak{n}_i, \mathfrak{f}_i, \mathfrak{m})$.

Figure 27 shows $\mathfrak{i}$ 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 δ_{inst}, δ_v and δ_l. The partial function δ_{inst} is represented by edges between instances and concrete classes. The partial function δ_v is represented by edges between instances and scalar values. They are labeled by the names of the corresponding attributes. The partial function δ_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 δ_{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 $\mathfrak{i}$ is noted $\mathcal{V}(\mathfrak{i})$ and is formally defined by figure 29.

$$
\begin{aligned}
\forall \mathfrak{i} \in \mathfrak{I}, \mathcal{V}(\mathfrak{i}) \quad \triangleq \quad & \\
& \forall i \in \mathfrak{i}.\mathcal{I}, \forall (n_c, n_a) \in \mathcal{N}^2, \\
& \mathfrak{i}.\delta_v(i, (n_c, n_a)) \neq \emptyset \implies \\
& n_c \in \alpha'_I(\mathfrak{i}.\delta_{inst}(i)) \land \exists (m, d) \in \mathcal{M} \times \mathcal{D}, \mathfrak{i}.\mathfrak{m}.\delta_A(n_c, n_a) = (m, d) \\
\bigwedge \quad & \\
& \forall i \in \mathfrak{i}.\mathcal{I}, \forall (n_c, n_r) \in \mathcal{N}^2, \\
& \mathfrak{i}.\delta_l(i, (n_c, n_r)) \neq \emptyset \implies \\
& n_c \in \alpha'_I(\mathfrak{i}.\delta_{inst}(i)) \land \exists (m, c') \in \mathcal{M} \times \mathcal{C}, \mathfrak{i}.\mathfrak{m}.\delta_R(n_c, n_r) = (m, c') \\
& \quad \land \ \forall i' \in \mathfrak{i}.\delta_l(i, (n_c, n_r)), c' \in \alpha'_I(\mathfrak{i}.\delta_{inst}(i')) \\
\bigwedge \quad & \\
& \forall i \in \mathfrak{i}.\mathcal{I}, \forall (n_c, n) \in \mathcal{N}^2, \\
& \text{let } card = |\mathfrak{i}.(\delta_l \cup \delta_v)(i, (n_c, n))| \text{ in:} \\
& \text{let } (min, max) = \mathfrak{i}.\mathfrak{m}.(\delta_R \cup \delta_A)(n_c, n).m \text{ in:} \\
& card \leq max \ \land \ card \geq min \\
\bigwedge \quad & \\
& \forall (i, i', n_c, n_r) \in \mathfrak{i}.\mathcal{I}^2 \times \mathcal{N}^2, i' \in \mathfrak{i}.\delta_l(i, (n_c, n_r)), \\
& \exists (n'_c, n'_r) \in \mathcal{N}^2, \mathfrak{i}.\mathfrak{m}.\delta_{opp}(n_c, n_r) = (n'_c, n'_r) \implies \\
& i \in \mathfrak{i}.\delta_l(i', (n'_c, n'_r))
\end{aligned}
$$

Fig. 29. Conformance property of µEcore models

4 Model co-evolution with µDif

So far, µDif is defined as metamodel refactoring language whose scope encompass the basic metamodel constructs of µEcore. Now, we aim at extending this scope to *model migration*. For that purpose, each μDif 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 μEcore, which itself has been extended to *models* (see figure 21). A *migration* is a context (defined by a model *mod*) followed by a non-empty ordered sequence of *operators*.

$$mig ::= \{\ mod\ \}\ (op)^{+} \quad \text{(migration)}$$

Fig. 30. Textual syntax of μDif 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 $\mathfrak{i} \in \mathfrak{I}$ corresponding to the following triplet:

$$\mathfrak{i} \triangleq \Big(\ (\mathcal{S},\mathcal{I})\ ,\ (\delta_{inst},\delta_v,\delta_l)\ ,\ \mathfrak{m}\ \Big)$$

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

$$\begin{aligned} &\Big(\ (\mathcal{S},\mathcal{I})\ ,\ (\delta_{inst},\delta_v,\delta_l)\ ,\ \mathfrak{m}\ \Big) \cup \Big(\ (\mathcal{S}',\mathcal{I}')\ ,\ (\delta_{inst'},\delta_{v'},\delta_{l'})\ ,\ \mathfrak{m}\ \Big) \\ \triangleq &\Big(\ (\mathcal{S}\cup\mathcal{S}',\mathcal{I}\cup\mathcal{I}')\ ,\ (\delta_{inst}\cup\delta_{inst'},\delta_v\cup\delta_{v'},\delta_l\cup\delta_{l'})\ ,\ \mathfrak{m}\ \Big) \end{aligned}$$

Sub-model We note $\mathfrak{i} \subseteq \mathfrak{i}'$ the *proposition* stating that each *model* component of $\mathfrak{i}$ is contained by the corresponding component of $\mathfrak{i}'$, provided $\mathfrak{i}$ and $\mathfrak{i}'$ have *the same metamodel*.

Substitution of model component We note $\mathfrak{i}.[x = y]$ the *model* $\mathfrak{i}$ where y has been *substituted* to the x component of $\mathfrak{m}$ (*e.g.* $\mathfrak{i}.[\mathfrak{m} = ...]$).

Empty model We note $\mathfrak{i}^{\emptyset}$ the *model* which *metamodel* corresponds to $\mathfrak{i}.\mathfrak{m}$, and whose *model components* are empty:

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

Note that the metamodel part actually remains unchanged:

$$\forall \mathfrak{i} \in \mathfrak{I},\ \mathfrak{i}^{\emptyset}.\mathfrak{m} = \mathfrak{i}.\mathfrak{m}$$

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 µDif 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 µDif operator taken from table 19. For instance, *op* can refer to $cc : \mathfrak{M} \times \mathcal{N} \rightarrow \mathfrak{M}$, the denotation of 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:

$$\text{let } \mathcal{P} \text{ be the parameter domain of } op, \quad \begin{array}{rcl} op & : & \mathfrak{M} \times \mathcal{P} \rightarrow \mathfrak{M} \\ & & (\mathfrak{m}, p) \ \mapsto \mathfrak{m}_{refactored} \end{array}$$

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 µDif operator and $\mathcal{P}$ its parameter domain. We note $\overrightarrow{op}^r$ 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):

$$\begin{array}{rcl} \overrightarrow{op}^r & : & \mathfrak{I} \times \mathcal{P} \rightarrow \mathfrak{I} \\ & & (\mathfrak{i}, p) \ \mapsto \mathfrak{i}' \ \text{ such that } \mathfrak{i}' \subseteq \mathfrak{i} \end{array}$$

This definition implies that discarded elements are actually taken from the model which is intended to be migrated.

Added elements Let *op* be the functional denotation of a given μDif operator and $\mathcal{P}$ its parameter domain. We note $\overrightarrow{op}^a$ the function mapping a *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 *refactored* input metamodel (*i.e.* the metamodel after its refactoring):

$$\begin{aligned} \overrightarrow{op}^a \quad : \quad & \mathfrak{I} \times \mathcal{P} \rightarrow \mathfrak{I} \\ & (\mathfrak{i}, p) \;\mapsto \mathfrak{i}' \;\text{ such that } \mathfrak{i}'.\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p) \wedge \mathfrak{i}' \setminus \big(\; \mathfrak{i}.[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)] \;\big) = \mathfrak{i}' \end{aligned}$$

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 *op* be the functional denotation of a given μDif operator and $\mathcal{P}$ its parameter domain. We note $\overrightarrow{op}$ the function mapping a *model* to the corresponding *migrated model* according to the semantics of *op*.

$$\begin{aligned} \overrightarrow{op} \quad : \quad & \mathfrak{I} \times \mathcal{P} \rightarrow \mathfrak{I} \\ & (\mathfrak{i}, p) \;\mapsto \big((\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)).[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]\big) \cup \overrightarrow{op}^a(\mathfrak{i}, p) \end{aligned}$$

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 *op* of table 19, the migration is formally defined by the explicit functions $\overrightarrow{op}^r$ and $\overrightarrow{op}^a$. 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 $\overrightarrow{op}$ into a valid output model.

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

$$\overrightarrow{C_{op}} \quad : \quad \mathfrak{I} \times \mathcal{P} \rightarrow \mathbb{B}$$

$$\forall (\mathfrak{i}, p) \in \mathfrak{I} \times \mathcal{P}, \quad \big(\mathcal{V}(\mathfrak{i}) \wedge \overrightarrow{C_{op}}(\mathfrak{i}, p)\big) \implies \mathcal{V}(\overrightarrow{op}(\mathfrak{i}, p))$$

If there is no specific condition, then by definition, $\forall (\mathfrak{i}, p) \in \mathfrak{I} \times \mathcal{P}, \overrightarrow{C_{op}}(\mathfrak{i}, p) =$ *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 μDif 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 μDif operator and $\mathcal{P}$ its parameter domain. We note $\overleftarrow{op}^r$ 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):

$$\begin{aligned} \overleftarrow{op}^r \quad : \quad & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} \to \mathfrak{I} \\ & (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \;\mapsto \mathfrak{i}' \text{ such that } \mathfrak{i}' \subseteq \big((\mathfrak{i} \setminus \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)) \cup \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p)\big) \\ & \qquad\qquad\qquad\qquad \wedge\ (\mathfrak{i} = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)) \implies (\mathfrak{i}' = \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p)) \end{aligned}$$

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 ($\mathfrak{i}' \subseteq (\mathfrak{i} \setminus \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p))$). The second specific subset is a *part* of the elements that have been *added* by the migration ($\mathfrak{i}' \subseteq \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p)$).

In the case of a non-modified migrated model ($\mathfrak{i} = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)$), by definition, the set of discarded elements matches with the set of elements that have been *added* by the migration ($\mathfrak{i}' = \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p)$).

Added elements Let *op* be the functional denotation of a given μDif operator and $\mathcal{P}$ its parameter domain. We note $\overleftarrow{op}^a$ 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):

$$\begin{aligned} \overleftarrow{op}^a \quad : \quad & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} \to \mathfrak{I} \\ & (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \;\mapsto \mathfrak{i}' \text{ such that } \mathfrak{i}'.\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m} \wedge \mathfrak{i}' \setminus \big(\ \mathfrak{i}.[\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}]\ \big) = \mathfrak{i}' \\ & \qquad\qquad\qquad\qquad \wedge\ (\mathfrak{i} = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)) \implies (\mathfrak{i}' = \overrightarrow{op}^r(\mathfrak{i}_{\mathsf{ini}}, p)) \end{aligned}$$

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 ($\mathfrak{i} = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)$), 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 ($\mathfrak{i}' = \overrightarrow{op}^r(\mathfrak{i}_{\mathsf{ini}}, p)$).

Transformation Let *op* be the functional denotation of a given µDif operator and $\mathcal{P}$ its parameter domain. We note $\overleftarrow{op}$ the function mapping a *migrated model* to the corresponding *initial model* according to the semantics of *op*.

$$\begin{array}{lcl} \overleftarrow{op} & : & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} \to \mathfrak{I} \\ & & (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \ \mapsto \big((\mathfrak{i} \setminus \overleftarrow{op}^r(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}})).[\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}]\big) \cup \overleftarrow{op}^a(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \end{array}$$

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 $\overrightarrow{op}^a$, $\overrightarrow{op}^r$, $\overleftarrow{op}^a$ and $\overleftarrow{op}^r$. 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 $\overleftarrow{op}$ back into a valid initial model.

Let *op* be the functional denotation of a given µDif operator and $\mathcal{P}$ its parameter domain. We note $\overleftarrow{C_{op}}$ the *predicate* giving the sufficient condition under which a valid *possibly modified* migrated model coming from a given valid *migrated* model is transformed by $\overleftarrow{op}$ back into a valid initial model:

$$\overleftarrow{C_{op}} \quad : \quad \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} \to \mathbb{B}$$

$$\forall (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \in \mathfrak{I} \times \mathcal{P} \times \mathfrak{I}, \quad \big(\mathcal{V}(\mathfrak{i}) \wedge \mathcal{V}(\mathfrak{i}_{\mathsf{ini}}) \wedge \overleftarrow{C_{op}}(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}})\big) \implies \mathcal{V}(\overleftarrow{op}(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}))$$

As for migration, if there is no specific condition, then by definition, $\forall (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \in \mathfrak{I} \times \mathcal{P} \times \mathfrak{I}, \overrightarrow{C_{op}}(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) = \mathit{true}$, 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* µDif *operator and* $\mathcal{P}$ *its parameter domain.*

$$\forall \mathfrak{i} \in \mathfrak{I}, \forall p \in \mathcal{P}, \quad \overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \mathfrak{i}$$

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

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \big((\overrightarrow{op}(\mathfrak{i}, p) \setminus \overleftarrow{op}^r(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i})).[\mathfrak{m} = \mathfrak{i}.\mathfrak{m}]\big) \cup \overleftarrow{op}^a(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i})$$

However, by definition of $\overleftarrow{op}^r$*, knowing* x *is the first parameter, since we directly have* $x = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)$ *(because the* x *corresponds here to* $\overrightarrow{op}(\mathfrak{i}, p)$ *and* $\mathfrak{i}_{\mathsf{ini}}$ *corresponds to* $\mathfrak{i}$*), then we also have* $\overleftarrow{op}^r(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \overrightarrow{op}^a(\mathfrak{i}, p)$*. Thus:*

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \big((\overrightarrow{op}(\mathfrak{i}, p) \setminus \overrightarrow{op}^a(\mathfrak{i}, p)).[\mathfrak{m} = \mathfrak{i}.\mathfrak{m}]\big) \cup \overleftarrow{op}^a(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i})$$

By definition of $\overleftarrow{op}^a$, knowing x is the first parameter, since we directly have $x = \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)$ (because the x corresponds here to $\overrightarrow{op}(\mathfrak{i}, p)$ and $\mathfrak{i}_{\mathsf{ini}}$ corresponds to $\mathfrak{i}$), then we also have $\overleftarrow{op}^a(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \overrightarrow{op}^r(\mathfrak{i}, p)$. Thus:

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \big((\overrightarrow{op}(\mathfrak{i}, p) \setminus \overrightarrow{op}^a(\mathfrak{i}, p)).[\mathfrak{m} = \mathfrak{i}.\mathfrak{m}]\big) \cup \overrightarrow{op}^r(\mathfrak{i}, p)$$

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

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \begin{pmatrix} \big(\ ((\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)).[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]) \cup \overrightarrow{op}^a(\mathfrak{i}, p) \setminus \overrightarrow{op}^a(\mathfrak{i}, p)\ \big) \\ \hfill .[\mathfrak{m} = \mathfrak{i}.\mathfrak{m}] \\ \cup\ \overrightarrow{op}^r(\mathfrak{i}, p) \hfill \end{pmatrix}$$

Now by definition:

$$\overrightarrow{op}^a(\mathfrak{i}, p) \setminus \big(\ \mathfrak{i}.[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]\ \big) = \overrightarrow{op}^a(\mathfrak{i}, p)$$

And Then:

$$\overrightarrow{op}^a(\mathfrak{i}, p) \setminus \big(\ (\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)).[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]\ \big) = \overrightarrow{op}^a(\mathfrak{i}, p)$$

Thus, adding and deleting $\overrightarrow{op}^a(\mathfrak{i}, p)$ from $\big(\ (\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)).[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]\ \big)$ has no effect. Hence:

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = \big(\ (\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)).[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)].[\mathfrak{m} = \mathfrak{i}.\mathfrak{m}]\ \big) \cup \overrightarrow{op}^r(\mathfrak{i}, p)$$

The double applying of metamodel substitution has no effect:

$$\overleftarrow{op}(\overrightarrow{op}(\mathfrak{i}, p), p, \mathfrak{i}) = (\mathfrak{i} \setminus \overrightarrow{op}^r(\mathfrak{i}, p)) \cup \overrightarrow{op}^r(\mathfrak{i}, p)$$

By definition of $\overrightarrow{op}^r(\mathfrak{i}, p)$:

$$\overrightarrow{op}^r(\mathfrak{i}, p) \subseteq \mathfrak{i}$$

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

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

□

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, μDif 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.

μDif 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 $\mathsf{\mu Dif}$ operator, $\mathfrak{i}$ 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:

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

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

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

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

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

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 $\mathcal{P}$ its parameter domain. Thereafter, we note $\overrightarrow{tool}^a$ (resp. $\overrightarrow{tool}^r$) the function mapping an input *migrated model* to the elements that are *added* (resp. *removed*) by the tool:

$$\begin{aligned} \overrightarrow{tool}^a \quad &: \quad \mathfrak{I} \times \mathcal{P} \rightarrow \mathfrak{I} \\ &\qquad (\mathfrak{i}, p) \ \mapsto tool(\mathfrak{i}, p) \setminus \mathfrak{i} \end{aligned}$$

$$\begin{aligned} \overrightarrow{tool}^r \quad &: \quad \mathfrak{I} \times \mathcal{P} \rightarrow \mathfrak{I} \\ &\qquad (\mathfrak{i}, p) \ \mapsto \mathfrak{i} \setminus tool(\mathfrak{i}, p) \end{aligned}$$

Contextualization diagnostic Let *op* be a given $\mathsf{\mu Dif}$ operator, $\mathfrak{i}_{\mathsf{ini}}$ be a given *initial* model, $\mathfrak{i}$ 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:

$$\{c \in \mathfrak{i}.\mathfrak{m}.\mathcal{C} \mid \exists i \in \overleftarrow{\delta p}^r(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}).\mathcal{I},\ \mathfrak{i}.\delta_{inst}(i) = c\}$$

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

$$\{r \in \mathfrak{i}.\mathfrak{m}.\mathcal{R} \mid \exists(i \mapsto I) \in \overleftarrow{\delta p}^r(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}).\delta_l,\ \mathfrak{i}.\delta_l(i, r) = I\}$$

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

$$\{a \in \mathfrak{i}.\mathfrak{m}.\mathcal{A} \mid \exists(i \mapsto S) \in \overleftarrow{\delta p}^r(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}).\delta_v,\ \mathfrak{i}.\delta_v(i, a) = S\}$$

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 μDif 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 $\overrightarrow{op}^r$, $\overrightarrow{op}^a$, $\overleftarrow{\delta p}^r$ and $\overleftarrow{\delta p}^a$. We also need to *validate* these operations in regard to *conformity* by the definitions of predicates $\overleftarrow{C_{op}}$ and $\overrightarrow{C_{op}}$.

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 $\mathcal{P}$:

$$\begin{array}{llll}
\overrightarrow{op}^r : & \mathfrak{I} \times \mathcal{P} & \to \mathfrak{I} \\
& (\mathfrak{i}, p) & \mapsto \mathfrak{i}^{\emptyset}
\end{array}
\qquad
\begin{array}{llll}
\overrightarrow{op}^a : & \mathfrak{I} \times \mathcal{P} & \to \mathfrak{I} \\
& (\mathfrak{i}, p) & \mapsto \mathfrak{i}^{\emptyset}.[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)]
\end{array}$$

$$\begin{array}{llll}
\overleftarrow{\delta p}^r : & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} & \to \mathfrak{I} \\
& (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \mathfrak{i}^{\emptyset}
\end{array}
\qquad
\begin{array}{llll}
\overleftarrow{\delta p}^a : & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} & \to \mathfrak{I} \\
& (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \mathfrak{i}^{\emptyset}.[\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}]
\end{array}$$

$$\begin{array}{llll}
\overrightarrow{C_{op}} : & \mathfrak{I} \times \mathcal{P} & \to \mathbb{B} \\
& (\mathfrak{i}, p) & \mapsto true
\end{array}
\qquad
\begin{array}{llll}
\overleftarrow{C_{op}} : & \mathfrak{I} \times \mathcal{P} \times \mathfrak{I} & \to \mathbb{B} \\
& (\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) & \mapsto true
\end{array}$$

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

$$\forall(\mathfrak{i}, p) \in \mathfrak{I} \times \mathcal{P},\ \ \mathfrak{i}^{\emptyset} \subseteq \mathfrak{i}$$

$$\forall(\mathfrak{i}, p) \in \mathfrak{I} \times \mathcal{P},\ \ \big((\mathfrak{i}^{\emptyset}.[\mathfrak{m} = op(\mathfrak{i}.\mathfrak{m}, p)].\mathfrak{m})\ =\ op(\mathfrak{i}.\mathfrak{m}, p)\big)\ \wedge\ \big(\mathfrak{i}^{\emptyset} \setminus \mathfrak{i} = \mathfrak{i}^{\emptyset}\big)$$

$$\forall(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \in \mathfrak{I} \times \mathcal{P} \times \mathfrak{I},\ \ \mathfrak{i}^{\emptyset} \subseteq \big((\mathfrak{i} \setminus \overrightarrow{op}(\mathfrak{i}_{\mathsf{ini}}, p)) \cup \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p)\big)\ \wedge\ (\mathfrak{i}^{\emptyset} = \overrightarrow{op}^a(\mathfrak{i}_{\mathsf{ini}}, p))$$

$$\begin{array}{ll}
\forall(\mathfrak{i}, p, \mathfrak{i}_{\mathsf{ini}}) \in \mathfrak{I} \times \mathcal{P} \times \mathfrak{I}, & \big((\mathfrak{i}^{\emptyset}.[\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}].\mathfrak{m})\ =\ \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}\big) \\
& \wedge\ \mathfrak{i}^{\emptyset} \setminus \big(\ \mathfrak{i}.[\mathfrak{m} = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}]\ \big) = \mathfrak{i}^{\emptyset} \\
& \wedge\ \mathfrak{i}^{\emptyset} = \overrightarrow{op}^r(\mathfrak{i}_{\mathsf{ini}}, p)
\end{array}$$

4.6 Model migration by operator in detail

We define now the 22 $\mathsf{\mu Dif}$ operators of table 19. For each of them we only give the *specific* definitions of $\overrightarrow{op}^r$, $\overrightarrow{op}^a$, $\overleftarrow{op}^r$, $\overleftarrow{op}^a$, $\overleftarrow{C_{op}}$ and $\overrightarrow{C_{op}}$. 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.

$$\begin{array}{lll} \overleftarrow{cc}^r & : & \mathfrak{I} \times \mathcal{N} \times \mathfrak{I} \rightarrow \mathfrak{I} \\ & & (\mathfrak{i}, n, \mathfrak{i}_{\mathsf{ini}}) \;\; \mapsto \left\{ \begin{array}{l} \mathfrak{i}^{\emptyset}.[\mathcal{I} = \{i \in \mathcal{I} \mid \delta_{inst}(i) = n\}] \\ \quad .[\delta_{inst} = \{(i, c) \in \mathcal{I} \times \mathfrak{m}.\mathcal{C} \mid c = n \wedge \delta_{inst}(i) = c\}] \end{array} \right. \end{array}$$

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:

$$\forall (n, \mathfrak{i}, \mathfrak{i}_{\mathsf{ini}}) \in \mathcal{N} \times \mathfrak{I}^2, \;\; \overleftarrow{cc}^r(\mathfrak{i}, n, \mathfrak{i}_{\mathsf{ini}}) \subseteq (\mathfrak{i} \setminus \overrightarrow{cc}(\mathfrak{i}_{\mathsf{ini}}, p))$$

Thus, if the migrated model is kept unchanged, *i.e.* if $\mathfrak{i} = \overrightarrow{cc}(\mathfrak{i}_{\mathsf{ini}}, p)$, then $\mathfrak{i} \setminus \overrightarrow{cc}(\mathfrak{i}_{\mathsf{ini}}, p) = \mathfrak{i}^{\emptyset}$. Then in this case we actually have:

$$\overleftarrow{cc}^r(\mathfrak{i}, n, \mathfrak{i}_{\mathsf{ini}}) = \mathfrak{i}^{\emptyset} = \overrightarrow{cc}^a(\mathfrak{i}, n)$$

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 $\mathsf{moveAttributeTypeTo}$). 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.

$$\begin{array}{lll} \overleftarrow{ca}^r : & \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} & \rightarrow \mathfrak{I} \\ & (\mathfrak{i}, n_a, n_c, n_d, \mathfrak{i}_{\mathsf{ini}}) \mapsto & \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad\qquad a = (n_c, n_a) \wedge \delta_v(i, a) = S\}] \\ \quad .[\mathcal{S} = \{s \in \mathcal{S} \mid \forall (i, a) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A},\ s \notin \delta_v(i, a)\}] \end{cases} \end{array}$$

Validity A valid model transformed by $\overrightarrow{ca}$ 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 $\overleftarrow{ca}$ 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.

$$\begin{array}{lll} \overleftarrow{cr}^r : & \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} & \rightarrow \mathfrak{I} \\ & (\mathfrak{i}, n_r, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) \mapsto & \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \\ \qquad\qquad r = (n_c, n_r) \wedge \delta_l(i, r) = S\}] \end{cases} \end{array}$$

Validity As for new attributes, a valid model transformed by $\overrightarrow{cr}$ 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 $\overleftarrow{cr}$ 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.*

$$\begin{array}{lll} \overrightarrow{dc}^r : & \mathfrak{I} \times \mathcal{N} & \to \mathfrak{I} \\ & (\mathfrak{i}, n) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\mathcal{I} = \{i \in \mathcal{I} \mid \delta_{inst}(i) = n\}] \\ \quad .[\delta_{inst} = \{(i, c) \in \mathcal{I} \times \mathfrak{m}.\mathcal{C} \mid c = n \wedge \delta_{inst}(i) = c\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad\qquad \delta_{inst}(i) = n \wedge \delta_l(i, r) = S\}] \\ \quad .[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad\qquad \delta_{inst}(i) = n \wedge \delta_v(i, a) = S\}] \end{cases} \end{array}$$

$$\begin{array}{lll} \overleftarrow{dc}^a : & \mathfrak{I} \times \mathcal{N} \times \mathfrak{I} \to \mathfrak{I} & \\ & (\mathfrak{i}, n, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\mathcal{I} = \{i \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \mid \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n\}] \\ \quad .[\delta_{inst} = \{(i, c) \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \times \mathcal{N} \mid c = n \wedge \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = c\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \times \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad\quad \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n \wedge \forall j \in S, j \in (\delta_l \cap \mathfrak{i}_{\mathsf{ini}}.\delta_l)(i, r)\}] \\ \quad .[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad\quad \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(v) = n \wedge \forall j \in S, j \in (\delta_v \cap \mathfrak{i}_{\mathsf{ini}}.\delta_v)(i, a)\}] \end{cases} \end{array}$$

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:

$$\begin{array}{ll} \overleftarrow{C_{dc}} : & \mathfrak{I} \times \mathcal{N} \times \mathfrak{I} \to \mathbb{B} \\ & (\mathfrak{i}, n, \mathfrak{i}_{\mathsf{ini}}) \mapsto (\mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \setminus \mathfrak{i}.\mathcal{I}) \cap \cup_{i \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I}} (\mathfrak{i}_{\mathsf{ini}}.\delta_l(i)) = \emptyset \end{array}$$

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.

$$
\begin{array}{rcll}
\overrightarrow{da}^r & : & \mathfrak{I} \times \mathcal{N}^2 & \rightarrow \mathfrak{I} \\
 & & (\mathfrak{i}, n_a, n_c) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad\qquad a = (n_c, n_a) \wedge \delta_v(i, (n_c, n_a)) = S\}] \end{cases}
\end{array}
$$

$$
\begin{array}{rcll}
\overleftarrow{da}^a & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \rightarrow \mathfrak{I} \\
 & & (\mathfrak{i}, n_a, n_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_v = \{(i, a, S) \in \mathfrak{i}.\mathcal{I} \times \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathfrak{i}_{\mathsf{ini}}.\mathcal{S}) \mid \\ \qquad\qquad a = (n_c, n_a) \wedge \mathfrak{i}_{\mathsf{ini}}.\delta_v(i, (n_c, n_a)) = S\}] \end{cases}
\end{array}
$$

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:

$$
\begin{array}{rll}
C_1 : & \mathcal{N}^2 \times \mathfrak{i} & \rightarrow \mathbb{B} \\
 & (n_a, n_c, \mathfrak{i}) & \mapsto \exists((x, y), d) \in \mathcal{M} \times \mathcal{D}, (\mathfrak{i}.\mathfrak{m}.\delta_A(n_c, n_a) = ((x, y), d) \wedge x = 0)
\end{array}
$$

$$
\begin{array}{rll}
C_2 : & \mathcal{N} \times \mathfrak{i}^2 & \rightarrow \mathbb{B} \\
 & (n_c, \mathfrak{i}, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \{i \in \mathcal{I} \mid \mathfrak{i}.\delta_{inst}(i) = n_c\} \setminus \{i \in \mathcal{I} \mid \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c\} \ = \ \emptyset
\end{array}
$$

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:

$$
\begin{array}{rcll}
\overleftarrow{C_{da}} & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \rightarrow \mathbb{B} \\
 & & (\mathfrak{i}, n_a, n_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto (C_1(n_a, n_c, \mathfrak{i}) \vee C_2(n_c, \mathfrak{i}, \mathfrak{i}_{\mathsf{ini}}))
\end{array}
$$

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.

$$\begin{array}{lll} \overrightarrow{dr}^r : & \mathfrak{I} \times \mathcal{N}^2 & \to \mathfrak{I} \\ & (\mathfrak{i}, n_r, n_c) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad\qquad r = (n_c, n_r) \wedge \delta_l(i, (n_c, n_r)) = S\}] \end{cases} \end{array}$$

$$\begin{array}{lll} \overleftarrow{dr}^a : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \to \mathfrak{I} \\ & (\mathfrak{i}, n_r, n_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \begin{cases} \mathfrak{i}^{\emptyset}.[\delta_l = \{(i, r, S) \in \mathfrak{i}.\mathcal{I} \times \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathfrak{i}.\mathcal{I}) \mid \\ \qquad\qquad r = (n_c, n_r) \wedge \forall j \in S, j \in \mathfrak{i}_{\mathsf{ini}}.\delta_l(i, r)\}] \end{cases} \end{array}$$

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:

$$\begin{array}{lll} C_1' : & \mathcal{N}^2 \times \mathfrak{i} & \to \mathbb{B} \\ & (n_r, n_c, \mathfrak{i}) & \mapsto \exists((x, y), c) \in \mathcal{M} \times \mathcal{C}, (\mathfrak{i}.\mathfrak{m}.\delta_R(n_c, n_r) = ((x, y), c) \wedge x = 0) \end{array}$$

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:

$$\begin{array}{lll} \overleftarrow{C_{dr}} : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \to \mathbb{B} \\ & (\mathfrak{i}, n_r, n_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \big((C_1'(n_r, n_c, \mathfrak{i}) \vee C_2(n_c, \mathfrak{i}, \mathfrak{i}_{\mathsf{ini}})) \big) \\ & & \quad \wedge (\mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \setminus \mathfrak{i}.\mathcal{I}) \cap \cup_{i \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I}}(\mathfrak{i}_{\mathsf{ini}}.\delta_l(i)) = \emptyset \end{array}$$

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.

$$\begin{array}{lcll}
\overrightarrow{sn_c}^r & : & \mathfrak{I} \times \mathcal{N}^2 & \to \mathfrak{I} \\
 & & (\mathfrak{i}, n_c, n'_c) & \mapsto \{\, \mathfrak{i}^{\emptyset}.[\delta_{inst} = \{(i,c) \in \mathcal{I} \times \mathfrak{m}.\mathcal{C} \mid c = n_c \wedge \delta_{inst}(i) = n_c\}] \\
 & & & \\
\overrightarrow{sn_c}^a & : & \mathfrak{I} \times \mathcal{N}^2 & \to \mathfrak{I} \\
 & & (\mathfrak{i}, n_c, n'_c) & \mapsto \{\, \mathfrak{i}^{\emptyset}.[\delta_{inst} = \{(i,c) \in \mathcal{I} \times \mathcal{N} \mid c = n'_c \wedge \delta_{inst}(i) = n_c\}] \\
 & & & \\
\overleftarrow{sn_c}^r & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \to \mathfrak{I} \\
 & & (\mathfrak{i}, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \{\, \mathfrak{i}^{\emptyset}.[\delta_{inst} = \{(i,c) \in \mathcal{I} \times \mathfrak{m}.\mathcal{C} \mid c = n'_c \wedge \delta_{inst}(i) = n'_c\}] \\
 & & & \\
\overleftarrow{sn_c}^a & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \to \mathfrak{I} \\
 & & (\mathfrak{i}, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \{\, \mathfrak{i}^{\emptyset}.[\delta_{inst} = \{(i,c) \in \mathcal{I} \times \mathcal{N} \mid c = n_c \wedge \delta_{inst}(i) = n'_c\}]
\end{array}$$

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.

$$\overrightarrow{snf}^r : \quad \mathfrak{I} \times \mathcal{N}^3 \quad \to \mathfrak{I}$$
$$(\mathfrak{i}, n_c, n_f, n'_f) \mapsto \begin{cases} \mathfrak{i}^\emptyset.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad a = (n_c, n_f) \land \delta_v(i, (n_c, n_f)) = S\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad r = (n_c, n_f) \land \delta_l(i, (n_c, n_f)) = S\}] \end{cases}$$

$$\overrightarrow{snf}^a : \quad \mathfrak{I} \times \mathcal{N}^3 \quad \to \mathfrak{I}$$
$$(\mathfrak{i}, n_c, n_f, n'_f) \mapsto \begin{cases} \mathfrak{i}^\emptyset.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad a = (n_c, n'_f) \land \delta_v(i, (n_c, n_f)) = S\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad r = (n_c, n'_f) \land \delta_l(i, (n_c, n_f)) = S\}] \end{cases}$$

$$\overleftarrow{snf}^r : \quad \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} \quad \to \mathfrak{I}$$
$$(\mathfrak{i}, n_c, n_f, n'_f, \mathfrak{i}_{\mathsf{ini}}) \mapsto \begin{cases} \mathfrak{i}^\emptyset.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad a = (n_c, n'_f) \land \delta_v(i, (n_c, n'_f)) = S\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad r = (n_c, n'_f) \land \delta_l(i, (n_c, n'_f)) = S\}] \end{cases}$$

$$\overleftarrow{snf}^a : \quad \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} \quad \to \mathfrak{I}$$
$$(\mathfrak{i}, n_c, n_f, n'_f, \mathfrak{i}_{\mathsf{ini}}) \mapsto \begin{cases} \mathfrak{i}^\emptyset.[\delta_v = \{(i, a, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{S}) \mid \\ \qquad a = (n_c, n_f) \land \delta_v(i, (n_c, n'_f)) = S\}] \\ \quad .[\delta_l = \{(i, r, S) \in \mathcal{I} \times \mathcal{N}^2 \times \mathcal{P}(\mathcal{I}) \mid \\ \qquad r = (n_c, n_f) \land \delta_l(i, (n_c, n'_f)) = S\}] \end{cases}$$

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:

$$
\begin{array}{rlll}
\overrightarrow{C_{slb}} & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} & \to \mathbb{B} \\
& & (\mathfrak{i}, n_c, n_f, x) & \mapsto \forall i \in \mathfrak{i}.\mathcal{I}, \big((|\delta_v(i,(n_c,n_f))| + |\delta_l(i,(n_c,n_f))|) \geq x\big)
\end{array}
$$

$$
\begin{array}{rll}
\overleftarrow{C_{slb}} & : \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} \times \mathfrak{I} & \to \mathbb{B} \\
& (\mathfrak{i}, n_c, n_f, x, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \text{let } ((x_{\mathsf{ini}}, y), n) = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.(\delta_A \cup \delta_R)(n_c, n_f) \text{ in:} \\
& & \quad \forall i \in \mathfrak{i}.\mathcal{I}, \big((|\delta_v(i,(n_c,n_f))| + |\delta_l(i,(n_c,n_f))|) \geq x_{\mathsf{ini}}\big)
\end{array}
$$

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* links, we need here to *randomly removing* links when they exceed the new upper bound during the migration or during the recontextualization. We also decided to avoid random actions during these transformations:

$$
\begin{array}{rlll}
\overrightarrow{C_{sub}} & : & \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} & \to \mathbb{B} \\
& & (\mathfrak{i}, n_c, n_f, y) & \mapsto \forall i \in \mathfrak{i}.\mathcal{I}, \big((|\delta_v(i,(n_c,n_f))| + |\delta_l(i,(n_c,n_f))|) \leq y\big)
\end{array}
$$

$$
\begin{array}{rll}
\overleftarrow{C_{sub}} & : \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{N} \times \mathfrak{I} & \to \mathbb{B} \\
& (\mathfrak{i}, n_c, n_f, y, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \text{let } ((x, y_{\mathsf{ini}}), n) = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.(\delta_A \cup \delta_R)(n_c, n_f) \text{ in:} \\
& & \quad \forall i \in \mathfrak{i}.\mathcal{I}, \big((|\delta_v(i,(n_c,n_f))| + |\delta_l(i,(n_c,n_f))|) \leq y_{\mathsf{ini}}\big)
\end{array}
$$

The migration preserves validity if the number of corresponding links is not greater than the new upper bound in the case of its decrease ($\overrightarrow{C_{sub}}$). The recontextualization preserves validity if there is no extra new links in the case of a upper bound increase ($\overleftarrow{C_{sub}}$).

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:

$$
\begin{array}{rll}
\overrightarrow{C_{sa}} & : \mathfrak{I} \times \mathcal{N} \times \mathbb{B} & \to \mathbb{B} \\
& (\mathfrak{i}, n_c, b) & \mapsto b \implies \{i \in \mathfrak{i}.\mathcal{I} \mid \mathfrak{i}.\delta_{inst}(i) = n_c\} = \emptyset \\
\overleftarrow{C_{sa}} & : \mathfrak{I} \times \mathcal{N} \times \mathbb{B} \times \mathfrak{I} & \to \mathbb{B} \\
& (\mathfrak{i}, n_c, b, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \neg b \implies \{i \in \mathfrak{i}.\mathcal{I} \mid \mathfrak{i}.\delta_{inst}(i) = n_c\} = \emptyset
\end{array}
$$

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.

$$\begin{array}{lll}
\overrightarrow{C_{sc}} : & \mathfrak{I} \times \mathcal{N}^2 \times \mathbb{B} & \to \mathbb{B} \\
& (\mathfrak{i}, n_c, n_r, b) & \mapsto b \implies \forall i \in \left\{ \bigcup_{j \in \mathfrak{i}.\mathcal{I}} \left(\mathfrak{i}.\delta_l(j, (n_c, n_r)) \right) \right\}, \\
& & \left(\exists (i', (n'_c, n'_r)) \in \mathfrak{i}.\mathcal{I} \times \mathfrak{i}.\mathfrak{m}.\mathcal{R}, i \in \mathfrak{i}.\delta_l(i', (n'_c, n'_r)) \right) \\
& & \implies (n'_c, n'_r) \notin \mathfrak{i}.\mathfrak{m}.\mathcal{R}_{\mathcal{C}} \\
\overleftarrow{C_{sc}} : & \mathfrak{I} \times \mathcal{N} \times \mathbb{B} \times \mathfrak{I} \to \mathbb{B} & \\
& (\mathfrak{i}, n_c, n_r, b, \mathfrak{i}_{\mathsf{ini}}) \mapsto \neg b & \implies \forall i \in \left\{ \bigcup_{j \in \mathfrak{i}.\mathcal{I}} \left(\mathfrak{i}.\delta_l(j, (n_c, n_r)) \right) \right\}, \\
& & \left(\exists (i', (n'_c, n'_r)) \in \mathfrak{i}.\mathcal{I} \times \mathfrak{i}.\mathfrak{m}.\mathcal{R}, i \in \mathfrak{i}.\delta_l(i', (n'_c, n'_r)) \right) \\
& & \implies (n'_c, n'_r) \notin \mathfrak{i}.\mathfrak{m}.\mathcal{R}_{\mathcal{C}}
\end{array}$$

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:

$$\begin{array}{rcll}\overrightarrow{C_{mft}} & : & \mathfrak{I}\times\mathcal{N}^3 & \to \mathbb{B}\\ & & (\mathfrak{i}, n_c, n_f, n_c') & \mapsto \left\{\bigcup_{j\in\mathfrak{i}.\mathcal{I}}\left(\mathfrak{i}.(\delta_v\cup\delta_l)(j,(n_c,n_f))\right)\right\} = \emptyset\end{array}$$

$$\begin{array}{rcll}\overleftarrow{C_{mft}} & : & \mathfrak{I}\times\mathcal{N}^3\mathbb{B}\times\mathfrak{I} & \to \mathbb{B}\\ & & (\mathfrak{i}, n_c, n_f, n_c', \mathfrak{i}_{\mathsf{ini}}) & \mapsto \left\{\bigcup_{j\in\mathfrak{i}.\mathcal{I}}\left(\mathfrak{i}.(\delta_v\cup\delta_l)(j,(n_c',n_f))\right)\right\} = \emptyset\end{array}$$

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:

$$\begin{array}{rcll}\overrightarrow{C_{mrtt}} & : & \mathfrak{I}\times\mathcal{N}^3 & \to \mathbb{B}\\ & & (\mathfrak{i}, n_c, n_r, n_c') & \mapsto \left\{\bigcup_{j\in\mathfrak{i}.\mathcal{I}}\left(\mathfrak{i}.\delta_l(j,(n_c,n_r))\right)\right\} = \emptyset\end{array}$$

$$\begin{array}{rcll}\overleftarrow{C_{mrtt}} & : & \mathfrak{I}\times\mathcal{N}^3\times\mathfrak{I} & \to \mathbb{B}\\ & & (\mathfrak{i}, n_c, n_r, n_c', \mathfrak{i}_{\mathsf{ini}}) & \mapsto \left\{\bigcup_{j\in\mathfrak{i}.\mathcal{I}}\left(\mathfrak{i}.\delta_l(j,(n_c,n_r))\right)\right\} = \emptyset\end{array}$$

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 μDif 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.

$$
\begin{array}{l}
\overleftarrow{asc}^r \ : \ \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} \ \rightarrow \mathfrak{I} \\
\qquad (\mathfrak{i}, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) \mapsto \begin{cases}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i,(c,n_a),S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\
\quad \delta_{inst}(i) = n_c \wedge \delta_v(i,(c,n_a)) = S \wedge c \in \mathfrak{m}.\alpha'_I(n'_c)\}] \\
\ .[\delta_l = \{(i,(c,n_r),S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{S}) \mid \\
\quad \delta_{inst}(i) = n_c \wedge \delta_l(i,(c,n_r)) = S \wedge c \in \mathfrak{m}.\alpha'_I(n'_c)\}]
\end{cases}
\end{array}
$$

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:

$$
\begin{array}{l}
\overrightarrow{C_{asc}} \ : \ \mathfrak{I} \times \mathcal{N}^2 \ \rightarrow \mathbb{B} \\
\quad (\mathfrak{i}, n_c, n'_c) \mapsto \forall c \in \mathfrak{i}.\mathfrak{m}.\alpha'_I(n'_c), \forall((n_f,n),(x,y)) \in \mathcal{N}^2 \times \mathcal{M} \\
\qquad\qquad ((x,y),n) = \mathfrak{i}.\mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}})(c,n_f) \implies x = 0
\end{array}
$$

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.

$$
\begin{array}{l}
\overrightarrow{rsc}^r \ : \ \mathfrak{I} \times \mathcal{N}^2 \times \ \rightarrow \mathfrak{I} \\
\qquad (\mathfrak{i}, n_c, n'_c) \ \mapsto \begin{cases}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i,(c,n_a),S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{A} \times \mathcal{P}(\mathcal{S}) \mid \\
\quad \delta_{inst}(i) = n_c \wedge \delta_v(i,(c,n_a)) = S \wedge c \in \mathfrak{m}.\alpha'_I(n'_c)\}] \\
\ .[\delta_l = \{(i,(c,n_r),S) \in \mathcal{I} \times \mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathcal{S}) \mid \\
\quad \delta_{inst}(i) = n_c \wedge \delta_l(i,(c,n_r)) = S \wedge c \in \mathfrak{m}.\alpha'_I(n'_c)\}]
\end{cases}
\end{array}
$$

$$
\begin{array}{l}
\overleftarrow{rsc}^a \ : \ \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} \ \rightarrow \mathfrak{I} \\
\qquad (\mathfrak{i}, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) \mapsto \begin{cases}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i,(c,n_a),S) \in \mathfrak{i}_{\mathsf{ini}}.\delta_v \mid i \in \mathfrak{i}.\mathcal{I} \\
\quad \wedge\, \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c \wedge c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha'_I(n'_c)\}] \\
\ .[\delta_l = \{(i,(c,n_r),S) \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \times \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathfrak{i}_{\mathsf{ini}}.\mathcal{I}) \mid \\
\quad i \in \mathfrak{i}.\mathcal{I} \wedge \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c \wedge c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha'_I(n'_c) \\
\quad \wedge\, \forall i' \in S, (i' \in \mathfrak{i}_{\mathsf{ini}}.\delta_l(i,(c,n_r)) \wedge i' \in \mathfrak{i}.\mathcal{I})\}]
\end{cases}
\end{array}
$$

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:

$$\begin{array}{rl}\overleftarrow{C_{rsc}} : \mathfrak{I} \times \mathcal{N}^2 \times \mathfrak{I} & \rightarrow \mathbb{B} \\ (\mathfrak{i}, n_c, n'_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \big(\forall c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha'_I(n'_c), \forall ((n_f, n), (x, y)) \in \mathcal{N}^2 \times \mathcal{M} \\ & \quad ((x, y), n) = \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}})(c, n_f) \implies x = 0 \big) \\ & \quad \bigvee \big(\{i \in \mathfrak{i}.\mathcal{I} \mid \mathfrak{i}.\delta_{inst}(i) = n_c\} \\ & \qquad \setminus \{i \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \mid \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c\} = \emptyset \big) \end{array}$$

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.

$$\begin{array}{rl}\cap_a : \mathfrak{M} \times \mathcal{C}^2 & \rightarrow \mathcal{A} \times \mathcal{M} \times \mathcal{D} \\ (\mathfrak{m}, (c_1, c_2)) & \mapsto \{((c, n), m, d) \in \mathfrak{m}.\delta_A \mid c \in \alpha'_I(c_1) \\ & \quad \wedge \exists c' \in \mathcal{C}\, (\mathfrak{m}.\delta_A(c', n) = (m, d) \wedge c' \in \alpha'_I(c_2))\end{array}$$

$$\begin{array}{rl}\cap_r : \mathfrak{M} \times \mathcal{C}^2 & \rightarrow \mathcal{R} \times \mathcal{M} \times \mathcal{C} \\ (\mathfrak{m}, (c_1, c_2)) & \mapsto \{((c, n), m, c_t) \in \mathfrak{m}.\delta_R \mid c \in \alpha'_I(c_1) \\ & \quad \wedge \exists (c', c'_t) \in \mathcal{C}^2, (\mathfrak{m}.\delta_R(c', n) = (m, c'_t) \\ & \quad \wedge c' \in \alpha'_I(c_2)) \wedge c_t \in \alpha'_I(c'_t))\end{array}$$

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.

$$
\begin{array}{lll}
\overrightarrow{msct}^r : & \mathfrak{I} \times \mathcal{N}^3 \times & \to \mathfrak{I} \\
& (\mathfrak{i}, n_c, n'_c, n''_c) & \mapsto \left\{ \begin{array}{l}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i, (c, n_a), S) \in \mathfrak{i}.\delta_v \mid \mathfrak{i}.\delta_{inst}(i) = n_c \\
\quad \wedge\ \forall (m, n), ((c, n_a), m, n) \notin \cap_a(\mathfrak{i}.\mathfrak{m}, n'_c, n''_c) \\
\quad \wedge\ c \in \mathfrak{m}.\alpha'_I(n'_c)\}] \\
\ .[\delta_l = \{(i, (c, n_r), S) \in \mathfrak{i}.\delta_l \mid \mathfrak{i}.\delta_{inst}(i) = n_c \\
\quad \wedge\ \forall (m, n), ((c, n_r), m, n) \notin \cap_r(\mathfrak{i}.\mathfrak{m}, n'_c, n''_c) \\
\quad \wedge\ c \in \mathfrak{m}.\alpha'_I(n'_c)\}]
\end{array} \right.
\end{array}
$$

$$
\begin{array}{lll}
\overleftarrow{msct}^r : & \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} & \to \mathfrak{I} \\
& (\mathfrak{i}, n_c, n'_c, n''_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \left\{ \begin{array}{l}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i, (c, n_a), S) \in \mathfrak{i}.\delta_v \mid \mathfrak{i}.\delta_{inst}(i) = n_c \\
\quad \wedge\ \forall (m, n), ((c, n_a), m, n) \notin \cap_a(\mathfrak{i}.\mathfrak{m}, n''_c, n'_c) \\
\quad \wedge\ c \in \mathfrak{m}.\alpha'_I(n''_c)\}] \\
\ .[\delta_l = \{(i, (c, n_r), S) \in \mathfrak{i}.\delta_l \mid \mathfrak{i}.\delta_{inst}(i) = n_c \\
\quad \wedge\ \forall (m, n), ((c, n_r), m, n) \notin \cap_r(\mathfrak{i}.\mathfrak{m}, n''_c, n'_c) \\
\quad \wedge\ c \in \mathfrak{m}.\alpha'_I(n''_c)\}]
\end{array} \right.
\end{array}
$$

$$
\begin{array}{lll}
\overleftarrow{msct}^a : & \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} & \to \mathfrak{I} \\
& (\mathfrak{i}, n_c, n'_c, n''_c, \mathfrak{i}_{\mathsf{ini}}) & \mapsto \left\{ \begin{array}{l}
\mathfrak{i}^{\emptyset}.[\delta_v = \{(i, (c, n_a), S) \in \mathfrak{i}_{\mathsf{ini}}.\delta_v \mid i \in \mathfrak{i}.\mathcal{I} \\
\quad \wedge\ \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c \wedge c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha'_I(n'_c) \\
\quad \wedge\ \forall (m, n), ((c, n_a), m, n) \notin \cap_a(\mathfrak{i}.\mathfrak{m}, n'_c, n''_c)\}] \\
\ .[\delta_l = \{(i, (c, n_r), S) \in \mathfrak{i}.\mathcal{I} \times \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\mathcal{R} \times \mathcal{P}(\mathfrak{i}.\mathcal{I}) \mid \\
\quad \wedge\ \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c \wedge c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha'_I(n'_c) \\
\quad \wedge\ \forall (m, n), ((c, n_r), m, n) \notin \cap_r(\mathfrak{i}.\mathfrak{m}, n'_c, n''_c) \\
\quad \wedge\ \forall i' \in S, (i' \in \mathfrak{i}_{\mathsf{ini}}.\delta_l(i, (c, n_r))\}]
\end{array} \right.
\end{array}
$$

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 corresponds to mandatory features.

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

$$\begin{array}{rcl}
\overrightarrow{C_{msct}}: & \mathfrak{I} \times \mathcal{N}^3 & \rightarrow \mathbb{B} \\
& (\mathfrak{i}, n_c, n_c', n_c'') & \mapsto \forall c \in \mathfrak{i}.\mathfrak{m}.\alpha_I'(n_c''), \forall((n_f, n), (x, y)) \in \mathcal{N}^2 \times \mathcal{M} \\
& & \big(((c, n_f), (x, y), n) \notin (\cap_a \cup \cap_r)(\mathfrak{i}.\mathfrak{m}, n_c', n_c'') \\
& & \wedge ((x, y), n) = \mathfrak{i}.\mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}})(c, n_f) \big) \implies x = 0
\end{array}$$

$$\begin{array}{rcl}
\overleftarrow{C_{msct}}: & \mathfrak{I} \times \mathcal{N}^3 \times \mathfrak{I} & \rightarrow \mathbb{B} \\
& (\mathfrak{i}, n_c, n_c', n_c'', \mathfrak{i}_{\mathsf{ini}}) & \mapsto \big(\forall c \in \mathfrak{i}_{\mathsf{ini}}.\mathfrak{m}.\alpha_I'(n_c'), \forall((n_f, n), (x, y)) \in \mathcal{N}^2 \times \mathcal{M} \\
& & \big(((c, n_f), (x, y), n) \notin (\cap_a \cup \cap_r)(\mathfrak{i}.\mathfrak{m}, n_c'', n_c') \\
& & \wedge ((x, y), n) = \mathfrak{i}.\mathfrak{m}.(\delta_{\mathcal{A}} \cup \delta_{\mathcal{R}})(c, n_f) \big) \implies x = 0\big) \\
& & \bigvee \big(\{i \in \mathfrak{i}.\mathcal{I} \mid \mathfrak{i}.\delta_{inst}(i) = n_c\} \\
& & \setminus \{i \in \mathfrak{i}_{\mathsf{ini}}.\mathcal{I} \mid \mathfrak{i}_{\mathsf{ini}}.\delta_{inst}(i) = n_c\} = \emptyset \big)
\end{array}$$

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.

$$\begin{array}{rcl}
[\![\cdot]\!]_{mig} & : & \mathcal{L}(mig) \rightarrow \mathfrak{I} \\
s & \mapsto & \begin{cases}
s \text{ matches with } (\{\ mod\ \}\ op) & : \\
\quad \overrightarrow{[\![op]\!]_{op}}\big([\![mod]\!]_{mod}\ ,\ [\![op]\!]_{param}\big) & \\
s \text{ matches with } (s'\ op) \text{ where } s' = (\{\ mod\ \}\ (op')^+\ ) & : \\
\quad \overrightarrow{[\![op]\!]_{op}}\big([\![s']\!]_{mig}\ ,\ [\![op]\!]_{param}\big) &
\end{cases}
\end{array}$$

Fig. 31. Valuation of µDif specifications including model migration

4.7 Specifications

We note $\mathcal{L}(mig)$ the sets of words yielded from *mig* in figure 30. Basically, $\mathcal{L}(mig)$ contains a specification made of one model *mod* (including its metamodel) plus an ordered non-empty set of operators applied to *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 $\mathcal{L}(mig)$. 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

1. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf.

List of Figures

Table of Contents