

MOLA syntax

0. Introduction

This document describes the precise syntax of MOLA and some elements of semantics. Since MOLA is a mixed graphical/textual language, for graphical elements only the abstract syntax via a metamodel is provided. For textual elements (various kinds of expressions and statements) the traditional BNF is provided. A transformation in MOLA consists of **one class** (metamodel) **diagram** and one or more **MOLA diagrams**, one of which must be main. For each of the diagrams first its metamodel is provided, then classes are briefly described, and where relevant, for textual elements the BNF is given. Terminal symbols are bold in BNF expressions, BNF notation elements themselves – in blue color. Highlighted syntax **elements** - not yet implemented.

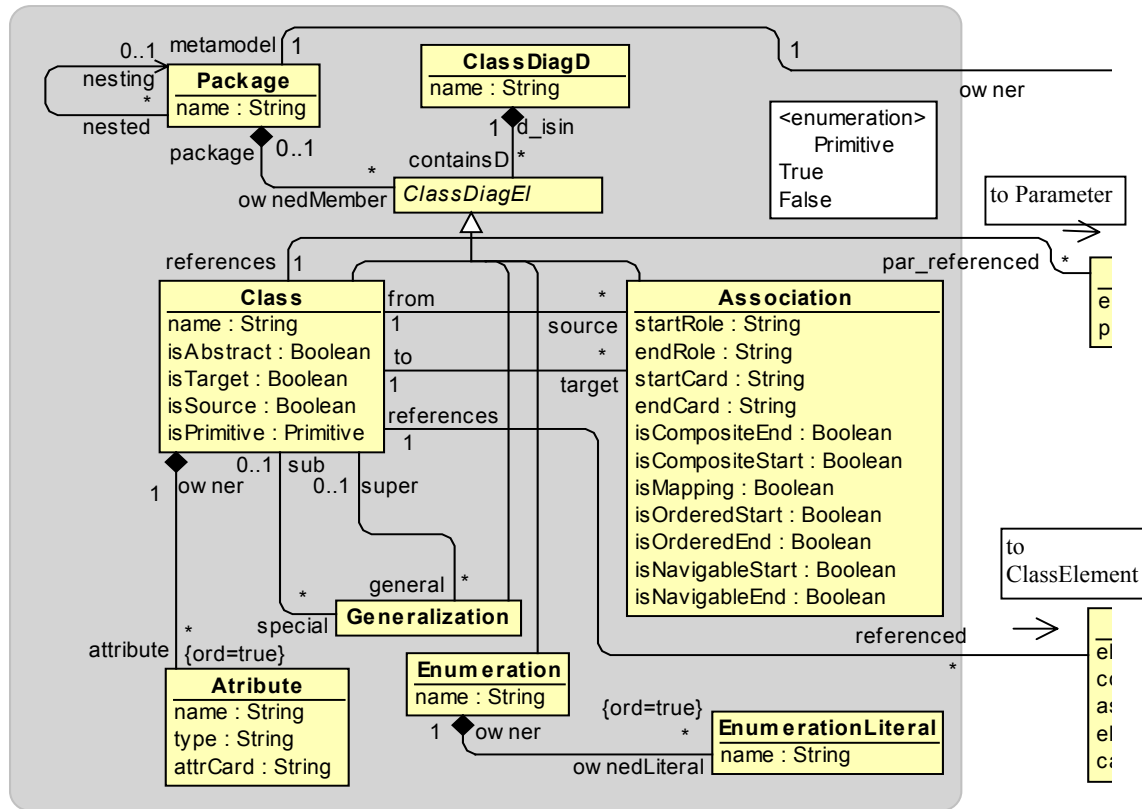
1. Lexics

Identifiers (names) in MOLA can contain letters, digits, underscore ("_"), but no other special characters (and no blanks). Names are case-sensitive. A name must start with a letter or underscore.

2. Class (metamodel) diagram

Class diagram in MOLA is used to define the both the source metamodel describing source models to be transformed and the target metamodel describing the resulting model. Both these metamodels must be combined in one class diagram. Source and target metamodels may coincide (for update transformations). In addition, the class diagram may contain temporary classes and associations used during the transformation execution and mapping associations for documenting the mapping between source and target models. All these kinds of class diagram elements have the same syntax and semantics, the difference is only in the way how support tools (import, export et al.) treat instances of the corresponding metaclasses. From the functionality and user point of view the class diagram in MOLA is equivalent to EMOF (as part of MOF 2.0). However, its internal metamodel is actually that used in early versions of UML (in order to simplify the MOLA tool support).

Metamodel of **class** diagram:



Class is the standard UML/EMOF class.

The attributes **isTarget**, **isSource** are used to specify the role of the class in transformation. Any combinations are meaningful, both being false means the class is a temporary one. The **isPrimitive** attribute is a technical facility used for introducing primitive data types in MOLA.

isPrimitive ::= True | False /* the value **True** is used to introduce primitive types as predefined classes (with one attribute **value** of the corresponding type), all "normal" classes have the value **False** here – this attribute is of type Enum, not Boolean, due to requirements of editor definition in EBM

Association is the standard UML association, with its both end properties included. Start and end of an association just determine its drawing direction, they have no semantic meaning.

startCard ::= assocCard

endCard ::= assocCard

assocCard ::= 1 | 0..1 | * | 1..* /* the default is 1

startRole ::= name | #name / # is used if **isMapping=true**

endRole ::= name | #name / # is used if **isMapping=true**,

/* **isComposite**, **isNavigable** is not used in MOLA semantics, but is used for import/export definition

/* **isOrdered** currently is not used

Attribute is the standard UML attribute (but not navigable association end!)

attrCard ::= 1 | 0..1 | * | 1..* /* default is 1, * currently not supported for attributes

type ::= **String** | **Integer** | **Boolean** | enumerationName /* enumeration must be defined

attributeName ::= name | tempAttrName

tempAttrName ::= ?name /* currently not supported

/* Association Class -> attribute is ordered in the MM

/* The name attribute is mandatory in all metaclasses where it appears (MOLA editor guarantees its presence)

/* The type metaattribute in Attribute is mandatory, presence is not guaranteed by the editor

/* All other metaattributes are optional. For all booleans the default is *false*. For cardinalities the default is specified above, other metaattributes have no defaults

/* Standard inheritance semantics is assumed – subclasses inherit attributes and associations from superclasses, but generalization as such is only

/* indirectly used in MOLA currently, **single inheritance permitted only !!!**

/* MOLA – a superclass may be the metaclass referenced in MOLA element, then actually **instances of each subclass match to this element**

/* Attributes / associations from superclasses are explicitly added to subclasses during compilation, it means associations duplicated in internal tables

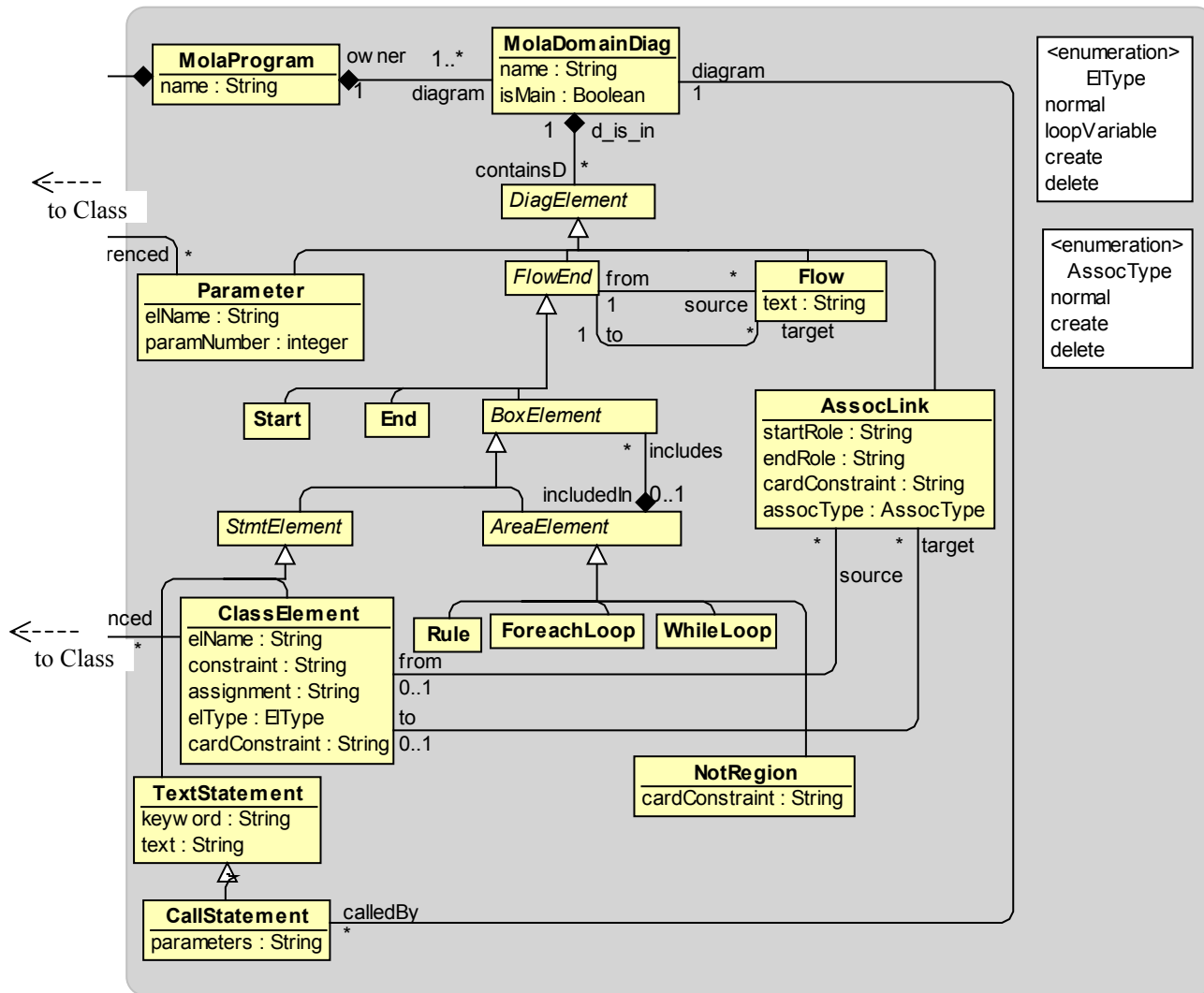
/* Additional metaattributes for association (2* isNavigable, 2* isOrdered) are stored in metatables after compilation – they are used for

/* XMI export/import configuration, but currently will not be directly used in transformations.

/* Metaclass *Package* (with its associations) currently is not explicitly used in MOLA, but is needed for XMI export/import configuration.

3. MOLA diagram

Metamodel of the MOLA diagram:



Highlighted syntax **elements** in this document are not yet implemented !!!

Abstract classes: **DiagElement**, **FlowEnd**, **BoxElement**, **StmtElement**, **AreaElement**, **Loop**

"Container" classes (which may be an independent part of a MOLA diagram) : **Start**, **End**, **TextStatement**, **CallStatement**, **Rule** (LoopHead is an informal subtype of Rule, there is no such metaclass!), **ForeachLoop**, **WhileLoop**, **NOT_region**, **Parameter**

"Element" classes (which must be part of Rule) : **ClassElement**, **AssocLink**

Area elements – Rule and ForeachLoop have no text, context (inclusion) rules for them are described in section 4.

ClassElement is the main element of Rule (or Loop head), used for defining a pattern.

```

elName ::= name | referenceName | /* class element name, the class itself is found via the association references, which points to a Class
      self          /* self is used only in navigation expressions – for navigation from this element
referenceName ::= @name /* see comments below (rule 12), where the referenced element must be located
      /* elType describes the role/action of the element in the pattern – normal means just match, other values are self-descriptive
      /* The most complicated parts of an element are constraint and assignment. Constraint must evaluate to true for a class instance to match the
      /* element in pattern matcing, assigment defines an attribute modification – either in existing (matched) or just created class instance.
constraint ::= simpleExtOCLExpr /* though a small subset of OCL is implemented, there are also few extensions
simpleExtOCLExpr ::= OCLboolExpr /* constraint is always a Boolean expression
assignment ::= assign { NL assign }* /* newLine is used as a separator – each assignment in a separate line
assign ::= attrName := simpleExpr /* type must be compatible, attrName – an attribute of this element class, may be temporary – with ? prefix
simpleExpr ::= intExpr | stringExpr | boolSimpleExpr | enumSimpleExpr
      /* isCopy metaattribute is ignored for elements now, therefore not shown in the metamodel
elemTerm ::= attrSpec | constant | primitiveParam /* primitiveParam may be of type String, Integer, Boolean, the user syntax is just the parameter
      name (as is, with the @ prefix). The MOLA compiler converts a primitiveParam reference to a reference to the value attribute of the created
      instance of the corresponding "primitive" predefined class (in runtime all parameters are references to instances)
attrSpec ::= attrName | elName.attrName | navig.attrName /* attribute of the current or specified class, including temporary attribute
navig ::= elName.roleName{.roleName}* /*role name at the far end from the class, must go to the "1-end" of the association, when in attrSpec, for
      /* set expressions no restrictions
      /* elName.attrName is also called reference in interpreter – to distinguish it from a local attribute
      /* elName can be self, an element name from the current rule – or referenceName according to Syntax rule 12
intElemTerm ::= elemTerm | (intExpr) | size(stringExpr) /* integer elemTerm only (integer constant , integer attribute or integer parameter)
factor ::= intElemTerm | factor * intElemTerm /* integer only
intExpr ::= factor | intExpr (+| -) factor /* integer only
stringExpr ::= stringfactor | stringExpr + stringfactor | /* + used instead of OCL concat
stringfactor ::= elemTerm | substrng(stringExpr, intExpr [, intExpr]) | toUpper(stringExpr) | toLower(stringExpr) /* for substrng the first and
      /* last character positions are specified, if the last position is omitted, then till the end of the string, here elemTerm – of string type only
      | toString(intExpr) /* standard integer representation as a string (omitted in OCL 2.0 ??)
boolSimpleExpr ::= true | false | attrSpec /* the specified attribute must have Boolean type, boolSimpleExpr is for assignments
enumSimpleExpr := enumLiteral | attrSpec /* enumLiteral without quotes, attrSpec must have the relevant type
constant ::= integerConst | stringConst
integerConst ::= [-]unsignedInteger
stringConst ::= 'string' | "string" /* single or double quotes – OCL uses single, but SQL and OOP languages double ones. String cannot contain ' or "

OCLboolExpr ::= boolFactor | OCLboolExpr or boolFactor /* this boolean expression is for constraints
boolFactor ::= boolTerm | boolFactor and boolTerm
boolTerm ::= relation | setRelation | (OCLboolExpr) | not boolTerm

```

```

relation ::= attrName = simpleExpr | attrName <> simpleExpr | attrName < simpleExpr | attrName > simpleExpr | attrName <= simpleExpr |
          attrName >= simpleExpr          /*<, <=, > and >= for integers only, = and <> for enums and booleans also, both types must be equal
setRelation ::= navig->size() (=|<>|<|>) integerConst | navig->isEmpty() | navig->notEmpty() /* set size relations
              navig (=|<>)navig /* a proper set equality/nonequality
              attrSpec->isEmpty() | attrSpec->notEmpty()
              /* navig may produce a set (in * direction), this kind may be used in set relations
              /* attrSpec with isEmpty can be used only for attributes with cardinality 0..1 in this release, this is not a proper set relation, but just
              /* an equivalent of SQL IS NULL for a column (* cardinality in the next release)

```

```

cardConstraint ::= NOT | OPT /* for MOLA elements (also NOT-regions, and association links)

```

AssocLink must correspond to an association in the metamodel between the relevant classes

```

startRole ::= name | #name / # is used if the corresponding association is mapping, only one of the roles may be present, if it is unique
endRole ::= name | #name / # is used if the corresponding association is mapping

```

```

/* assocType describes the role/action of the link, normal means just match

```

```

startCard ::= assocCard /* startPrompt, endPrompt, isDirected must be ignored for links, cardinalities not used for links in this release
endCard ::= assocCard

```

```

assocCard ::= 1 | 0..1 | * | 1..*

```

```

constraintNoteText ::= simpleExtOCLExpr /* reserved for separate textual OCL constraints on pattern as a whole

```

CallStatement is used to invoke a MOLA subprogram

```

callStatement ::= diagramReference ( [parameters] ) /* diagram reference is visible as the diagram name, but is is a reference in MOLA repository

```

```

parameters ::= actualParameter{,actualParameter}*

```

```

actualParameter ::= referenceName:className | stringExpr | intExpr /* The compiler converts the expression to an assignment to the value
                    attribute of the created instance of the "primitive" class, and places reference to this instance in the call, as for object params, afterwards a delete action
                    for this instance is generated.

```

The type of the actual parameter must coincide with the equally positioned formal parameter (see more in rule 11)

```

flowText ::= [ ELSE ] /* See more in rules 9, 14

```

Parameter.references – may point to a "normal" user class, or one of the predefined "primitive" classes – String, Integer, Boolean, which all have one attribute value – of the corresponding type. For ClassElement references can point only to a user class.

4. Syntax rules (constraints) and Semantics comments:

1. MOLA diagram (MOLADomainDiag) may contain directly all container kinds: Start (1..1), End (1..*), **TextStatement**, CallStatement, Rule, ForeachLoop, **WhileLoop** (all 0..*) and also Parameters – containment is via containsD association, parameters are always contained directly in a diagram
2. Start, End, CallStatement, Rule may contain no other containers (**Rule may contain Not_region, which in turn may be nested**) /* CallStatement not inside Rule – it is a separate container !!
3. ForeachLoop and **WhileLoop** may contain 0..* CallStatement, Rule, ForeachLoop, WhileLoop – but there must be 1..* contained elements in totality (including the loop head); the containment is via includes association
4. Rule which contains 1..* loop variable (ClassElement with elType = LoopVariable) is defined as having the (implicit) subtype LoopHead (in this release we allow only **one** loop variable per loop head)
5. ForeachLoop must contain just 1 LoopHead, **WhileLoop – 1..* LoopHeads**; LoopHead may be the only element of a loop. If there are more elements per loop, LoopHead may have no incoming Flow (a Flow with to association to it)
6. Rule must contain 1..* ClassElements via includes association (LoopHead subtype - currently just 1 LoopVariable and 0..* other ClassElements) , no other elements may be contained (note that AssocLink instances are not formally included in a container in the MOLA metamodel)
7. ClassElement must have one references link to a Class in the metamodel, elName property may be empty. If several ClassElements reference the same Class, they must be distinguishable by elName. Other properties of ClassElement are optional, isCopy currently is ignored. Constraint and assignment must conform to the specified here textual syntax. The d_is_in link shows the containment in the appropriate MOLADomainDiag
8. AssocLink must correspond to a metamodel Association, which is between the Classes referenced by ClassElements being the endpoints of the link. The correspondence is recognized by startRole or endRole properties (or both of them, but only one is mandatory to be specified), the role must be attached to the appropriate end of the link – the same one as in the Association. The d_is_in link shows the containment in the appropriate MOLADomainDiag, but the includedIn link to the nearest container is not set in the editor – this link must be inferred from the corresponding link for endpoints – ClassElements (must be equal for both, a link cannot cross rule boundaries!).
9. Currently a container (except END – there none and Rule - there may be two) must have just 0..1 outgoing Flow (associated via source link), a Rule can have one non-marked outgoing Flow and one marked ELSE (only one of them may also be present) . Start and LoopHead may have no incoming flows. Flows from Start (in a diagram) or from LoopHead (in a loop) form one continuous path, if there are no ELSE Flows present. This path at diagram level must terminate at End, inside a loop any container may be the last one – a special case is just a single loop head. Several paths (resulting from both normal and ELSE flows present in some rules) can merge – it is allowed for several Flows to enter a container. A flow may reach the directly containing loop border from inside – it is an explicit indication that the next iteration must be started (“continue” in conventional programming). This construct is just a “syntactic sugar” – if there is no outgoing flow at all (with the relevant mark, see also 14), the next iteration is started anyway. Currently a flow cannot cross the loop border (**later an “exit” construct will also be allowed**).
10. Mola diagram, which is not main, may contain 0..* Parameters (no Parameters for main). Parameter has an elName (starting with @ character) and it references a Class. Primitive (String, Integer, Boolean) parameters are permitted (actually they are defined as a reference to the predefined primitive String class, similarly also for Integer, Boolean)
11. CallStatement referencing a MOLADomainDiag via diagram link must have an actual parameter list (a textual one, in the given order) matching to the (formal) parameters for this diagram – for which the order is defined by paramNumber (starting from 1). The match is performed according to the

order, the referenced **Classes** must coincide for the actual and formal parameter (the Class of the actual parameter may also be a subclass of that for formal parameter), but the names may differ. Primitive-typed parameters can have types String, Integer, Boolean. A parameter in the list (**actualParameter**) must be a valid reference (see 12) to a **ClassElement** (or a formal parameter in turn) – or a String expression for String parameters (Integer expression for Integer). All object parameters actually are in-out – as references, since any actions based on them affect the attributes and links of the real referenced instance. String (integer) parameters are by value (in) – an expression may be supplied as the actual parameter. But all parameters are kept on runtime stack, to support **recursive calls** (with several copies of a subprogram active simultaneously).

12. A reference **ClassElement** (one which has **@name** as **eName**) must reference (i.e., have the same **eName** – but with **@** removed and **Class** reference) as a **ClassElement** in a loop head of the containing loop or nesting loop, or in a rule (non-loop, only via its unmarked flow) preceding the given container via control flow or, finally, it must be a (formal) parameter (with **@** included) of the diagram. More formally (first consider the case where there are no ELSE flows present), if we are in a **Rule** (which may be a **LoopHead**) and have a reference element, we must resolve it by a "normal" **ClassElement** in a rule preceding (via control flows) the given one in the current containing loop, or (if not found) go a level up (to the containing loop) and again "go upstream" the flows and try to find a rule containing the appropriate **ClassElement**, etc. In "going upstream", we should never "step down" (i.e., to a loop head nested in a loop at the current level), only a rule – a "normal" one or loop head at the current level must be searched. An element defined in a **Rule** may be referenced only in its non-marked continuation, also not after path merge. Now the branch/merge situation in a more general setting. If we have only one path back (via "reversed" flows), any definition may be used, unless it is reached directly via reversed ELSE flow. If there is more than one path back, only definitions from fragments common to all paths are used for resolution. Or in other words, when there are several paths back, they must be split into fragments within one container – loop or diagram. All these fragments have a common point – the loop head for a loop (or start for a diagram). An element may be used for reference resolution, if it is present in all fragments, and in no fragment it reached directly via ELSE flow. In the next level up, again the fragments must be found (if several). A completely alternative possible match is a parameter of the given MOLA (sub)program – its name already contains **@**, and the reference must be resolved to the corresponding actual parameter in the call statement (which in turn may be a parameter of that program). The resolution of a reference pointing to a parameter is completed, in fact, only during runtime, then the corresponding actual parameter bound to this parameter is found. The same diagram may be invoked by several **CallStatements**, and recursive calls are not prohibited (some examples use them), so in principle the call semantics is similar to programming languages. Though the described resolving algorithm actually defines some search order, in this release we will not explicitly support some name visibility policy – it is the responsibility of MOLA programmer to make references unambiguous.
13. Target metaclass attributes, which have cardinality 1 ("mandatory"), must be assigned values in some MOLA assignment. If the value is not set, they get the value NULL ("undefined"). Attributes with the cardinality 0..1 simply may be absent. A similar situation must be in a correct source model (mandatory values must be present).
14. Rule which is not a **LoopHead** (but follows it via control flows) may have a pattern of its own - in addition to references to **LoopHead** elements, the semantics is – if the pattern matches for a set of relevant instances, the rule is executed once, if not – the rule is not executed. The **unmarked** control flow (if any) following this rule is not continued in the "not case" (i.e., the next iteration or process end is assumed, if the ELSE flow is not present). If the flow marked **ELSE** is present, it is traversed in this case. Thus a rule plays the role of a graphic if-then-else in MOLA. If several instance sets match, arbitrary ("the first") one is taken – in fact, this is a semantic error in MOLA program.
15. **cardConstraint NOT** on a **ClassElement** is permitted, **NOT on a region or link is delayed**. No graph topology constraints are present (except that two NOT-elements **may not be** linked). The meaning is – simply try to find an instance set (all for loop, one for rule) for the positive class elements of the pattern (satisfying attribute constraints and including the used references), where there are no instances of NOT-elements linked to positive ones by specified links (in the current instance space).

16. Precise semantics of FOREACH loop is the following. The pattern (normal elements and links) in the loop head is matched against the current instance set in the repository (model), with the requirement that all constraints evaluate to true and NOT elements does not match. Only for the loop variable all matching instances are registered, for other elements any one valid matching instance is taken (the **Exists** semantics!). Thus the match for the first iteration is found. When the flow for this iteration completes, the match **is reevaluated**, and if there is an instance of the loop variable **not already used** for the iteration, a new iteration is started. And so on, until there are no more unused instances of the loop variable. Thus, for example, the instance set for the loop variable may be replenished during the iteration (a danger of infinite loop!), a “For-While” semantics may be emulated (by including the continuation condition in one of the constraints).
17. Delete element semantics – if a class instance is deleted in a Delete-element, all association instances linked to this instance are deleted also. Delete link – just delete the given link. It is forbidden to delete the loop variable instance in the loop head – a separate rule must be added to the loop for this.