MOLA syntax

0. Introduction

This document describes the precise (expected) syntax of MOLA 2 and some elements of semantics. MOLA 2 is the language implemented via compiler to L3 (and then to L0). 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** (currently) **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. Yellow highlighted syntax elements - not to be implemented in the first version of MOLA 2 compiler (via Lx languages). New constructs in MOLA 2 (with respect to MOLA 1) are highlighted green.

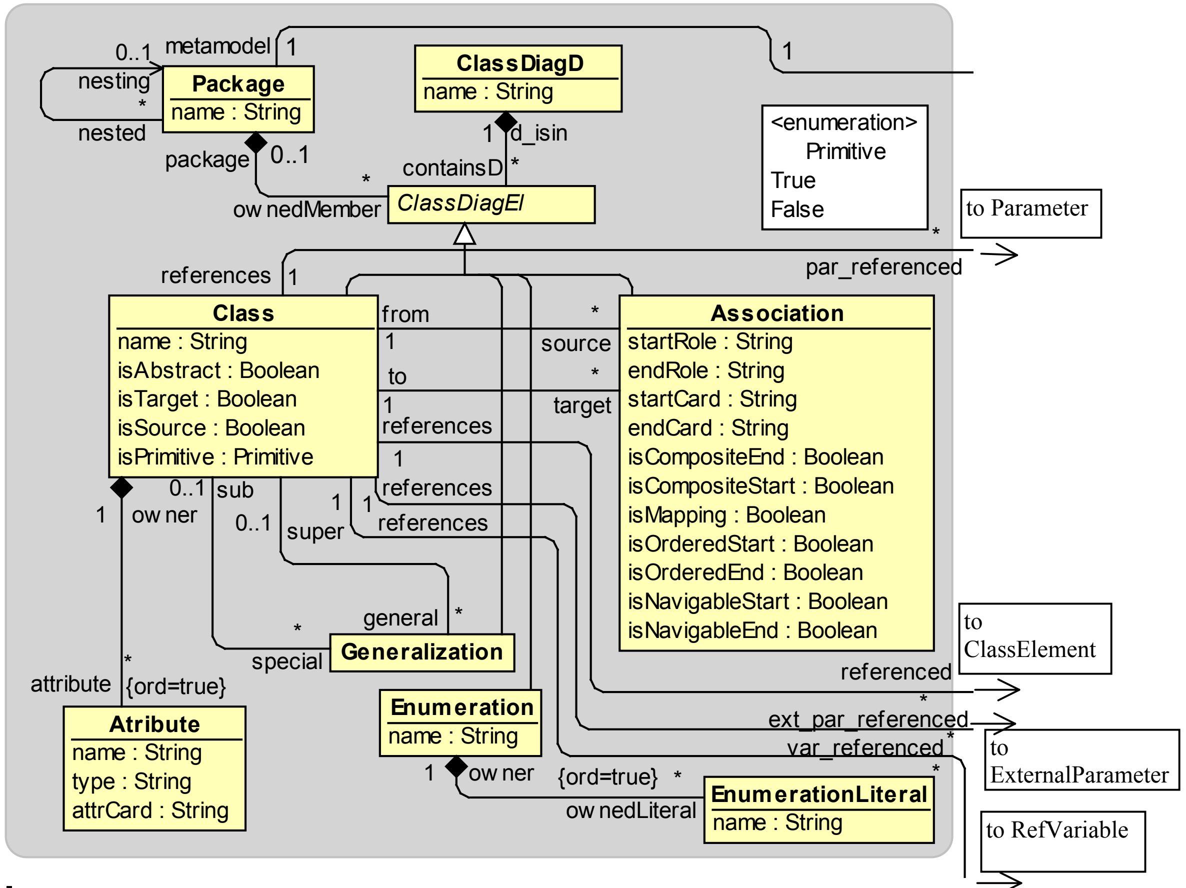
1. Lexics

Identifiers (names) in MOLA can contain letters, digits, underscore ("_"), but no other special characters (and no blancs). 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 (elementary) 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.

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 /* # is no more supported for mapping association role names in MOLA2 (it had no special semantics in MOLA 1 anyway)
endRole ::= name /* the isMapping metattribute also has no special semantics for MOLA transformations (some use for import/axport)
/* isComposite, isNavigable is not used directly in MOLA transformations, but is used for import/export definition (isComposite is used also for
/* model **repository definition**, where it has the **cascade delete** semantics – detail instances are deleted automatically, when the master is deleted
/* by a MOLA action, the transitive closure on composition is applied in that case)
/* isOrdered has the standard MOF semantics, it must be used for associations on which after is based

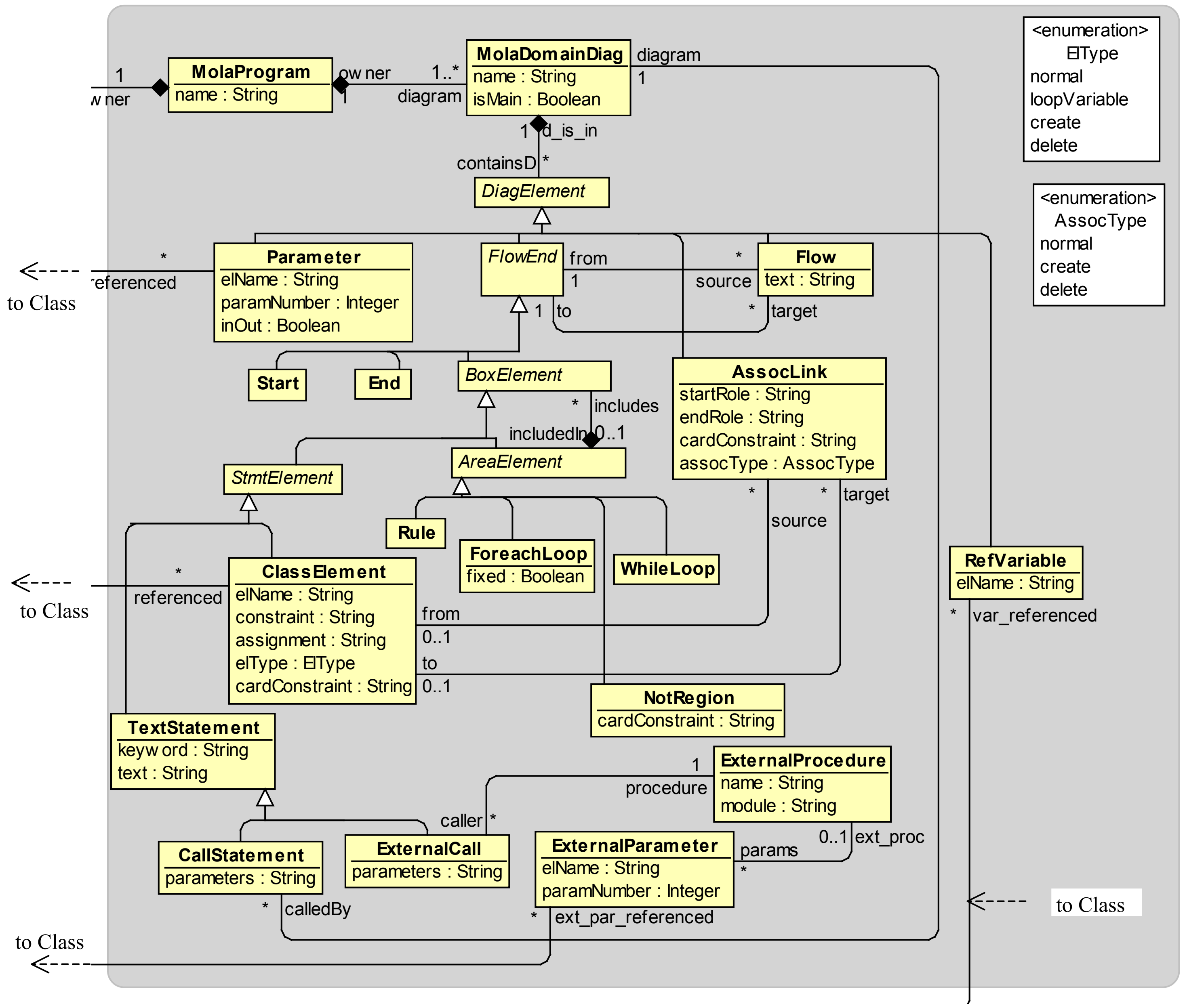
Attribute is the standard UML attribute (but not navigable association end!)
attrCard ::= **1** | **0..1** | ***** | **1..*** /* default is 1, * and 1..* currently are **not** supported for attributes
type ::= **String** | **Integer** | **Boolean** | enumerationName /* enumeration must be defined
attributeName ::= name | tempAttrName
tempAttrName ::= _name /* it is just an informal indication, formally any name may start with "_", there are no semantic consequences for temps

/* 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 metaatribute 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
/* Currently there is a runtime repository imposed restriction that only **one attribute per class** may have the **given enumeration** as its type

/* Standard inheritance semantics is assumed – subclasses inherit attributes and associations from superclasses,
/* currently **only the single inheritance is permitted !!!**
/* an abstract superclass may be the class referenced in MOLA element (see MOLA diagram section), then actually **instances of each subclass match to this element**
/* Additional metaattributes for association (2* isNavigable) are used for XMI export/import configuration,
/* but currently will not be directly used in transformations.
/* Metaclass *Package* (with its associations) can be used in MOLA as a namespace (to make Classes with equal names unique in references). If there is no
/* class name uniqueness problem, packages may be absent.
/* Metamodel definition is used in MOLA 2 for the automatic generation of runtime repository metamodel (only the standard (EMOF) features are used
/* for this), if the runtime metamodel is obtained by another means, it must be consistent to the definition in MOLA

3. MOLA diagram

Metamodel of the **MOLA** diagram:



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, Parameter, ExternalCall, RefVariable
"Element" classes (which must be part of a `Rule`) : ClassElement, AssocLink (also NOT_region though it is a conteiner)
Area elements – Rule and WhileLoop have no text, context (inclusion) rules for them are described in section 4., ForeachLoop has the `fixed` attribute (with default = true)

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 (to the correct
                                      /* one, if necessary, selected using its package reference)
            self                   /* self is used in navigation expressions – for navigation from this element, and in object assignments/relations
referenceName ::= @name |  /* element from another rule (see comments below (rule 12), where the referenced element must be located) or parameter
                  @refVariableName  |  /* reference variable (referencing  a class, i.e., "pointer variable" or a primitive typed variable)
                  @parameterName
        /* 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
             | objectAssignInClassEl /* (object assignment doesn't combine with attribute assignments, for reference variables and in-out parameters
                  /* only) – as an alternative form to text statement
assign ::= attrName := (simpleExpr | NULL )/* type must be compatible, attrName – an attribute of this element class, may be temporary – with "_"
                                /* prefix, NULL  only for [0..1] attributes (currently of types Integer or Boolean)
simpleExpr ::= intExpr | stringExpr | boolSimpleExpr | enumSimpleExpr

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).
      @refVariableName  /* having an elementary type, i.e., an "elementary variable"

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 – 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)
                   toInteger(stringExpr) | stringIndex /* an index function; toInteger returns –1 if string is invalid
```

```
factor ::= intElemTerm  | factor * intElemTerm     /* integer only
intExpr ::= factor | intExpr (+| -) factor       /* integer only
stringExpr ::= stringfactor | stringExpr + stringfactor  |       /*  + used instead of OCL concat
stringfactor ::= elemTerm | substring(stringExpr, intExpr [, intExpr]) | toUpper(stringExpr) | toLower(stringExpr)   /* for substring 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)  | toString(boolSimpleExpr)   /* standard integer representation as a string (omitted in OCL 2.0 ??)
             | toString(enumSimpleExpr)
boolSimpleExpr ::= true | false | attrSpec  | toBool(stringExpr)  /* the specified attribute must have Boolean type, boolSimpleExpr is for
                        /* assignments
enumSimpleExpr := enumLiteral | attrSpec  /* enumLiteral without quotes or type prefix, attrSpec must have the relevant type
             | toEnum(stringExpr)        /* cannot be used as part of a more complicated expression, i.e., may be used in assignment to enum attribute
                        /* or variable, or in comparison to enum variable/attribute or constant (because the type is found from the context)
constant ::= integerConst | stringConst    /* boolean and enum constants have been defined here directly as parts of a simple expression
integerConst ::= [-]unsignedInteger  /* no - - or + - is permitted!
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 ::= boolFactor | boolFactor and boolTerm
boolTerm ::= relation | setRelation | (OCLboolExpr) | not boolTerm | objectRelation | stringIndexRelation | TypeRelation
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
                        /* simpleExpr may contain
setRelation ::= navig->size() (=|<>|<|>) integerConst | navig->isEmpty() | navig->notEmpty() |     /* set size relations
             navig (=|<>|<=|>=)navig  |   /* a proper set equality/nonequality/inclusion
             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

objectRelation ::= self (=|<>) referenceName /* this relation can be used in normal elements, constant references or reference variables/parameters,
                              /* the reference also may be of any kind, but both must have compatible classes (including subclasses) –
                              /* the semantics is instance id equality at the given moment
                self (=|<>)NULL  /* for reference variables and parameters only
stringIndexRelation ::= indexOf (constrStringExpr, constrstringExpr) (=|<>|<=|>=|>|<)IntExpr | /* all expressions (string and integer
                             /* on the lefthand side may contain only simple arguments - see below, RHS may be any
                        substring(constrStringExpr, constrIntExpr[,constrIntExpr]) (=|<>)stringExpr  /* no nested substring in
                           /* the lefthandside expression, only one level permitted
```

constrStringExpr ::= attrSpec | constant /* of type String – may be replaced by other syntax element
substring(constrStringExpr, constrIntExpr[,constrIntExpr]**)**
constrIntExpr ::= attrSpec | constant /* of type Integer

TypeRelation ::= **self.isTypeOf(**typespec**)** /* true if the type of this instance is exactly that defined by typespec (it should be subclass of the class used
/* in this element definition

typespec ::= className | packageName**::**className /* if the class is in a package, the package prefix is required

cardConstraint ::= **NOT** | **OPT** /* for MOLA elements (also NOT-regions, and association links)
ordered | **reverse_ordered** |/* **only** on a link connected to loop variable (the **ordered end** of the corresponding association must
/* be connected to the loop variable class in MM, and there must be **only one** such link connected to the loop variable),
/* the loop must be traversed according to ordering - forwards or backwards
clone | **after** /* clone – instance cloning relation, classes must be the same, one is "create" ; after – insertion order for several links
/* (ordered, of the same type), in both cases this is a "pseudolink" between class elements, no roles should be specified,
/* there must be type=**normal**, for after : isdirected=true

TextStatement is an independent construct, used for processing/analyzing variables and parameters
textStatement ::= {objectAssign | varAssign}* /* assignments are placed in the **text** field, one per line (as in the assignment part of a class element),
/* newLine is used as a separator. A text statement may contain assignments and/or constraint
[elementaryConstraint] /* constraint in the **keyword** field (above the separator), visible in editor as Constraint:
objectAssign ::= pointer **:=** (referenceName | typecast | **NULL)** /* assigned pointer must be of the same class or subclass
/* , reference name may be of any kind
objectAssignInClassEl ::= **self :=** (referenceName | typecast | **NULL)** /* for use in class element only
pointer::= @refVariableName | @ParameterName /* only an inout-parameter will hold the assigned value after the return from procedure, but locally
/* any parameter may be assigned a new value
varAssign::= @varname **:=** simpleExpr /* type must be compatible, varname must be of a primitive type (or primitive parameter)

elementaryConstraint::= @varname **(=|<>|<|>|<=|>=)**simpleExpr | /* expr of the same type as variable, < , > for integers only
pointer **(=|<>)(**referenceName | **NULL**) /*

stringIndex ::= **indexOf (**stringExpr, stringExpr**)** /* the only index function. If the first string not found in the second, returns 0 (normal count from 1)

AssocLink must correspond to an association in the metamodel between the relevant classes
startRole ::= name / * no more # for mapping associations, only one of the roles may be present (in the pattern navigation direction, see more in
endRole ::= name /* the section on pattern annotations
/* 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..*

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 | stringExpr | intExpr | boolSimpleExpr /* The compiler converts the expression to an assignment to the **value**
| enumSimpleExpr | typecast
/* 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)
/* for in-out formal parameters only reference variables and in-out parameters may be supplied
/* for **inOut** property in **Parameter** definition the default is **false**, therefore parameters in old models with that property missing must be
/* treated as non-inOut
/* for MOLA 2 no class name is added to the reference

typecast ::= typespec**(**referenceName**)**
ExternalCall is used to invoke an external procedure
ExternalCall ::= extProcedureReference **(** [parameters] **)** /* only class-typed parameters permitted (including reference variables)
ExternalProcedure + its ExternalParameters defined as a set (visible in tree) /* types must match in call and definition – classes only, subtyping is
/* permitted, parameter definition order is defined by the natural order in the model. The `module` attribute of ExternalProcedure names an
/* "implementation unit" (dll, jar etc.), where the external procedure is located (as an entry point)

Others
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.

ForeachLoop has a Boolean attribute fixed /* it is true by default and mandatory in new models, but missing in old ones, therefore null=true

refVariable definition : @refVariableName and reference to a class ("pointer variable") or MOLA primitive type ("elementary variable")

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

4. Syntax **rules (constraints) and** Semantics **comments:**

1. MOLA diagram (MOLAdomainDiag) may `contain` directly all `container` kinds: Start (1..1), End (1..*), TextStatement, CallStatement, ExternalCall, Rule, ForeachLoop, WhileLoop (all 0..*) and also Parameters and RefVariable definitions – containment is via containsD association, parameters are always contained directly in a diagram (here the term `container` is used not in the sense that it always contains something – e.g., Start contains nothing, but it is a top-level part of a diagram)
2. Start, End, CallStatement, TextStatement, ExternalCall, 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, TextStatement, 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). A special kind of loopheads is for While loops – this rule has **no loop variable**, it is just a rule **with no incoming flow**.
5. ForeachLoop must contain just 1 LoopHead, WhileLoop – 1..* LoopHeads (currently WhileLoop must contain just one loophead); 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 (in Foreach case, 0 in While case) 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. Several ClassElements may reference the same Class, but they must be distinguishable by elName. Other properties of ClassElement are optional. Constraint and assignment must conform to the specified here textual syntax. The d_is_in link shows the containment in the appropriate MOLAdomainDiag. Each ClassElement must have a name unique within a MOLA diagram, or it may be reused in several patterns, but always with the same class. This guarantees that the element name alone (without specifying the class) can be used as a reference, e.g., in actual parameter list, typecast etc. Element name may not be reused within the same rule (except for Regions in future versions) .
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 (see Section 5 on restrictions for use of only one role name). 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, TextStatement - there may be two) must have just 0..1 outgoing Flow (associated via source link), a Rule (or TextStatement) 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 coventional 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. A flow can also cross the loop border - an “exit” construct is also allowed, but a flow cannot enter a loop body from outside. Otherwise, any flow targets in a diagram are permitted (not only those according to structured programming rules), e.g., a while-loop may be emulated by simple backward flows.

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). By default parameters are in, but parameters may also be in-out. The class instance to which such parameter points, may be replaced by another one in the subprogram, and the caller will see the new instance after the call (the actual parameter then must be a reference variable or another parameter). But for all class parameters the attribute modifications of the instance will be visible in the caller.
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 elName) must reference (i.e., have the same elName – but with @ removed and the same Class) a ClassElement in a loop head of the containing loop or nesting loop, or in a rule (non-loop, but only via its unmarked flow) preceding the given container via control flow or, finally, it must be a (formal) parameter or variable of the diagram. To put it short, a reference element **must point to** a certain **instance** of the relevant class at the moment when it is used in a pattern, as a call parameter or attribute qualifier in an expression. There are some formal criteria which help to find out whether a reference will have a value at the given point (or be a "NULL-pointer"). First let us 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 (for ELSE-exit there will be no matched value!). After both paths merge, we cannot be sure which of the paths actually was taken, therefore the reference should not be used (unless additional conditions can guarantee that actually the "if-branch" was taken). An element may be used for reference resolution, if it is matched in all branching and then merging paths, though in different rules (in all cases only the "if-branch" counts). A parameter of the given MOLA (sub)program always may be used as a reference, the reference then is 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. A reference ClassElement may also be based on a (class-typed) reference variable – certainly its value must be set (by assignments) before the use. In general, 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 or is outside a loop) 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 the rule pattern, arbitrary ("the first") one is taken ("pure existence"). In fact, in most cases this is a semantic error in MOLA program, however, an "OCL-like" assertion about the model containing only existential quantifiers can be specified by such a MOLA rule. A text statement containing a constraint (on values of parameters, variables or known element references) also can play the role of if-then-else.
15. cardConstraint **NOT** on a ClassElement is permitted, also NOT on a link or a region. 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). NOT on a link means that the given link is not present between the selected instances.
16. Precise semantics of FOREACH loop is the following. Two kinds of FOREACH loop exist – **fixed** and **non-fixed**. For both kinds 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!). For **fixed** loops the whole iteration set is defined this way. For **non-fixed** ones only 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!), and a "For-While" semantics may be emulated (by including the continuation condition in one of the constraints – don't forget to use the **non-fixed** mode). However, in both cases it is not permitted to use in the pattern constraints (i.e., conditions which govern valid instance selection for the iteration) variables, which change their value during execution – the loop semantics is undefined for such a case. Though the current implementation does not implement true non-fixed loops, there is an exception also from the fixed-semantics – some of the new instances of the relevant class generated during the loop execution may occurr in the loop execution list, try to avoid such a situation. But it is completely valid to delete some of the instances which would later be on the execution list – the loop is not executed for them.
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. Caution – if an instance is deleted (in a loop or rule) then links to (directly) **next matched** pattern elements (see matching rules in section 5) are also lost and these element references are no more valid. Store them in additional pointer variables if you need these references after the delete operation.
18. Reference variables ("class pointers") can be declared in any MOLA program. They gain their values via object assignment (the right hand side may be an object reference, parameter or another reference variable). Reference to the given class or a subclass may be assigned. Reference variables may be used in any kind of patterns (similarly to reference elements – with value already set) and as actual parameters for calls. Elementary typed Reference variables are also permitted.
19. If external procedures (along with their parameters), visible only in the MOLA model tree, are defined, they can be invoked via external call statements. Only class typed parameters can be used (in-only), actual parameters may be of any relevant kind. The way parameters are passed to the corresponding OOP operation, is language dependent and will be described separately (currently C++ is considered)

20. In the FOREACH loop the **ordered / reverse_ordered** option on a pattern link (only one!) to the loop variable prescribes the instances of the relevant class to be traversed in the order imposed by this link (with respect to a fixed instance – reference element at the other end of this link – the "owner"). The corresponding association must be ordered also in MM. The actual order is determined during instance creation (or model import which must be of kind also supporting ordering). Though the option currently is not supported, if the loop variable instance set is based on one ordered (in the metamodel) link and instances have been created in the desired order, the instances are traversed in this order during the loop execution.
21. In the current version `after`-links can link only `create`-elements (with ordering based on `create`-links for one ordered association), `after`-links can form a simple chain only.

5. MOLA annotations for efficient compilation to L0/L3

The first version of MOLA via L0/L3 **requires annotations** in patterns for a correct compilation. In future these annotations will be optional, since in many cases they can be generated automatically (on the basis of metamodel multiplicities etc.). However, when knowledge on actual instance multiplicities is used (a sort of model preconditions) these annotations will be necessary in future too (for performance improvement). Restrictions on patterns in the future will be removed. Annotations are used for elements – to guide the compiler to a correct "start point" for the pattern, anf for links – to guide the pattern traversal.

The following annotation keywords may be used in cardConstraint for **elements**: **single** and **start** . An annotation of this kind may be used **once** per **pattern fragment** (a connected set of pattern elements, only elements and links defining the pattern matter, but not e.g., **create** links). Also, if two parts of a fragment are linked by a "c-link" (see later) or NOT-link only, they are considered to be separate fragments (this property may be used to split up a too complicated pattern fragment – see later).

The **single** keyword means that the pattern match (for this fragment!) must be started with the given element and the first instance found (if any) fits. In other words, a "dynamic singleton situation" is assumed for this class. If no instance is found, the pattern fails. If there is an attribute constraint in this element, the constraint is evaluated on the chosen instance, and if it is false the pattern fails.

The **start** keyword means that the match must be started locally by this element, but several instances may exist and the right one is chosen using a constraint. The constraint includes the attribute-based one in this element and that (recursively) based on outgoing links: both "1-links" and "*-links" (see next). In other words, it is a "**suchthat**" (L1) search for the class, and the "suchthat" condition is induced by the whole fragment.

If the pattern fragment contains a **reference** (of any kind), i.e. a known instance, **no annotation** for this fragment is necessary, the search always starts from this reference (if there is more than one reference, an arbitrary one is chosen as the first, the situation should be avoided whenever possible). Thus a fragment always must have a **unique initial node**, from which the match starts. Hence, each pattern fragment is treated as a **rooted tree**, which is traversed from the root (initial node) in all possible directions during the match (**non-tree** fragments currently are **not supported**, **except** for **c-links** and **NOT-links**, see later).

To sum up, each pattern fragment must contain one "initial node"- reference, `single` or `start`. References may be more than one, if there is a `single` or `start` node, then namely this node is used as initial, but not references. If the fragment contains several references, but no `single` or `start` node, it is recommended to split up the fragment using c-links, e.g., decide from which of the references a natural match should start and mark links going to other references as c-links (these links then mean additional constraints during the match, but other references themselves are treated as separate trivial fragments).

A **link** also may have an annotation "**keyword**" **1** in its cardConstraint . This optional keyword **should** be specified for links, leading from the fragment start element (of any kind) to other pattern elements (or from elements already reached) whenever there is a "**unique continuation**" – actually only one instance is reachable by the given link from the given source instance in valid model data. Then the "first" reachable instance is chosen. The target element of the link may contain an attribute constraint, then if the constraint fails on the chosen instance the whole pattern fails (if the search started from a **single** node) or another instance is chosen for the current search (which was started by a **start** node or a non-1-link). Links annotated this way are called "1-links" in the sequel. If all links in a fragment are 1-links and the fragment starts with a single node (or reference), the match is unique – at first the initial instance is found, then those reachable directly from it and so on, no alternative search is done at all (pure **first** or **first-by-from** in L0 terminology). If anything fails, the pattern fails. 1-links from a **start**-element just serve as constraints during search. But it is not an error to **omit 1** for a link where actually the search is unique – only a slightly more inefficient code will be generated.

A link with **no annotation** is an ordinary MOLA link where no uniqueness assumptions are made. Such a link means a general "**suchthat**" (L1) search (i.e., an iteration until found) for the class at the link destination (away from the initial node), using constraints – both attribute and next link based. Such a link is called "***-link**" in this description. Several consecutive *-links invoke nested suchthat searches.

A link may have also an annotation "**keyword**" **c** in its cardConstraint This link (**c-link**) is meant to connect two nodes of a pattern fragment tree – thus this pattern fragment actually contains a closed **loop**. There may be several c-links per fragment. Another possible situation is that a c-link connects two fragments. The meaning of c-link is "check the presence of a link". If the end nodes of this link are already known or reachable (from the initial node) by 1-links, this means an additional constraint – the chosen instances must have the specified link. If one of the reaching links is a ***-link**, c-link serves as an additional constraint for selection of the right instance. This way in most cases a pattern with a "find an instance having two specified links" condition can be specified in the current MOLA subset. It is chosen in a nondeterministic way, to which of the elements the c-link refers (as a constraint), if it links two elements in two branches, both reachable by *-links. Nearly always the situation can be avoided by marking another link to be a c-link (closer to a known instance, the possible loop in the pattern is broken anyway). It should be noted that the semantics of c-link is quite opposite to that of **NOT-link** – a c-link instance must exist between the given class instances, but NOT-link must not exist (however, both of them in a similar way may be used to **"break loops"** in patterns). To sum up, the c-link feature is sufficient to define the desired match semantics for any practically usable pattern in MOLA.

In a **Foreach loop head** annotations are used in a similar way. A **single** node or reference may start the loop (full iterative) search, and the **loop variable node** must be reachable by **one *-link** from the initial node, this link defines the actual "loop search space". **Start** nodes currently are not supported in loops. In a special case, the loop variable may be also the **initial node** (when there are **no** annotated nodes or references, e.g., the loop consists of just the loop node). There may be more links (1- or *-) from the loop variable, they define additional constraints, which are included in the loop search (**foreach** in L3) condition. *-links invoke nested (until first found) searches. Currently the whole loop head must consist of one fragment. In addition, it is **not** allowed to **delete** the starter node in loop actions.

Currently for links in a fragment the role names must be specified at the "**navigable end**" – away from the initial node (this restriction will later be removed, this is for making the compilation easier – L0/L3 commands require just these roles). For c-links both role names must be specified, for NOT-links – any of the names.

Currently, for **create** links the role name must be specified as the **end role** (in MOLA editor) (or both roles).

Currently **NOT-elements** must be **leaves** in a fragment tree – only one link may connect such a node to the other part of the fragment.