The MOLA Language

Reference Manual

Version 2.0 final

December, 2007

Table of Contents

1 Introduction 5

1.1 Structure of the MOLA reference..... 5

1.2 Naming conventions..... 6

2 Kernel package - metamodel definition facilities in MOLA 7

2.1 Element..... 7

2.2 Comment 8

2.3 NamedElement 8

2.4 PackagableElement..... 9

2.5 Package..... 9

2.6 Model..... 10

2.7 Type..... 10

2.8 Enumeration 11

2.9 EnumerationLiteral..... 11

2.10 PrimitiveType 12

2.11 Class 13

2.12 Generalization 14

2.13 TypedElement 15

2.14 Property 15

2.15 Association 17

3 MOLA package - transformation definition facilities in MOLA 18

3.1 PackagableElement..... 19

3.2 Package..... 20

3.3 Model..... 21

3.4 Procedure..... 21

3.5 MOLAprocedure 22

3.6 ExternalProcedure 23

3.7 ProcedureElement 24

3.8 Annotation 24
3.9 FlowEnd 25
3.10 Flow 25
3.11 FlowEndContainer 26
3.12 Start 27
3.13 End 28
3.14 CallStatement 29
3.15 CallParam 30
3.16 DecisionStatement 31
3.17 Rule 32
3.18 TextStatement 33
3.19 Loop 34
3.20 ForeachLoop 35
3.21 WhileLoop 37
3.22 ReferencableElement 38
3.23 Parameter 39
3.24 Variable 41
3.25 ClassElementDef 42
3.26 ClassElement 43
3.27 Link 46
3.28 AssocLink 47
3.29 Assignment 49
4 Expressions 51
4.1 Base elements 51
4.2 Pointer expression 52
4.3 Set expression 52
4.4 Enumeration expression 52
4.5 Integer expression 52

4.6 String expression 53

4.7 Simple boolean expression 54

4.8 Simple expressions summary 55

4.9 Simple constraints 55

4.10 Constraint expression 56

5 Additional remarks on syntax and semantics 57

5.1 Pattern semantics 57

5.2 Action part of the rule 60

1 Introduction

This paper describes precise abstract syntax of MOLA and some elements of the semantics. Since MOLA is graphical transformation language which has some textual elements, the abstract syntax for graphical elements is provided via metamodel. The BNF is provided for textual elements.

A transformation in MOLA consists of *a metamodel* (set of class diagrams) and *a set of MOLA procedures* (diagrams similar to activity diagrams)**.** The metamodel describes a model being transformed by the MOLA transformation. The algorithm of the transformation is described by MOLA procedures. Since the metamodel and MOLA procedures are graphical diagrams which contain some textual elements, they have been described using the *MOLA metamodel***.** The MOLA metamodel consists of two packages - `Kernel` and `MOLA`. They describe means of defining a metamodel and MOLA procedures respectively. (Note, that here and further in this document the term *metamodel* without additional qualifiers is used to denote the part of MOLA transformation)

1.1 Structure of the MOLA reference.

At first common issues of the MOLA language are discussed. Then descriptions of `Kernel` and `MOLA` packages follow.

The description of `Kernel` and `MOLA` packages begins with a brief overview and class diagram of the package. The description of the `MOLA` package includes also description of data types and expression syntax used in MOLA. The class diagram of the `MOLA` package has been divided into several diagrams to enhance the readability. The precise definition of `MOLA` package abstract syntax is given by the complete class diagram.

A description of each metaclass follows.

The description has several parts. The first part - a *brief overview* of the metaclass describes the purpose and usage of MOLA elements specified by it.

The second part is the abstract syntax of the metaclass. That includes the following:

- *generalization*: list of direct superclasses. Actually, there is no multiple generalization in the MOLA metamodel. Thus there is only one superclass in the list.

- *specialization*: list of direct subclasses.

- *attributes*: list of attributes owned by this metaclass and brief description of each of them. If the possible values of attributes are described by some grammar, the BNF is presented.

- *associations*: list of associations connected to the metaclass.

Lists of attributes and associations include also attributes and associations that do not belong to the metaclass directly, but are owned by its superclasses. They are called *inherited* in MOLA reference and marked as *[inh].*

This reference includes also compiler-related attributes (a kind of compiler ``directives''). These attributes are marked *[cd].*

- *constraints*: general constraints on MOLA elements specified by this metaclass. If the constraint is related to attribute or association, then this constraint is included in brief description of attribute or association respectively.

- *notation*: example of the notation of the MOLA element specified by this metaclass. That includes the graphical notation of the MOLA element. It can be a diagram, a graphical or textual diagram element or a project tree node. In fact, this includes an example of the visual representation of the MOLA element specified by this metaclass.

If some features of the metaclass are absent (e.g. no attributes or constraints), then they are not mentioned in the description.

The third part of the description is execution *semantics* of the MOLA elements specified by the metaclass.

The description of metaclasses is followed by a section on MOLA expressions and a section on general issues of semantics and syntax of MOLA.

1.2 Naming conventions

Most of elements in MOLA have *identifiers* (names). An identifier can contain letters ('a'-'z', 'A'-'Z'), digits (0-9) and underscore ('_'). No other symbols are allowed. The identifier must start with a letter or underscore. There are some restrictions on naming of several elements of MOLA, which will be explained in sections describing them.

```
<name>::=<letter>(<letter>|<digit>)*

<digit>::='0'|'1'|…|'9'

<letter>::='a'|'b'|…|'z'|'A'|'B'|…|'Z'|'_'
```

2 Kernel package - metamodel definition facilities in MOLA

The `Kernel` package (see Fig. 1) describes means of the metamodel definition in MOLA. The metamodel is defined using the metamodelling language similar to the EMOF definition. This language is called MOLA MOF. In fact, MOLA MOF and EMOF define the same abstract syntax for a metamodel definition language. The main structure of MOLA MOF does not differ from EMOF. A great number of metamodels can be built, which all describe the same set of models. MOLA MOF and EMOF describe the same model (metamodel definition language), but their own metamodels slightly differ.

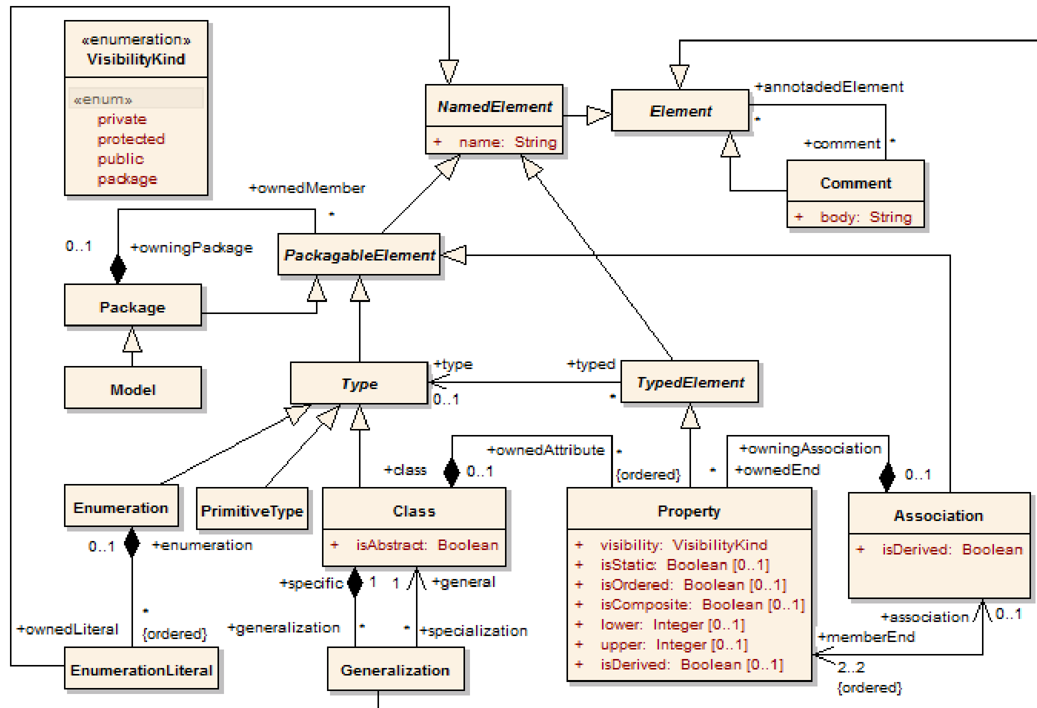

Fig. 1. `Kernel` package

2.1 Element

Brief overview. `Element` is an abstract metaclass with no superclass. It is used as common superclass for all metaclasses in the MOLA metamodel.

Specializations

There are subclasses `Comment`, `NamedElement`, `Generalization`, `MOLA::PackageableElement`, `MOLA::ProcedureElement`.

Associations.

`comment[*]` - comments on this element.

2.2 Comment

Brief overview. `Comment` is an element of a metamodel or MOLA procedure, which provides additional informal information on some elements of a MOLA transformation. It adds no semantic information to the annotated elements, but it can be useful for the transformation writer.

Generalizations

It is specialized from `Element`

Attributes

`body:String` - specifies the text of this comment

Associations

[inh] `comment[*]` - comments on this comment.

`annotatedElement[*]` - elements that is commented by this comment.

Notation. A comment is shown as a rectangle with the upper right corner bent (see Fig. 2). The rectangle contains the body of the comment. The connection to each annotated element is shown by a separate dashed line.

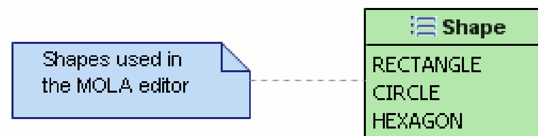

Fig. 2. Comment and enumeration notation

2.3 NamedElement

Brief overview. `NamedElement` is an abstract metaclass which represents metamodel elements which have name.

Generalizations

It is specialized from `Element`.

Specializations

There are subclasses `PackagableElement`, `TypedElement`, `EnumerationLiteral`.

Attributes

`name:String` - the name of the named element.

`<named-element-name>::=<name>`

Associations

[inh] `comment[*]` - comments on this named element.

2.4 PackagableElement

Brief overview. `PackagableElement` is an abstract metaclass which represents metamodel elements which may be grouped into packages. It is allowed to group only packageable elements. If the element is grouped into package then it is owned by that package. Thus the element may be only in one package.

Generalizations

It is specialized from `NamedElement.`

Specializations

There are subclasses `Package`, `Type, Association`.

Attributes

[inh] `name:String` - the name of the packageable element. The name must be unique within the elements grouped into the same package. The name is used to reference this element within the package. To reference a packageable element outside the package the full qualified name must be used. The full qualified name is obtained by prefixing the name with full qualified name of the owning package and '::' symbol.

```
<full-packageable-element-name>::=[<full-owningpackage-name>'::']<named-element-name>
```

Associations

[inh] `comment[*]` - comments on this packageable element.

`owningPackage [0..1]` - a package that owns this packageable element.

Notation. Packageable element may be shown in the project tree as a node, which has an icon and a name.

2.5 Package

Brief overview. `Package` is an element of the metamodel that is used to group packageable elements. Grouped elements are owned by the grouping package. The package may be owned by other package.

Generalizations

It is specialized from `PackagableElement.`

Specializations

There is a subclasse `Model`.

Attributes

[inh] `name:String` - the name of the package.

Associations

[inh] `comment[*]` - comments on this package.

[inh] `owningPackage [0..1]` - a package that owns this package.

`ownedMember`[*] - packageable elements that are owned by this package. The package may not contain itself directly or indirectly.

Notation. Package may be shown in the project tree as a node.

2.6 Model

Brief overview. Model is special case of package - the root package - that means the package without owning package. There is exactly one model in a MOLA transformation.

Generalizations

It is specialized from `Package`.

Attributes

[inh] `name:String` - the name of the model.

Associations

[inh] `comment[*]` - comments on this model.

[inh] `owningPackage [0..1]` - package that owns this model. A model has no owning package.

[inh] `ownedMember`[*] - packageable elements that are owned by this model.

`molaModel`[0..1] - MOLA model that conforms to this model (a metamodel)

Notation. Model may be shown in the project tree as a node, which has a special icon and a name.

2.7 Type

Brief overview. `Type` is an abstract metaclass that serves as a constraint on the range of the values represented by typed element of a metamodel and referenceable element of a MOLA procedure.

Generalizations

It is specialized from `PackagableElement`.

Specializations

There are subclasses `Enumeration, PrimitiveType, Class`.

Attributes

[inh] `name:String` - the name of the type.

<type-name>::=<enumeration-name>|<primitive-type-name>|<class-name>

Associations

[inh] comment[*] - comments on this type.

[inh] owningPackage [0..1] - a package that owns this type.

Typed[*] - typed elements constrained by this type.

molaElem[*] - referenceable elements of a MOLA procedure constrained by this type.

2.8 Enumeration

Brief overview. Enumeration is a kind of type, whose instances may be any of a number of user-defined enumeration literals.

Generalizations

It is specialized from Type.

Attributes

[inh] name:String - the name of the enumeration.

<enumeration-name>::=<name>

Associations

[inh] comment[*] - comments on this enumeration.

[inh] owningPackage [0..1] - a package that owns this enumeration.

[inh] Typed[*] - typed elements, actually, class attributes constrained by this enumeration.

[inh] molaElem[*] - referenceable elements, actually, elementary variables of a MOLA procedure constrained by this enumeration.

ownedLiteral[*] - ordered set of literals owned by this enumeration.

Notation. An enumeration is shown as a rectangle which is divided by horizontal line (see Fig. 2). The upper compartment contains the name label and a special icon. The lower compartment contains the list of enumeration literals owned by this enumeration. An enumeration may be displayed in the project tree. The enumeration literals may be shown as child nodes of the enumeration node.

2.9 EnumerationLiteral

Brief overview. EnumerationLiteral is a user-defined data value for an enumeration.

Generalizations

It is specialized from NamedElement.

Attributes

[inh] name:String - the name of the enumeration literal. The name must be unique within the owning enumeration.

```
<enumeration-literal-name>::=<name>
```

Associations

[inh] comment[*] - comments on this enumeration literal.

enumeration[0..1] - an enumeration that owns this enumeration literal.

Notation. An enumeration literal is typically shown as a name, one to a line, in the lower compartment of the enumeration symbol (see Fig. 2). An enumeration literal may be shown as child node of the enumeration node in the project tree.

Semantics. Enumeration literals may be used as enumeration constants. Note that enumeration constants are without the prefixed enumeration name. Mainly they are used to set enumeration-typed attributes or elementary variables. Thus, the precise enumeration literal type can be obtained from the context of the enumeration constant usage (attribute type, variable type, etc. …).

```
<enumeration-constant>::=<enumeration-literal-name>
```

2.10 PrimitiveType

Brief overview. PrimitiveType defines a predefined data type, without any relevant substructure (i.e., it has no parts). There are three predefined primitive types in MOLA: String, Integer and Boolean. There are no other primitive types and there is no way to define them in MOLA.

Generalizations

It is specialized from Type.

Attributes

[inh] name:String - the name of the primitive type.

```
<primitive-type-name>::='String'|'Integer'|'Boolean'
```

Associations

[inh] comment[*] - comments on this primitive type.

[inh] owningPackage [0..1] - a package that owns this primitive type. A primitive type is owned by the model.

[inh] Typed[*] - attributes constrained by this primitive type.

[inh] molaElem[*] - referenceable elements, actually, elementary variables of a MOLA procedure constrained by this primitive type.

Semantics. Primitive types are predefined data types in MOLA. Values of the String type are sequences of characters. The set of characters allowed in MOLA is unspecified. String constants are specified using the value within single or double quotes. Values of the Integer type are whole numbers. There is no special notation for integer constants - a constant is specified using the value. Values of the Boolean type are *true* and *false*. The *true* and *false* keywords serve also as Boolean constants.

```
<string-constant>::=’’’(<any-character>)*’’’ | ’”’(<any-
character>)*’”’
```

```
<boolean-constant>::=true | false
```

```
<integer-constant>::=<whole-number>
```

2.11 Class

Brief overview. Class specifies a classification of objects and the attributes that characterize the structure of those objects. A class is a kind of type which describes a set of objects that have the same attributes, constraints and semantics. Classes may have attributes, connected associations and form class hierarchy.

Generalizations

It is specialized from Type.

Attributes

[inh] name:String - the name of the class.

```
<class-name>::=<name>
```

isAbstract:Boolean - indicates that class is abstract.

Associations

[inh] comment[*] - comments on this class.

[inh] owningPackage [0..1] - a package that owns this class.

[inh] Typed[*] - association ends that are connected to this class.

[inh] molaElem[*] - referenceable elements, actually, pointers of a MOLA procedure constrained by this class.

specialization[*] - Generalization instances where this class is a superclass.

generalization[*] - Generalization instances where this class is a subclass. Currently only single generalization for a class is permitted in MOLA.

ownedAttribute[*] - attributes and navigable association ends owned by the class.

Notation. A class is shown as a rectangle which is divided by horizontal line (see Fig. 3). The upper compartment contains the name label and a special icon. The lower compartment

contains the list of attributes owned directly by the class. An item of the list consists of attribute name, type and multiplicity of property if it is other than 1

It is recommended:

- To put the class name in italics if the class is abstract, otherwise boldface

- Capitalize the first letter of class names.

- Begin attribute names with a lowercase letter.

A class may be displayed in the project tree as a node with a special icon. Attributes may be shown as child nodes of the class node.

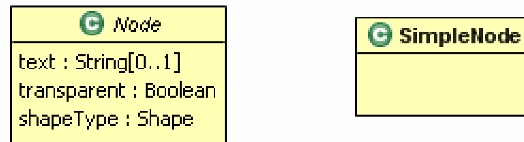

Fig. 3. Class and attribute notation

2.12 Generalization

Brief overview. `Generalization` is a relationship between a more general class and a more specific class. Each instance of the specific class is also an indirect instance of the general class. Thus, the specific class inherits the properties of the more general class. Circular generalizations are not permitted.

Generalizations

It is specialized from `Element`.

Associations

[inh] `comment[*]` - comments on this generalization.

`general`[1] - References the general class in the generalization relationship

`specific`[1] - References the specializing class in the generalization relationship

Notation. Generalization is shown as line with a hollow triangle as an arrowhead between the symbols representing the involved classes (See Fig. 4). The arrowhead points to the symbol representing the general class.

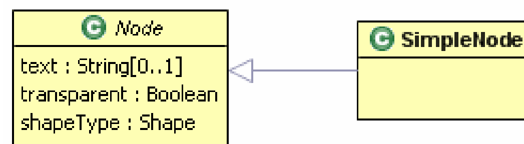

Fig. 4. Generalization notation

2.13 TypedElement

Brief overview. `TypedElement` is an abstract metaclass which represents metamodel elements which have a type that serves as a constraint on the range of values the typed element can represent.

Generalizations

It is specialized from `NamedElement`.

Specializations

There is a subclass `Property`.

Attributes

[inh] `name:String` - the name of the typed element.

Associations

[inh] `comment[*]` - comments on this typed element.

`type`[0..1] - type which constrains the possible values of typed element.

2.14 Property

Brief overview. `Property` is structural element that characterizes the structure of objects classified by the same class. There are two types of properties - attributes and association ends. Attributes are primitive or enumeration typed in MOLA. They describe properties of the class that can be expressed by value. An association end connects a class to an association that describes a relationship of this class to another class (or the same class)

Generalizations

It is specialized from `TypedElement`.

Attributes

[inh] `name:String` - the name of the property. The name must be unique within scope of properties determined by the owning class. More precisely, the name must be unique within the set of all attributes and association ends belonging to this class and all superclasses of it. In MOLA this set includes also non-navigable association ends. No property redefinition is permitted in MOLA. The name of an association end is called *rolename* in MOLA.

```
<property-name>::=<atribute-name> | <role-name>
```

```
<atribute-name>::=<name>
```

```
<role-name>::=<name>
```

`isOrdered`:`Boolean`[0..1] - indicates that this property is ordered. Only an association end may be ordered in MOLA (it makes sense only for the '*' multiplicity). Both association ends that are members of the same association may not be ordered at the same time. The ordering of an association end specifies that links corresponding to this association are

ordered with respect to the instance at the opposite end of these links (it is relevant only for a set of links starting from a common instance). The ordering is taken into account by loops and actions in MOLA.

`isComposite`:`Boolean`[0..1] - indicates that this property is composite. Only association ends have this attribute set. That means that the association whose member is the association end, is a composition. The class at the opposite end of the association is part of the class at the composite association end. Both ends of an association cannot be composite.

`lower`:`Integer`[0..1] - specifies the lower bound of multiplicity interval for this property. Values may be 0 or 1 in MOLA.

`upper`:`Integer`[0..1] - specifies the upper bound of multiplicity interval for this property. `Upper` must be greater or equal than `lower`. Values may be 1 or '*' (this value is internally coded as -1). The upper bound of multiplicity interval may only be 1 for an attribute.

Other attributes of `Property` actually are not used in MOLA.

Associations

[inh] `comment[*]` - comments on this property.

[inh] `Type`[0..1] - the type which constrains the possible values of the attribute or the class connected by the association end. The type is a primitive type or an enumeration if the property is an attribute. The type is a class if the property is an association end.

`class`[0..1] - the class that owns this attribute or navigable association end.

`owningAssociation`[0..1] - the association that owns this non-navigable association end.

`association`[0..1] - the association whose member is this association end.

`assocLinkStart`[*] - MOLA association links whose source ends correspond to this association end.

`assocLinkEnd`[*] - MOLA association links whose target ends correspond to this association end.

Notation. This section provides the notation of properties that are attributes. For notation of association ends see Section 2.15. The attribute is shown as string that consists of name, the colon (':'), type name and multiplicity interval string. (See Fig. 3)

```
<attr-multiplicity-interval-string>::='0..1' | '1'
```

Multiplicity interval string may be omitted for '1' (this is the default value).

```
<attribute-notation>::=

  <attribute-name>' : '<type-name>'['<attr-multiplicity-
interval-string>']'
```

2.15 Association

Brief overview. `Association` specifies a semantic relationship that can occur between two classified instances.

Generalizations

It is specialized from `PackagableElement`.

Attributes

[inh] `name:String` - the name of the association.

Associations

[inh] `comment[*]` - comments on this association.

[inh] `owningPackage [0..1]` - the package that owns this association.

`ownedEnd`[*] – non-navigable association ends of the association.

`memberEnd`[2..2] - properties that serves as association ends.

`assocLink`[*] - MOLA association links which are constrained by this association.

Notation. An association is normally drawn as a solid line connecting two classes, or a solid line connecting a single class to itself (the two ends are distinct) (See Fig. 5). A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance.

An association end is the connection between the line depicting an association and the rectangle depicting the connected class. The name of an association end may be placed near the end of the line. The multiplicity string may be placed at the opposite side of the line. If the association end is composite the connection of the line is shown as a filled diamond. If the association end is ordered then the symbol {ordered} is shown near it. If the association end is navigable then it is shown as an open arrow. The navigability is used only to be compatible with the standard notation in MOF. It has no special semantics for MOLA.

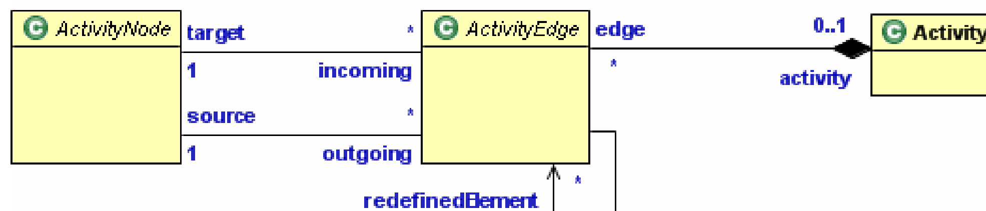

Fig. 5. Association notation

3 MOLA package - transformation definition facilities in MOLA

The MOLA package (see Fig. 6) describes facilities for the procedure definition in MOLA. The executable part of a MOLA transformation consists of MOLA procedures (the procedures written in MOLA). They may be grouped into packages. A MOLA procedure is a diagram similar to the activity diagram. A MOLA procedure contains statements that are executed when the transformation is started. The execution order is determined by control flows. There are call statements, loops, rules and text statements in a MOLA procedure. The MOLA procedure contains also start and end points - statements that denote the entry point and termination points of the MOLA procedure. The rule and the foreach-loop are the most typical statements of the MOLA language. Each rule in MOLA has the pattern and action part. Both are defined using class elements and association links. There are also textual elements that specify constraints and assignments in the rule.

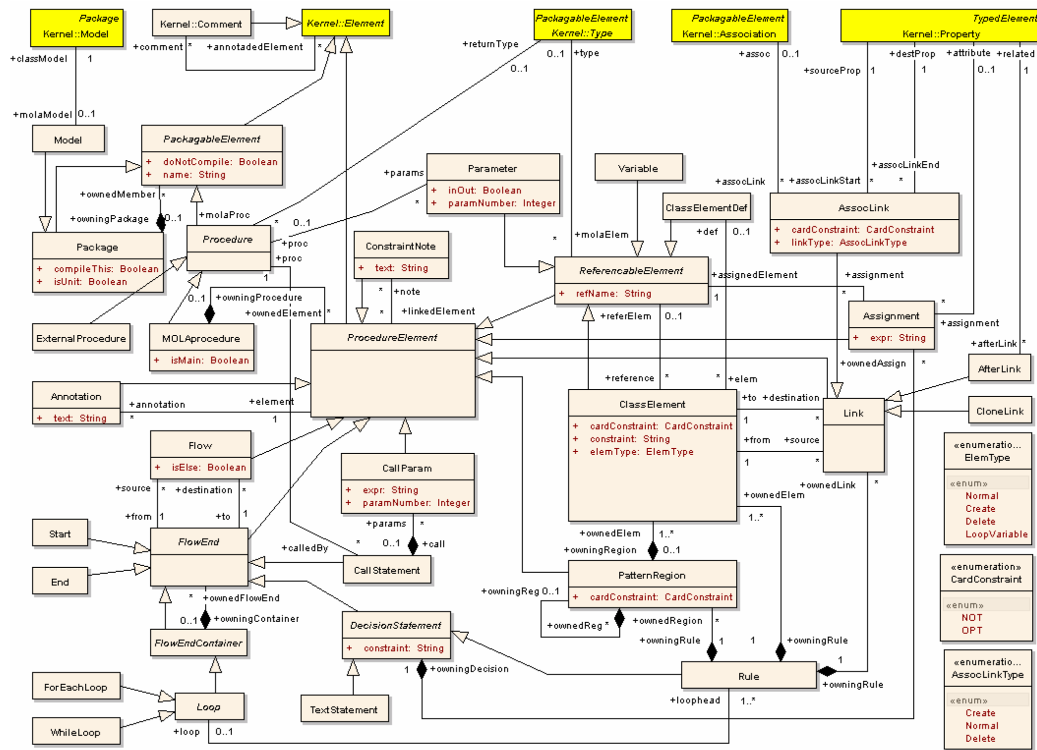

Fig. 6. MOLA package

The fragment of the MOLA metamodel that describes the means of procedure grouping is shown in Fig 7. The executable part of a MOLA transformation consists of procedures. Procedures may be grouped using packages.

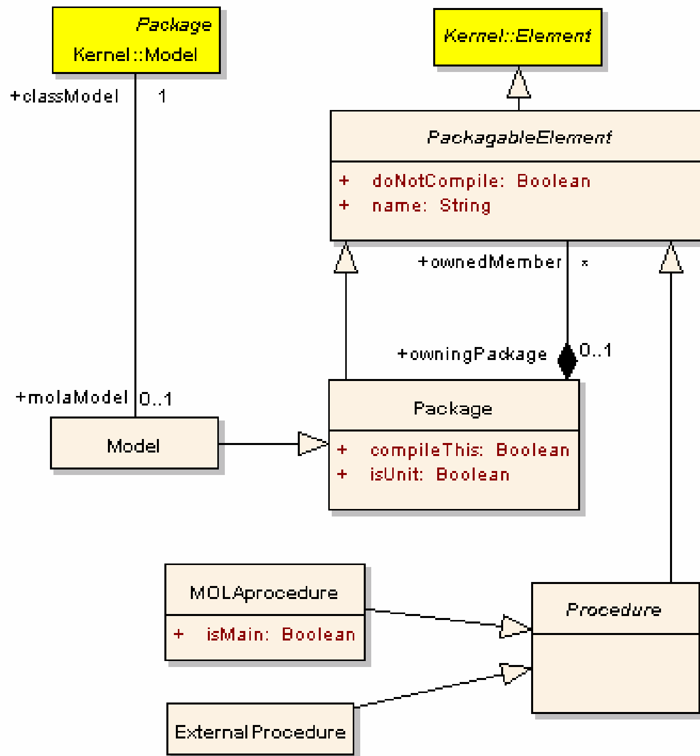

Fig. 7. Procedures and packages in MOLA

3.1 PackagableElement

Brief overview. `PackagableElement` is an abstract metaclass which represents MOLA elements which may be grouped into packages. If the element is grouped into a package then it is owned by that package.

Generalizations

It is specialized from `Kernel::Element`.

Specializations

There are subclasses `Package`, `Procedure`.

Attributes

`name:String` - the name of the packageable element. The name must be unique within the elements grouped into the package. The name is used to reference this element within the package. To reference a packageable element outside the package the full qualified name must be used. The full qualified name is obtained by prefixing the name with full qualified name of the owning package and '::' symbol.

```
<full-packageable-element-name>::=
      [<full-owningpackage-name>'::']<name>
```

[cd] `doNotCompile:Boolean` - indicates that the packageable element must be excluded from the compilation scope.

Associations

[inh] `comment[*]` - comments on this packageable element.

`owningPackage [0..1]` - a package that owns this packageable element.

Notation. Packageable element may be shown in the project tree as a node, which has an icon and a name.

3.2 Package

Brief overview. `Package` is a MOLA element that is used to group packageable elements. Grouped elements are owned by the grouping package. The package may be owned by other package. A package may serve as a compilation unit.

Generalizations

It is specialized from `PackagableElement.`

Specializations

There is a subclass `Model`.

Attributes

[inh] `name:String` - the name of the package.

[inh][cd] `doNotCompile:Boolean` - indicates that the package must be excluded from the compilation scope.

[cd]`isUnit:Boolean` - indicates that the package is a compilation unit. A compilation unit includes a set of MOLA procedures which all are recompiled if one of them has been changed. The compilation unit includes all MOLA procedures that are owned directly or indirectly by owned packages which are not compilation units.

[cd]`compileThis:Boolean` - indicates that the unit must be compiled. This attribute may be set only if the package is a unit (`isUnit=true`). If the `doNotCompile` has been set to *true* then this unit is not compiled.

Associations

[inh] `comment[*]` - comments on this package.

[inh] `owningPackage [0..1]` – the package that owns this package.

`ownedMember`[*] - packageable elements that are owned by this package. The package may not contain itself directly or indirectly.

Notation. Package may be shown in the project tree as a node, which has a special icon and a name. All owned elements may be shown as child nodes of the package node.

3.3 Model

Brief overview. `Model` is a special case of package - the root package - that means the package without owning package. There is exactly one model in a MOLA transformation. The model serves as a default compilation unit.

Generalizations

It is specialized from `Package`.

Attributes

[inh] `name:String` - the name of the model.

[inh][cd] `doNotCompile:Boolean` - indicates that the model will be excluded from compilation scope. By setting this attribute to *true* nothing will be compiled.

[inh][cd]`isUnit:Boolean` -indicates that the package is a compilation unit. A compilation unit includes a set of MOLA procedures which all are recompiled if one of them has been changed. The compilation unit includes all MOLA procedures that are owned directly or indirectly by owned packages which are not compilation units. The model is compilation unit. Thus, `isUnit=true` for a model.

[inh][cd]`compileThis:Boolean` - indicates that the unit must be compiled.

Associations

[inh] `comment[*]` - comments on this model.

[inh] `owningPackage [0..1]` - a package that owns this model. There is no owning package for a model.

[inh] `ownedMember`[*] - packageable elements that are owned by this model.

`classModel`[1] - `Kernel::model` which specifies metamodel that describes models to be transformed by MOLA procedures.

Notation. Model may be shown in the project tree as a node, which has the special icon and a name.

3.4 Procedure

Brief overview. `Procedure` is an abstract metaclass which represents procedures in MOLA. There are two types of procedures - MOLA procedure and external procedure. A procedure may have parameters. A procedure may be called from a MOLA procedure.

Generalizations

It is specialized from `PackageableElement`.

Specializations

There are subclasses `MOLAprocedure, ExternalProcedure`.

Attributes

[inh]`name:String` - the name of the procedure.

[inh][cd] `doNotCompile:Boolean` - indicates that this procedure must be excluded from the compilation scope. Call statements that call this procedure are compiled as empty statements doing nothing.

Associations

[inh] `comment[*]` - comments on this procedure.

[inh]`owningPackage [0..1]` - the package that owns this procedure.

`params`[*] - parameters of this procedure.

`calledBy`[*] - call statements that are calling this procedure.

Semantics. A procedure contains statements written in MOLA or represents an external procedure written in any programming language. A procedure may be *called* from a MOLA procedure. When the execution of the procedure is ended then the control is passed back to the calling procedure.

3.5 MOLAprocedure

Brief overview. MOLA`procedure` is a procedure which is implemented using MOLA transformation language. One MOLA procedure in a MOLA model is the main procedure.

Generalizations

It is specialized from `Procedure.`

Attributes

[inh] `name:String` - the name of the MOLA procedure.

[inh][cd] `doNotCompile:Boolean` - indicates that the MOLA procedure must be excluded from the compilation scope.

`isMain:Boolean` - indicates that this MOLA procedure is the main procedure of a MOLA transformation. This procedure is called when the transformation is started. The execution of the transformation ends, when the execution of the main procedure has ended.

Associations

[inh] `comment[*]` - comments on this packageable MOLA procedure.

[inh] `owningPackage [0..1]` - the package that owns this MOLA procedure.

[inh] `params`[*] - parameters of this procedure. These links are not used, since parameters of a MOLA procedure are owned elements of the MOLA procedure and can be obtained via `ownedElement` association.

[inh] calledBy[*] - call statements that are calling this MOLA procedure.

ownedElement[*] - MOLA procedure elements owned by this MOLA procedure

Notation. A MOLA procedure is a diagram. Facilities used in a MOLA diagram are described in next subsections. MOLA diagram may be shown also as a node in the project tree.

Semantics. When a procedure is called MOLA *statements* are executed.

3.6 ExternalProcedure

Brief overview. ExternalProcedure is a procedure which is implemented using another programming language than MOLA (currently C++).

Generalizations

It is specialized from Procedure

Attributes

[inh] name:String - the name of the external procedure.

[inh][cd] doNotCompile:Boolean - indicates that the external procedure must be excluded from the compilation scope.

Associations

[inh] comment[*] - comments on this external procedure.

[inh] owningPackage [0..1] - the package that owns this external procedure.

[inh] params[*] - parameters of this external procedure

[inh] calledBy[*] - call statements that are calling this procedure.

Notation. An external procedure is shown in the project tree.

The fragment of the MOLA metamodel that describes the facilities for handling control inside the MOLA procedure is shown in Fig 8. The MOLA procedure consists of statements and control flows.

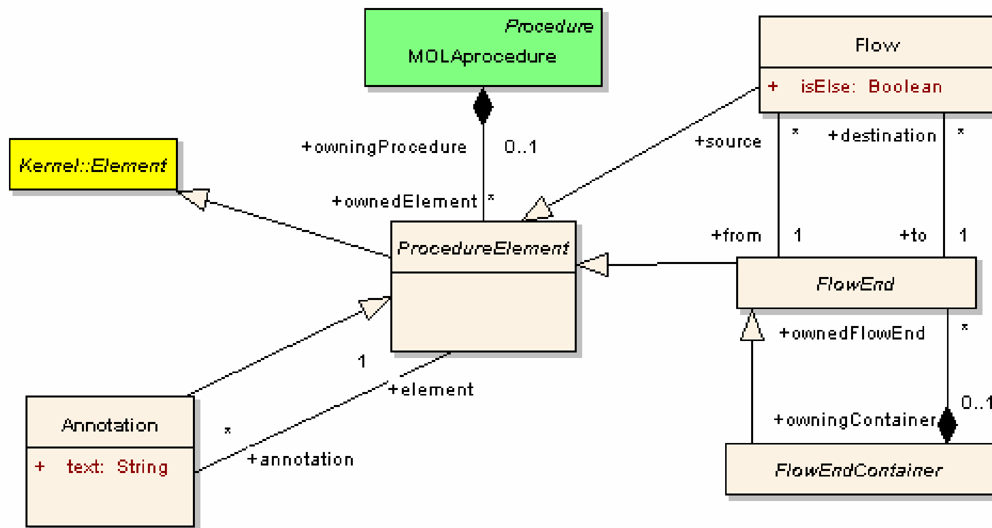

Fig. 8. Control flows in MOLA

3.7 ProcedureElement

Brief overview. `ProcedureElement` is an abstract superclass for all classes that describe elements of the MOLA procedure.

Generalizations

It is specialized from `Kernel::Element`

Specializations

There are subclasses `Annotation, ConstraintNote, Flow, FlowEnd, CallParam, ReferencableElement, Link, Assignment.`

Associations.

[inh]`comment[*]` - comments on this procedure element.

`owningProcedure`[0..1] - the MOLA procedure which owns this procedure element. All procedure elements must have the owning MOLA procedure. The only exception is `Parameter` (for external procedures).

`annotation`[*] - annotations added to this procedure element.

3.8 Annotation

Brief overview. `Annotation` is a textual element of the MOLA procedure which adds additional semantic information on an element of the MOLA procedure.

Generalizations

It is specialized from `ProcedureElement`

Attributes

text:String - the text of the annotation.

Associations.

*[inh]*comment[*] - comments on this annotation.

*[inh]*owningProcedure[1] - the MOLA procedure which owns this annotation.

element[1] - the procedure element this annotation is linked to.

Notation. The notation of an annotation is dependent of an element the annotation is added to.

Semantics. The semantics is dependent of an element the annotation is linked to.

3.9 FlowEnd

Brief overview. FlowEnd is an abstract metaclass which represents statements (executable elements) of the MOLA procedure. A statement may be a rule, loop, call, procedure start, procedure end or textual decision statement. The control of the procedure execution is passed to the next statement via control flow. A statement may be owned by another statement – a container.

Generalizations

It is specialized from ProcedureElement

Specializations

There are subclasses Start, End, FlowEndContainer, CallStatement, DecisionStatement.

Associations.

*[inh]*comment[*] - comments on this statement.

*[inh]*owningProcedure[0..1] - MOLA procedure which owns this statement.

source[*] - outgoing control flows.

destination[*] - incoming control flows

owningContainer[0..1] - the statement container which owns this statement, if any.

Notation. A statement is shown as a node in the control flow graph of the MOLA procedure.

Semantics. A statement is an executable element of the MOLA procedure. A statement is executed accordingly to the semantics of the concrete statement type.

3.10 Flow

Brief overview. Flows - control flows - determine the execution order of statements in the MOLA procedure. A control flow is an arrow which connects two statements. A control flow may be marked as an alternate (*ELSE*) control flow.

Generalizations

It is specialized from `ProcedureElement`

Attributes

`isElse:Boolean` - indicates that the control flow is an alternate (*ELSE*) control flow.

Associations.

[inh]`comment[*]` - comments on this flow.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this procedure element.

`from`[1] - the statement the control flow is going from. The flow is outgoing with respect to this statement.

`to`[1] - the statement the control flow is going to. The flow is incoming with respect to this statement.

Notation. Control flow is shown as a dashed line with a hollow triangle as an arrowhead between the symbols representing involved statements (see Fig. 9). The arrowhead points to the symbol representing the statement the control flow is going to. If the control flow is alternate (*ELSE*) control flow then a label '{ELSE}' is shown above the line near the statement the control flow is going from.

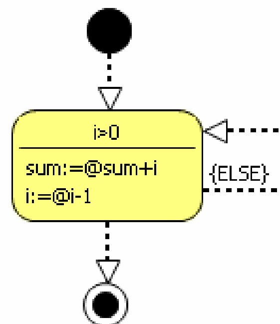

Fig. 9. Control flow notation

Semantics. Control flow connects exactly two statements. A statement may have any number of incoming control flows and no more than two outgoing control flows. If there are two outgoing flows then one of them must be an alternate (*ELSE*) control flow. Exact number of incoming and outgoing control flows is determined by the precise type of the statement. The way the control flow that determines the next statement is chosen depends also on the statement type.

3.11 FlowEndContainer

Brief overview. `FlowEndContainer` is an abstract metaclass which represents statements that may contain other statements.

Generalizations

It is specialized from `FlowEnd`

Specializations

There is a subclass Loop.

Associations.

*[inh]*comment [*] - comments on this statement.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this container

*[inh]*source[*] - outgoing control flows.

*[inh]*destination[*] - incoming control flows

*[inh]*owningContainer[0..1] - the statement container which owns this statement, if any.

ownedFlowEnd[*] – statements owned by this container. The container may not contain itself directly or indirectly.

The fragment of the MOLA metamodel that describes all possible statements in MOLA is shown in Fig 10.

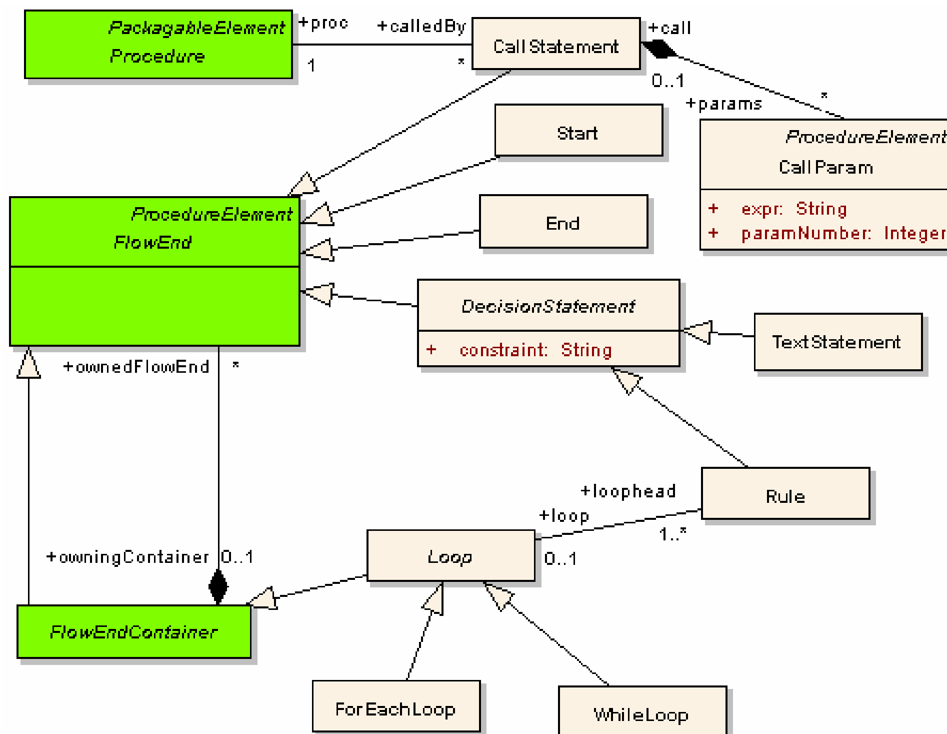

Fig. 10. Statements in MOLA

3.12 Start

Brief overview. Start is an element of MOLA procedure which represents the starting point of a MOLA procedure.

Generalizations

It is specialized from `FlowEnd`

Associations.

[inh]`comment[*]` - comments on this starting point.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this starting point

[inh]`source`[*] - outgoing control flows. Starting point has exactly one outgoing control flow. No alternate outgoing control flow is allowed.

[inh]`destination`[*] - incoming control flows. Starting point has no incoming control flows.

[inh]`owningContainer`[0..1] - the statement container which owns this statement, if any. Starting point may not be owned by a container

Notation. A starting point is shown as solid black circle. (See Fig. 11)

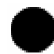

Fig. 11. Start symbol notation

Semantics. MOLA procedure may have exactly one starting point. The statement which will be executed next is determined by the outgoing flow. All statements that are not inside containers must be reachable by control flows from the starting point of the procedure.

3.13 End

Brief overview. `End` is an element of MOLA procedure which represents the termination point of a MOLA procedure.

Generalizations

It is specialized from `FlowEnd`

Associations.

[inh]`comment[*]` - comments on this termination point.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this termination point

[inh]`source`[*] - outgoing control flows. Termination point has no outgoing control flows.

[inh]`destination`[*] - incoming control flows. Termination point has at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this termination point, if any.

Notation. A termination point is shown as a solid circle with a hollow circle (See Fig. 12).

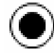

Fig. 12. End symbol notation

Semantics. A MOLA procedure must have at least one termination point. The termination point ends the execution of the MOLA procedure. Current values of in-out parameters are passed back to the calling MOLA procedure.

3.14 CallStatement

Brief overview. `CallStatement` is an element of MOLA procedure which represents call of a procedure. It can be either MOLA procedure or external procedure which is called by the call statement. The called procedure is executed. When it terminates the next statement after this call statement is executed. Call parameters are expressions.

Generalizations

It is specialized from `FlowEnd`

Associations.

[inh]`comment[*]` - comments on this call statement.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this call statement

[inh]`source`[*] - outgoing control flows. Call statement has one or none outgoing control flow. The outgoing control flow may be absent if the call statement is owned by a loop.

[inh]`destination`[*] - incoming control flows. Call statement has at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this call statement, if any.

`proc`[1] - the procedure which is called by this call statement.

`params`[*] - call parameters of the call statement.

Notation. A call statement is shown as a khaki rounded rectangle if the called procedure is a MOLA procedure (see Fig. 13a). A call statement is shown as a yellow rounded rectangle with pink border if the called procedure is an external one (see Fig. 13b). The body of the call statement symbol contains the name of the called procedure and the parameter list enclosed in parenthesis. Parameters are separated by commas. If the called procedure does not belong to the same package the full qualified name must be used.

```
<call-statement-notation>::=

(<called-procedure-name>|<called-procedure-full-qualified-
name>)'('[<call-parameters>]')'

<call-parameters>::=<call-parameter>{', '<call-parameter>}*
```

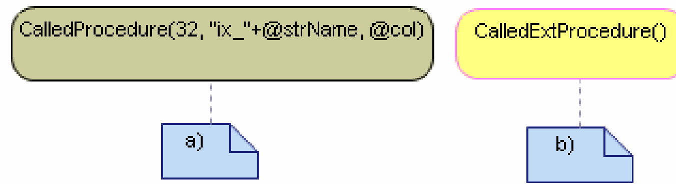

Fig. 13. Call statement notation a) to a MOLA procedure; b) to an external procedure

Semantics. A call statement is used to call MOLA or an external procedure. Call parameters must be supplied according to the order and types that are specified by definition of the called procedure. If a parameter is defined as *in* then the corresponding call parameter may be an arbitrary expression of the same type as the parameter (parameter is passed by value). If a parameter is defined as *in-out* then the corresponding call parameter must be a referenceable element (variable or parameter, but not pointer to class element) of the same type as the parameter (parameter is passed by reference).

The next statement is executed when the called procedure ends the execution. The next statement is determined by the outgoing control flow. The outgoing control flow may be absent if the call statement is owned by a loop. Then the next iteration of the loop is executed.

3.15 CallParam

Brief overview. `CallParam` is a textual element of MOLA procedure which is used to specify the value of the parameters that are passed to a called procedure.

Generalizations

It is specialized from `ProcedureElement`

Attributes

`expr:String` – expression (see Chapter 4 for details) that is evaluated and the result is passed as a call parameter value, when the call statement is executed. The type of the expression is constrained by the definition of the called procedure - the parameter with the same number constrains the type of the expression.

`<call-parameter>::=<expression>`

`paramNumber:Integer` – the number of the call parameter.

Associations.

[inh]`comment[*]` - comments on this call parameter. No comments may be added to the call parameter.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this call parameter. This link may be absent.

`call`[1] - call statement that owns the call parameter.

Notation. A call parameter is a textual element and it contains the expression which must be evaluated. A call parameter is shown in the body of the call statement. (see Fig. 13a). Call parameters are separated by commas.

3.16 DecisionStatement

Brief overview. `DecisionStatement` is an abstract metaclass which represents the conditional statements in MOLA. The condition of a decision statement is examined. The action is executed depending on the state of condition. There are two types of conditional statements – graphical (rule) and textual (text statement). The decision statement may contain *actions (action part)* – object and link creations, deletions, attribute and variable assignments.

Generalizations

It is specialized from `Flowend`

Specializations

There are subclasses `Rule, TextStatement`.

Attributes

`constraint:String[0..1]` - the constraint expression (See Chapter 4 for details) that forms the textual part of the condition.

`<constraint>::=<constraint-expression>`

Associations.

[inh]`comment[*]` - comments on this decision statement.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this decision statement

[inh]`source`[*] - outgoing control flows. A decision statement has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the decision statement is owned by a loop.

[inh]`destination`[*] - incoming control flows.

[inh]`owningContainer`[0..1] - the statement container which owns this decision statement, if any.

`ownedAssign`[*] – attribute and variable assignments owned by this decision statement.

Semantics. A decision statement may contain a condition. If the condition holds, then the action part of the decision statement is executed and the next statement is found by outgoing control flow without *ELSE* mark. If the condition fails, then the action part is not executed and the next statement is found by the alternate (*ELSE*) outgoing control flow. If the condition is not present then it is assumed to be *true*. The procedure execution is terminated, if the corresponding outgoing control flow is absent and the decision statement is not owned by a loop. If the decision statement is owned by a loop, then the next iteration of the loop is executed.

3.17 Rule

Brief overview. `Rule` is a graphical decision statement - the most typical construct in MOLA. The condition is specified by a *pattern* and a textual constraint. A pattern is built using class elements and association links. A class element corresponds to a metamodel class. An association link connecting two class elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class elements and links, which are compatible to the metamodel for this transformation. The pattern has been matched if appropriate instances in the model have been found.

The action part of the rule is built also using class elements and association links. Class elements of the action part represents object creation and deletion, but association links - link creation and deletion. Textual assignments for matched instances are used to set attribute values. Action part may use the instances found by the pattern matching.

There is special type of rule - *loophead*. A loophead specifies the condition of a loop.

Generalizations

It is specialized from `DecisionStatement`

Attributes

[inh]`constraint:String[0..1]` - the constraint expression (See Chapter 4 for details) that forms additional textual part of the condition.

Associations.

[inh]`comment[*]` - comments on this rule.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this rule

[inh]`source`[*] - outgoing control flows. A rule has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the decision statement is owned by a loop. The loophead may not contain alternate outgoing control flow.

[inh]`destination`[*] - incoming control flows. There are no incoming control flows if the rule is a loophead.

[inh]`owningContainer`[0..1] - the statement container which owns this rule, if any.

[inh]`ownedAssign`[*] - attribute and variable assignments owned by class elements in this rule.

`loop`[0..1] - the loop, if this rule is the loophead of that rule. This loop must be also the owner of this loophead.

`ownedElem`[1..*] - class elements that specify the pattern and actions of the rule.

`ownedLink`[*] - association links that specify the pattern and actions of the rule.

Notation. A rule is shown as a gray rounded rectangle (See Fig. 14). Class elements and association links are shown inside the symbol of a rule. Class elements may contain constraints, attribute assignments and annotations.

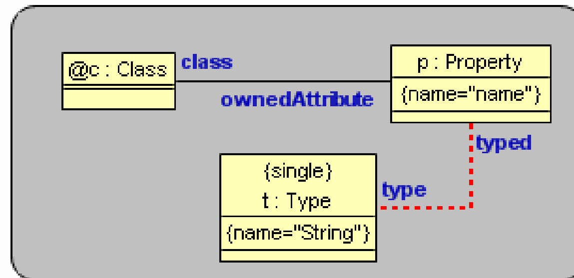

Fig. 14. Rule notation

Semantics. The condition of a rule is formed using pattern and an additional textual constraint. The condition holds if the pattern matches and the textual constraint (if present) evaluates to *true*. More on pattern semantics see Chapter 5. If the condition holds the action part of the rule is executed.

The action part of the rule includes:

- Instance and link creation via *create* class elements and association links respectively.
- Instance and link deletion via *delete* class elements and association links respectively.
- Assigning values of attributes.

There is a special type of rule – *loophead*. A loophead specifies the condition of a loop. A while-loop is being executed while the condition of the loop holds. The loophead of a foreach loop contains a special class element – *loop variable*. The foreach loop is being executed for each distinct instance that corresponds to the loop variable.

3.18 TextStatement

Brief overview. `TextStatement` is a textual decision statement. The condition of a text statement is specified using a textual constraint. A text statement is used when no pattern matching and no instance creation is needed. The condition operates with variables and already matched instances. A text statement may contain assignments. New values can be set to variables and to attributes of instances.

Generalizations

It is specialized from `DecisionStatement`

Attributes

[inh]`constraint:String[0..1]` - the constraint expression (see Chapter 4 for details) that forms the condition of the text statement.

```
<constraint>::=<constraint-expression>
```

Associations.

[inh]`comment[*]` - comments on this text statement.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this text statement

[inh]`source`[*] - outgoing control flows. A text statement has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the text statement is owned by a loop.

[inh]`destination`[*] - incoming control flows. There is at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this text statement, if any.

[inh]`ownedAssign`[*] - attribute and variable assignments owned by this text statement.

Notation. A text statement is shown as yellow rounded rectangle. (See Fig. 15). The rectangle is divided by horizontal line. The upper compartment contains textual condition, but the lower compartment contains list of assignments. Any of the compartments may be absent.

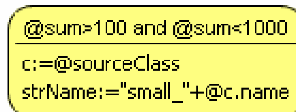

Fig. 15. Text statement notation

Semantics. The condition of a text statement is formed using textual constraint. The constraint is a constraint expression, which is evaluated to find the state of the condition. Assignments form the action part of the text statement.

3.19 Loop

Brief overview. `Loop` is an abstract metaclass which represents the loop statements in MOLA. The loop concept in MOLA is very similar to classical programming languages - some code is repeatedly executed if the condition of the loop holds. There are while and foreach loops in MOLA. A loop is a container – it owns other statements.

Generalizations

It is specialized from `FlowEnd`

Specializations

There are subclasses `ForeachLoop, WhileLoop`.

Associations.

[inh]`comment[*]` - comments on this loop.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this loop

[inh]`source`[*] - outgoing control flows. Loop has one or none outgoing control flow. The outgoing control flow may be absent if the loop is owned by another loop.

[inh]`destination`[*] - incoming control flows. Loop has at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this loop, if any.

`loophead`[1..*] - loopheads of the loop. The loop must have exactly one loophead.

Notation. The loop may contain other executable elements. They are shown inside the box representing the loop. The precise notation is dependent on the type of the loop.

Semantics. A loop can be *while*, *foreach fixed* and *foreach not-fixed*. A loop owns exactly one rule, called *loophead*. The loophead describes the condition of the loop. The loophead has no incoming control flow and no more than one outgoing control flow which is not alternate (ELSE) control flow. The loophead action part is executed if the condition of the loop holds. All other statements owned by the loop are called a *loop body*. It must be reachable from the loophead by control flows. If a statement in the loop body has no appropriate outgoing control flow then the next iteration of the loop is executed. If the condition of the loop fails then the next statement is executed. The next statement is found via outgoing control flow of the loop. The outgoing control flow may be absent if the loop is owned by another loop. Then the next iteration of the owning loop is executed.

Outgoing control flows may go out of the loop, but it is not permitted to draw an incoming control flow to the statement inside the loop from statement which is not directly or indirectly owned by the loop.

3.20 ForeachLoop

Brief overview. `ForeachLoop` is a loop, which is used to execute statements for each instance of a selected class such that the pattern matches. The condition of a foreach loop is specified by the loophead. A pattern of a loophead contains a special class element – a *loop variable*. The loop variable specifies the set of instances the loop is executed trough.

Generalizations

It is specialized from `Loop`

Attributes

`notFixed:Boolean` – indicates that a foreach loop is *not-fixed*. Otherwise it is *fixed.*

Associations.

[inh]`comment[*]` - comments on this foreach loop.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this foreach loop

[inh]`source`[*] - outgoing control flows. Foreach loop has one or none outgoing control flow. The outgoing control flow may be absent if the foreach loop is owned by another loop.

[inh]`destination`[*] - incoming control flows. Foreach loop has at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this foreach loop, if any.

[inh]`loophead`[1..*] – loopheads of the foreach loop. The foreach loop must have exactly one loophead.

Notation. A fixed foreach loop is shown as a rectangle with bold borders (see Fig. 16a). A not-fixed foreach loop is shown as a rectangle with bold borders and the right upper corner bent (see Fig. 16b). The loophead is shown as rule, which contains a loop variable – a class element with bold borders. The loop body is drawn inside the foreach loop symbol.

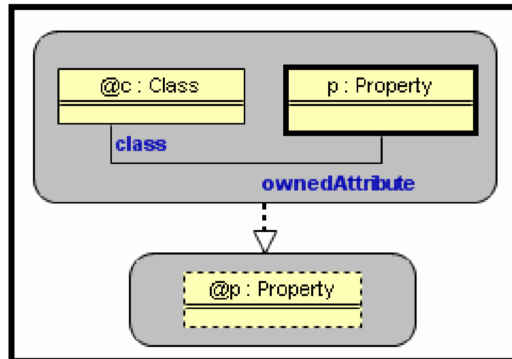

Fig. 16a Fixed foreach loop notation

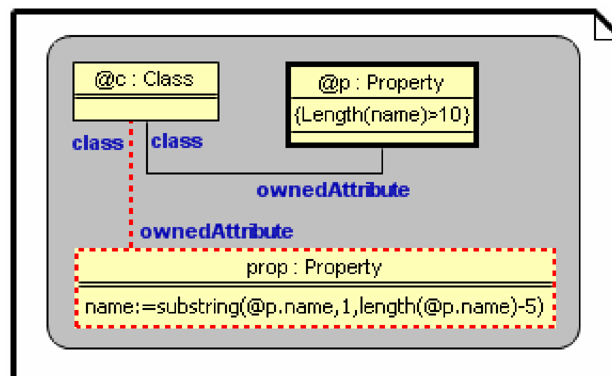

Fig. 16b Not-fixed foreach loop notation

Semantics. A foreach loop is a loop which is executed over a set of instances that are specified by the *loop variable*. The loop variable is a special class element. There is exactly one loop variable in a loophead. To find the instances a foreach loop must iterate through, the condition (loophead pattern and additional constraints) is evaluated (the pattern matched). The loop body is executed for all possible matches for the loophead, which differ by instances allocated to the loop variable. The order instances are traversed is dependent of the loophead pattern (see Chapter 5)

The condition is evaluated only once for a *fixed* foreach loop. Namely, all instances of the loop variable which have the corresponding pattern match are selected. The actions of the

loophead and loop body are executed for each selected instance. The fixed foreach loop corresponds to foreach loop in traditional programming languages.

The condition is evaluated on every iteration for a *not-fixed* foreach loop. Namely, the first instance of the loop variable which has the corresponding pattern match and has not been selected yet, is selected. The actions of the loophead and loop body are executed for it. Thus the not-fixed foreach loop is used when foreach loop iteration may produce a new instance of the loop variable that must be processed.

3.21 WhileLoop

Brief overview. `WhileLoop` is a loop, which is used to execute statements while the loop condition holds. A while-loop contains a condition which is specified using pattern included in the special rule - *loophead*.

Generalizations

It is specialized from `Loop`

Associations.

[inh]`comment[*]` - comments on this while-loop.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this while loop

[inh]`source`[*] - outgoing control flows. while loop has one or none outgoing control flow. The outgoing control flow may be absent if the while loop is owned by another loop.

[inh]`destination`[*] - incoming control flows. While-loop has at least one incoming control flow.

[inh]`owningContainer`[0..1] - the statement container which owns this while loop, if any.

[inh]`loophead`[1..*] - loopheads of the while loop. The while-loop must have exactly one loophead.

Notation. A while-loop is shown as a 3D-rectangle with bold borders (see Fig. 17). The loophead is shown as rule which has no incoming control flow. The loop body is drawn inside the foreach loop symbol.

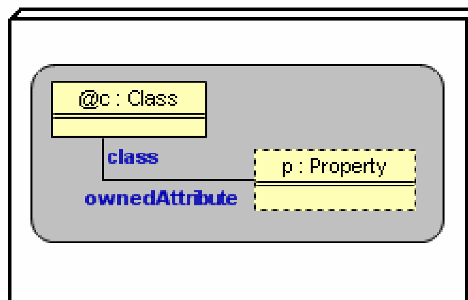

Fig. 17 While-loop notation

Semantics. A while-loop is a loop which is executed as long as the condition holds. The condition of the while-loop consists of a pattern and a textual constraint. If the condition of the while-loop holds then the loophead action part and loop body is executed, else the next statement found by the outgoing control flow is executed.

The fragment of the MOLA metamodel that describes the means of variable definition in MOLA is shown in Fig. 18.

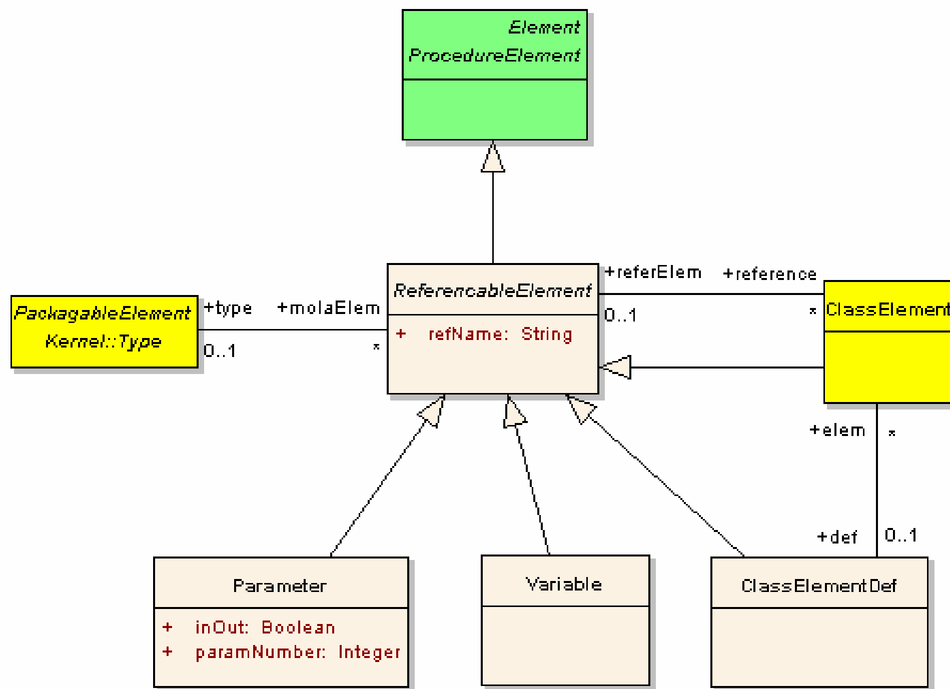

Fig. 18. Referencable elements

3.22 ReferencableElement

Brief overview. `ReferencableElement` is an abstract metaclass that represents procedure elements which may be referenced in a pattern and in an expression. In fact, referencable elements are used to introduce variable concept in MOLA. A referencable element has a name and a type and may have a value. The value of a referencable element is constrained by a type (class, enumeration or primitive type) defined in the metamodel. Parameters, variables and class elements are referencable elements in MOLA.

Generalizations

It is specialized from `ProcedureElement`

Specializations

There are subclasses `Parameter, Variable, ClassElementDef, ClassElement.`

Attributes

`refName:String[0..1]` - the name of the referencable element which is used to reference it within a pattern or an expression. If the name is set for the referencable element, it must be unique within the owning procedure.

```
<referencable-element-name>::=<pointer-name>|<elementary-variable-name>
```

```
<pointer-name>::=<name>
```

```
<elementary-variable-name>::=<name>
```

Associations.

[inh]`comment[*]` - comments on this referencable element.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this referencable element.

[inh]`annotation`[*] - annotations added to this referencable element.

`type`[0..1] - the type which constrains values of the referencable element. If the referencable element is class-typed, then it is called *pointer*, otherwise - *elementary variable*.

`reference`[*] - class elements that reference this referencable element. Only pointers may be referenced by a class element.

`assignment`[*] - assignments where the value of the referenceable element is set. If the referencable element is a pointer then values of instance attributes also may be set.

Semantics. Referencable elements in MOLA are parameters, variables and class elements. A referencable element has name and type, which is specified by a type defined in the metamodel.

A referencable element may have a value in the runtime. A referencable element may be used in the left hand side of an assignment. Then the value of the referenceable element is set. If a referencable element is a pointer, then values of attributes may be also set.

A referencable element may be referenced by a class element.

3.23 Parameter

Brief overview. `Parameter` is a referencable element, which defines a parameter of a procedure. When a procedure is called, then call parameters must be supplied. Those call parameters must comply to the parameter definitions.

Generalizations

It is specialized from `ReferencableElement`

Attributes

[inh]`refName:String[0..1]` - the name of the parameter which is used to reference it within a pattern or an expression.

`inOut:Boolean` – indicates that the call parameter is passed to a procedure by reference. Otherwise it is passed by value.

`paramNumber:Integer` - number of the parameter. Parameters of a procedure are numbered. Numbering starts from one. Parameter numbers determine the order call parameters must be supplied by a call statement.

Associations.

[inh]`comment[*]` - comments on this parameter.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this parameter. If the parameter is used for an external procedure, then this link can not be set.

[inh]`type`[0..1] - the type which constrains values of the parameter. The parameter may be of any type. A call parameter (actually the expression) supplied to call statement must be of the same type as the parameter with the same parameter number.

[inh]`reference`[*] - class elements that reference this parameter. Only class-typed parameters may be referenced by a class element.

[inh]`assignment`[*] - assignments where the parameter is in the left side. The parameter value may be set by an assignment. If the parameter is a pointer then values of instance attributes also may be set.

`proc`[0..1] - the procedure this parameter corresponds to. This link must be set, if the procedure is external.

Notation. A parameter may be shown in the project tree as child node of the procedure owning it. If a parameter is owned by a MOLA procedure, then it is shown as white convex flag if the parameter is *in* (see Fig. 19a), and it is shown as white hexagon, if the parameter is *in-out* (see Fig. 19b). The body of the parameter symbol contains the upper and the lower part. The upper part is the string which consists of the parameter name, prefixed with '@'-symbol, and type name, prefixed by the ' : 'string. If the type is included in a package - the full name of the owning package in the curly brackets is displayed in the lower part. The lower right corner contains the parameter number.

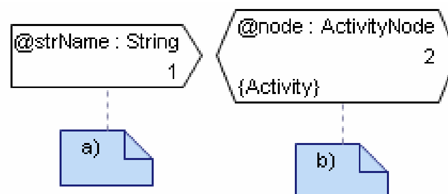

Fig. 19. Parameter notation a) in b) in-out

Semantics. Parameters define the types and order of arguments - call parameters - that must be supplied when a procedure is called. The type may be a class, an enumeration or a

primitive type. The order of parameters is determined by parameter numbers. Parameter numbering must be started from one.

A parameter is a referencable element, whose value is set when procedure is started. Principles of the parameter passing mechanism are identical to traditional programming languages. A parameter may be passed by reference, then it is called *in-out parameter,* or by value, then it is called *in parameter*. Note, if the parameter is pointer and even it is *in*, changes done to the instance (create link, set attribute, etc.) are permanent.

A parameter may be referenced by a class element

3.24 Variable

Brief overview. `Variable` is a referencable element, which defines a variable in the MOLA procedure. A variable gains the value via assignment operation. It may be used in a pattern, in a textual constraint or in an assignment. A variable may be of any type.

Generalizations

It is specialized from `ReferencableElement`

Attributes

[inh]`refName:String[0..1]` - the name of the variable which is used to reference it within an expression or assignment.

Associations.

[inh]`comment[*]` - comments on this variable.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this variable.

[inh]`type`[0..1] - the type which constrains values of the variable.

[inh]`reference`[*] - class elements that reference this variable. Only class-typed variables may be referenced by a class element.

[inh]`assignment`[*] - assignments where the value of the variable is set. If the variable is a pointer then values of instance attributes also may be set.

Notation. A variable is shown as a white rectangle (see Fig. 20). The body of the variable symbol contains the upper and the lower part. The upper part is the string which consists of the variable name, prefixed with ‘@‘-symbol, and type name, prefixed by ‘ : ‘-string. If the type is included in a package then the full name of the owning package in the curly brackets is displayed in the lower part.

Fig. 20. Variable notation

Semantics. A variable can be used in a pattern, in an expression or in an assignment.

A variable may be referenced by a class element

3.25 ClassElementDef

Brief overview. `ClassElementDef` is a referencable element that represents the definition of a non-reference class element. A class element definition defines a pointer, which can be reused by several class elements. This is a MOLA element which has no direct counterpart in the graphical syntax of MOLA.

Generalizations

It is specialized from `ReferencableElement`

Attributes

[inh]`refName:String[0..1]` - the name of the class element.

`<class-element-name>::=<name>`

Associations.

[inh]`comment[*]` - comments on this class element definition. No comments may be added to the class element definition.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this class element definition.

[inh]`type`[0..1] - the type which constrains values of the class element. This type may only be a class.

[inh]`reference`[*] - class elements that reference this class element definition. In fact, these class elements reference the class element which is defined by this definition.

[inh]`assignment`[*] - assignments where the value of attribute of the instance (pointed to by a class element using this definition) is set. A class element definition may be used in the left side of the assignment only to set the value of an attribute in text statement.

`elem`[*] – non-reference class elements that are defined by the class element definition. There must be at least one such class element. All defined class elements must be in different rules.

Semantics. This metaclass is added to the MOLA metamodel mainly to solve the problem of the name reuse of class elements. Class elements may have the same name and type, if they are used in different patterns. When the reference to such class element is needed, then there is ambiguity. To solve this issue this little bit artificial metaclass is introduced. Thus for each non-reference class element the corresponding class element definition is introduced “behind the scene“ - such class element must have `def` link to the appropriate definition. Certainly, several class elements may share the same definition if they have the same name. There is no graphical representation of the class element definition. If the class element references other class element, technically the `referElem` link must be set to the definition of that class element, not to the class element itself.

The fragment of the MOLA metamodel that describes the facilities of the pattern definition is shown in Fig 21. The patterns in MOLA are defined using class elements and association links.

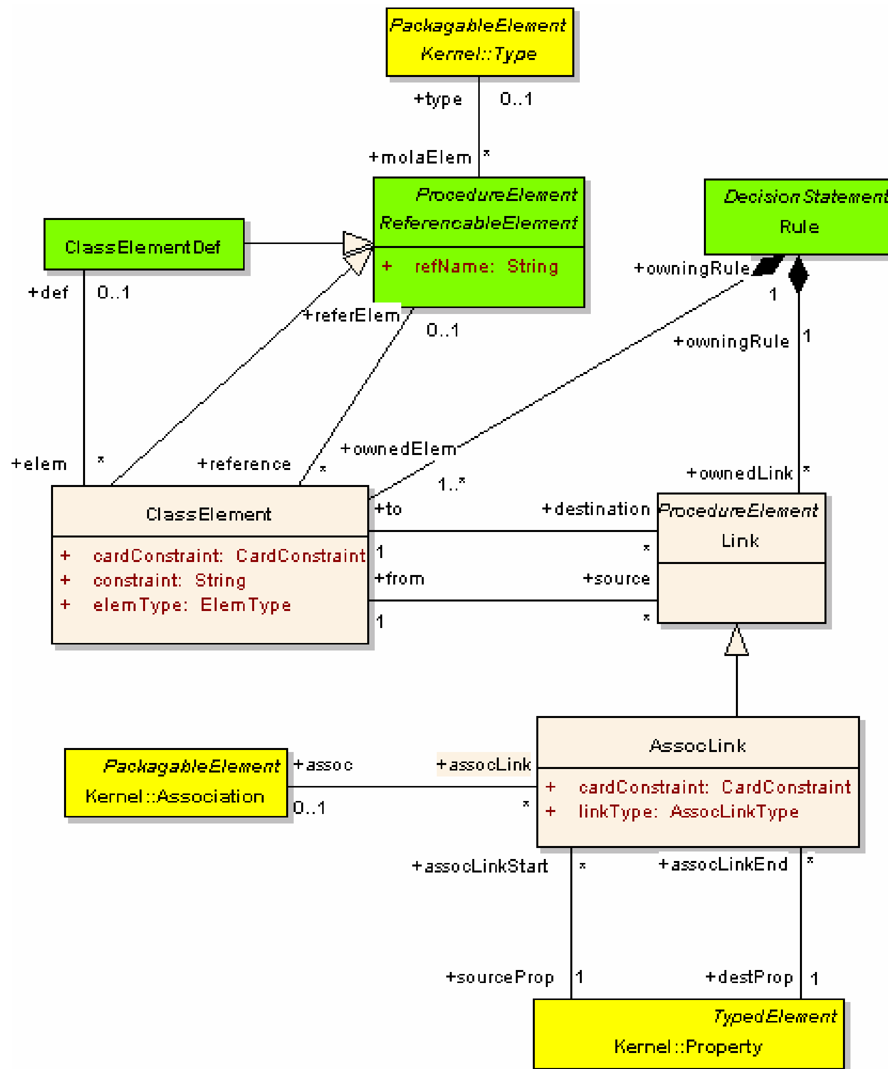

Fig. 21. Pattern definition elements

3.26 ClassElement

Brief overview. `ClassElement` is a graphical element that is used to define a pattern and actions of a rule. The class element represents an instance of a particular metamodel class. The class element may be *non-reference* or *reference*. A non-reference class element is used to get a pointer to a particular instance. A reference class element is used to reference a known instance. Non-reference class elements can be *normal*, *create*, *delete* and *loop variable*. Reference class element can be *normal* or *delete*. Normal class element is used to build a pattern. It specifies that the condition of the rule may be *true* only if there is an instance which matches to the features defined by the class element. Create class element is used to create an instance of the particular type. Delete elements are used to delete the

particular instance. Loop variable is a special type of class element that is used to denote the instance set a foreach loop is iterating through. All class elements may contain attribute value assignments, which are executed if the condition of the rule holds.

Generalizations

It is specialized from `ReferencableElement`

Attributes

[inh]`refName:String[0..1]` - the name of the class element. It is derived from the class element definition or the referenced element.

`constraint:String`[0..1] - additional constraint on this class element. It is a constraint expression (see Chapter 4). The instance is matched only if the constraint is *true*. Constraints can be added to class elements that are used to build the pattern of the rule. They can be *normal*, *delete* or *loop variable* class elements. Mainly the constraints are set on attribute values. Note, that a constraint may contain a reference (pointer) to another class element of the same pattern. For example, in order to find two instances with the same name, the constraint actually must apply to the whole pattern (not only to the class element).

`<constraint>::=<constraint-expression>`

`elemType` - the type of the class element. It may be *normal, create*, *delete* or *loop variable*.

`cardConstraint:CardConstaint`[0..1] - the cardinality constraint. It may be set for a normal class element. Currently, the only cardinality constraint is the *NOT* constraint. A class element with NOT constraint is called the *NOT-element*. The semantics of the NOT-element is opposite to a normal class element without cardinality constraint. The condition of a pattern holds only if there is no instance which matches to the features defined by the NOT-element. A NOT-element may not participate in any action.

Associations.

[inh]`comment[*]` - comments on this class element.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this class element.

[inh]`annotation`[*] - annotations added to this class element. Annotations are used to add semantics to the pattern. An annotation may be added to a *normal* or *delete* non-reference class element which has no cardinality constraints. No more than one annotation may be added. Currently there are two types of annotations – *single* and *start*. More on semantics of the pattern see Chapter 5

[inh]`type`[0..1] - the type which constrains values of the class element. It is derived from the class element definition or referenced element. Class element may be only class-typed.

[inh]`reference`[*] - class elements that reference this class element. Actually, this link is never set. If any class element references another class element, the link is set to the definition of this class element.

[inh]`assignment`[*] - assignments where the values of instance attributes are set.

`owningRule`[1] - the rule that is owning this class element.

`def`[0..1] - class element definition for a *non-reference* class element.

`referElem`[0..1] - the referencable element that is referenced by this class element. This link is set for class element that is a *reference*. A class element may reference a class-typed referencable element (variable, parameter or class element). If it references other class element, then this link actually is set to the definition of that class element.

`source`[*] - outgoing association links.

`destination`[*] – incoming association links.

Notation. Class element is shown as a rectangle. It may be placed only inside a rule. A class element has three compartments. The upper compartment contains the name of the class element and the type name. The name of the class element is prefixed by '@'symbol if the class element is a reference. If the class element is annotated or it has cardinality constraint then the annotation text or cardinality constraint text respectively is shown above the name of the class element in curly brackets. If the type is owned by a package, the full name of the package is shown below the class element name (also in curly brackets). The middle compartment of the class element may contain additional constraints on this class element – mainly constraints on attribute values. Constraints are also placed in curly brackets. The lower compartment of the class element may contain the list of attribute assignments (one per line). If the class element is *normal* then the border of the class element symbol is a black solid line (see Fig. 22a). If the class element is *create* then the border of the class element symbol is a red dotted line (see Fig. 22b). If the class element is *delete* then the border of the class element symbol is black dashed line (see Fig. 22c). If the class element is a *loop variable* then the border of the class element symbol is black bold line (see Fig. 22d).

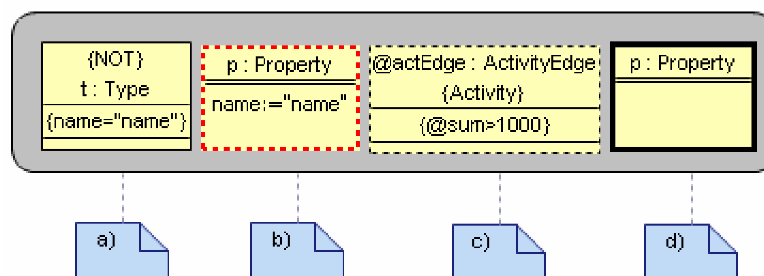

Fig. 22. Class element notation a) normal b) create c) delete d) loop variable

Semantics. A class element represents a particular instance (exactly one) in the model. It has a name - an identifier that can be used to refer to this instance in the current MOLA procedure. A class element has also the type specification - class from the metamodel - that constrains the scope of possible instances that can be represented by this class element.

Class elements may be of two kinds - *reference* and *non-reference*. A non-reference class element gets the pointer to particular instance when a pattern has been matched or a create instance operation has been performed in the rule owning the class element (during the execution of the rule). A reference class element gets the pointer to particular instance before

the execution of the rule. It refers to the referencable element that already has the value. It may be a parameter, a variable or already matched class element.

A *normal* class element is used in a pattern. It expresses the fact that the condition of the rule may hold (i.e., the pattern matches) only if there exists an instance of the type specified by the class element and the constraint for this class element evaluates to *true*. If the pattern matches then the class element represents the matched instance. The *normal* class element may be a reference. Then it represents the particular instance in the pattern. There may be a constraint also set on the reference. The role of a reference class element in a pattern is to add additional constraints on non-reference class elements in the pattern. The *delete* element plays the same role as a *normal* element in a pattern, but the particular instance is deleted in the action part of the rule. When the instance is deleted, all links connected to it also are deleted.

The *create* class element is used to create an instance of the particular type. A create class element may be only *non-reference*. If the condition of the rule holds (pattern matches), the instance denoted by a *create* class element is created. Then it represents the newly created instance.

The *loop variable* is used in the loophead of a foreach loop. It denotes the class element that is used to find the set of instances to iterate through. It is a *non-reference* class element. Otherwise it has the same semantics as *normal* class element.

A class element may contain assignments, which set attribute values for the corresponding instance. They are performed if the condition of the rule (or loophead) holds.

A class element may be connected to another class element by an association link. An association link may play different roles. It may be part of the pattern - it may form a constraint on links connecting the relevant instances. An association link may also be in the action part of the rule - be a create link or delete link.

3.27 Link

Brief overview. `Link` is an abstract metaclass that represents directed connectors that may be drawn in a rule between two class elements. Mainly it is used to specify constraint on particular link or to create or delete a particular link.

Generalizations

It is specialized from `ProcedureElement`

Specializations

There is a subclass `AssocLink`.

Associations.

[inh]`comment[*]` - comments on this connector.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this connector.

[inh]`annotation`[*] - annotations added to this connector.

owningRule[1] - the rule that owns the connector.

from[1] - the class element the connector is going from - the source of the connector.

to[1] - the class element the connector is going to - the target of the connector.

Constraints.

The source and the target class elements must be in the same rule.

Notation. A connector is a line that connects two class elements. A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance. The precise notation is dependent on the precise type of a connector.

3.28 AssocLink

Brief overview. AssocLink is a connector, called association link. It is used to define a pattern and actions of a rule. The association link represents an instance of the particular metamodel association - a link. It has no identifier, thus it can not be referenced. The association link is used as a part of the pattern. Then it adds the constraint to the condition - the existence of a link between two instances. The association link is used also to create or to delete a link between two instances.

Generalizations

It is specialized from Link

Attributes.

linkType:AssocLinkType - type of the association link. It may be *normal*, *create* or *delete*.

cardConstraint:CardConstaint [0..1] - the cardinality constraint. It may be set for a *normal* association link. Currently, the only cardinality constraint is *NOT* constraint. An association link having the NOT constraint is called a *NOT-link*. The semantics of the NOT-link is opposite to a normal association link without cardinality constraint. The condition of the pattern holds only if there is no link (of the specified association) which connects the two given instances.

Associations.

*[inh]*comment[*] - comments on this association link.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this association link.

*[inh]*annotation[*] - annotations added to this association link. Currently there may be only one annotation - *c* annotation. It adds semantics to the pattern. More on semantics of the patterns see Chapter 5.

*[inh]*owningRule[1] - the rule that owns the association link.

[inh]`from`[1] - the class element the association link is going from - the source of the association link. The association link actually is not directed. Therefore it is irrelevant whether the connected class element is a source or target.

[inh]`to`[1] - the class element the association link is going to - the target of the association link. The association link actually is not directed. Therefore it is irrelevant whether the connected class element is a source or target.

`assoc`[1] - the association the association link corresponds to. It must be a valid association that connects the classes that correspond to the source and the target class elements.

`sourceProp`[1] - the association end that is connected to the source class (the class corresponding to the source class element). It must be a member end of the association denoted by the `assoc` link.

`destProp`[1] - the association end that is connected to the target class (the class corresponding to the target class element). It must be a member end of the association denoted by the `assoc` link.

Notation. An association link is shown as a line that connects two class element symbols. Names of association ends are shown at the appropriate end of the line. If the association link is constrained or annotated, then the text of the constraint or annotation is shown in curly brackets at the middle of the line.

The *normal* association link is shown as a black solid line (see Fig. 23a). The *create* association link is shown as a red dotted line (see Fig. 23b). The *delete* association link is shown as a black dashed line (see Fig. 23c).

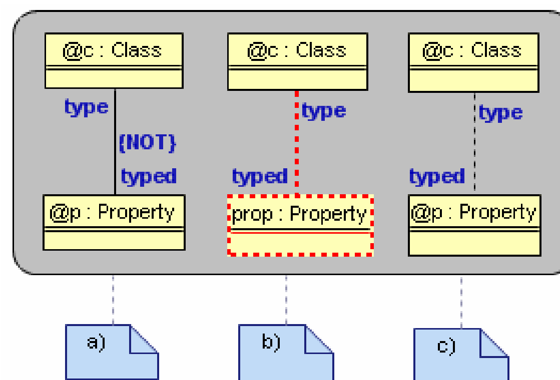

Fig. 23. Association link notation a) normal b) create c) delete

Semantics. An association link represents a particular link in the model. Each association link has the type specification – it corresponds to an association in the metamodel. The association must be a valid association that connects classes the class elements correspond to.

A *normal* association link is used as a constraint in the pattern. The pattern matches only if a link of the given type exists between instances denoted by the source and the target class elements. A *normal* association link may have a *NOT* constraint. Then it has the opposite semantics - the pattern matches only if there is no link of the given type between the instances

denoted by the source and the target class elements. A *normal* link may connect *normal*, *delete* and *loop variable* class elements.

A *create* association link is used to create a link of the given type between instances denoted by the source and the target class elements. A *create* association link may connect *normal*, *create* and *loop variable* class elements. If an association end of the appropriate association is ordered then the created link is added as the last one in the existing sequence of ordered links.

A *delete* association link is used as a *normal* association link in the pattern, except that it may not have the *NOT*-constraint. If the pattern matches then the appropriate link is deleted. It may connect *normal* and *loop variable* class elements.

The fragment of the MOLA metamodel that describes the facility for the assignment definition is shown in Fig 24.

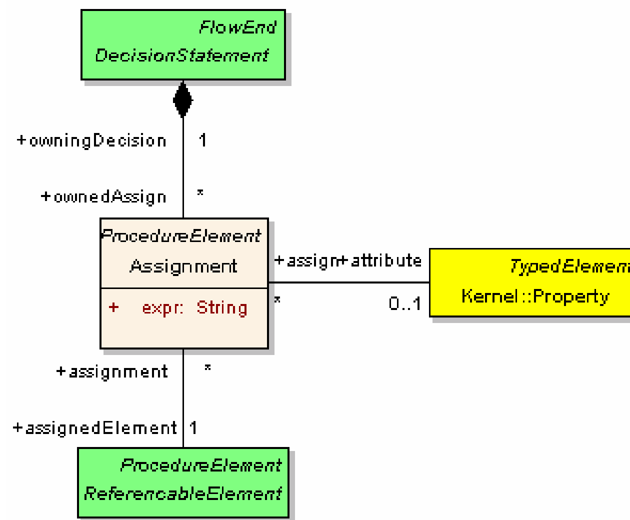

Fig. 24. Assignments in MOLA

3.29 Assignment

Brief overview. `Assignment` is a textual element that is used to set value of a referencable element or an attribute value of a particular instance. It may be in the action part of a text statement or in a class element.

Generalizations

It is specialized from `ProcedureElement`

Attributes.

`expr:String` – expression (see Chapter 4 for details) that is evaluated and the result is assigned to a referencable element or an attribute. The type of the expression is constrained by the definition of the referencable element or attribute.

```
<assignment-expression>::=<expression>
```

Associations.

[inh]`comment[*]` - comments on this assignment.

[inh]`owningProcedure`[0..1] - the MOLA procedure which owns this assignment.

`assignedElement`[1] – the referencable element whose value is being set, if the `attribute` link is not present. Otherwise it denotes the pointer to the instance whose attribute is set.

`owningDecision`[1] - the decision statement (rule or text statement) which owns the assignment.

`attribute`[0..1] - the attribute of the instance which must be set. The pointer to the instance is denoted by the `assignedElement` link. The attribute must be a valid attribute of the class that corresponds to the pointer.

Notation. Assignments are textual elements that may be contained by class elements and text statements (See Fig 22a, b and Fig. 15). An assignment is a string which consists of the left hand side that specifies the element whose value is set, the assignment sign ':='and the right hand side - an expression.

```
<assignment-notation>::=(<referencable-element-
name>|<attribute-specification>)':='<assignment-expression>
```

Semantics. The assignment is a textual element of the MOLA procedure. It is used to assign value to referencable elements or to the attribute of the instance denoted by a pointer. The assignment has two parts - the left hand side and the right hand side. The left hand side contains the element to be set. It may be an attribute specification, variable or parameter. If an assignment is within a class element then the left hand side may only be an attribute name of the corresponding class. If an assignment is within a text statement then the left hand side may be a variable name, parameter name or attribute specification. The attribute specification actually is a pointer (a class-typed variable or parameter or a class element) name, followed by an attribute name. A value may not be assigned to a pointer being a class element reference - it can receive its value only via pattern matching or instance creation. The right hand side of the assignment contains an expression of the appropriate type.

4 Expressions

MOLA is a graphical model transformation language, which has also textual elements. The most important part of textual elements in MOLA is *expressions*. They are used to describe constraints and to assign values to various elements - attributes of an instance, pointers, elementary variables and call parameters. Each expression has the result - a value of the particular type. The result may be a value of the primitive type (String, Integer or Boolean) or enumeration, or a pointer to an instance in the model. This section provides the description of expressions that are allowed in MOLA and the BNF-grammar that describes them.

4.1 Base elements

Base elements of expressions in MOLA are constants, elementary variables, pointers, navigations and attribute specifications. These are elements that hold values directly. There are four types of constants in MOLA:

- string constants (See Section 2.10)
- integer constants (See Section 2.10)
- boolean constants (See Section 2.10)
- enumeration constants (See Section 2.9)

An elementary variable (See Section 3.22) holds a value of the primitive type or enumeration. It is specified using the name of the elementary variable.

A pointer (See Section 3.22) points to an instance in the model. It is specified using the name of the pointer. The prefix '`@`' must be used before the name of the pointer. The prefix may be omitted only if the pointer is used in the constraint of the pattern and it points to the non-reference class element of the same pattern. If the pointer is used in a class element and it points to itself, then the keyword `self` must be used.

The navigation is similar construct to the OCL navigation. It denotes the set of instances obtained via navigable links. The navigation is specified using a pointer name and appropriate rolenames separated by dots. The pointer denotes the instance the navigation starts from. The rolename separated from pointer name by dot denotes the type of links the navigation is performed over. The rolename must be a valid name of the association end leading to the class from the class denoted by the pointer. A set of instances is obtained this way. Each next rolename in the navigation string denotes the next type of a link to navigate over. The result is a set of instances. The simplest navigation is a pointer. It is interpreted as a set that consists of one instance.

An attribute specification holds a value of the particular attribute of the instance. This instance is specified using the navigation which results to one instance. It may be a pointer or a navigation via `1` or `0..1` links. The navigation may be omitted if the expression is used in a class element and the attribute specification denotes the attribute of the instance this class element points to.

```
<elementary-variable>::=<elementary-variable-name>

<pointer>::=['@']<pointer-name> | 'self'

<navigation>::=<pointer>('.'<role-name>)*
```

```
<attribute-specification>::=[<navigation>'.']<atribute-name>
```

4.2 Pointer expression

The result of the pointer expression is a pointer. The pointer expression may be:

- pointer
- `NULL` keyword denotes the value of the pointer which does not point to any instance.
- down-cast. The pointer may be down-casted accordingly to the class hierarchy defined in the metamodel. The down-cast is specified using the full qualified class name. The upcast is performed automatically in MOLA and does not need special specification.

```
<pointer-expression>::=<pointer> | 'NULL' | <type-cast>

<type-cast>::=<full-class-name>'('<pointer>')'
```

4.3 Set expression

The result of the set expression is a set of instances. The set expression is specified using navigation.

```
<set-expression>::=<navigation>
```

4.4 Enumeration expression

The result of the enumeration expression is an enumeration value. The enumeration expression may be:

- enumeration constant
- attribute specification. The attribute type must be the enumeration.
- elementary variable whose type is the enumeration.
- `toEnum` function. The function converts the result of a string expression to the enumeration constant. Note, the exact enumeration type of the result is determined by the context the function is used.

```
<enum-expression>::=

      <enumeration-literal-name> | <attribute-specification> |

      <elementary-variable> | 'toEnum('<string-expression>')'
```

4.5 Integer expression

The result of the integer expression is an integer value. The integer expression base elements are:

- integer constant
- attribute specification. The attribute type must be Integer.
- elementary variable whose type is Integer.
- Integer functions:
    - o `toInteger` function converts the result of a string expression to an integer value.
    - o `size` function returns the size of the result of a string expression or the size of the set specified by a set expression.
    - o `indexOf` function returns the starting position of the substring in the string. Both are specified by a string expression. Note that character

numbering starts form 1 in MOLA for strings. First argument specifies the substring to find, the second – the string value where to search.

Integer expression base elements may be used in more complex integer expressions. Standard arithmetical operations may be applied – addition (+), subtraction (-) and multiplication (*). The parenthesis may be also used.

```
<integer-expression>::=<int-factor> | <int-expression>(‘+‘|‘-
‘)<int-factor>

<int-factor>::=<int-term> | <int-factor> ‘*‘ <int-term>

<int-term>::=

      <integer-constant> | ‘(‘<integer-expression>‘)‘ |

      <int-operation> | <attribute-specification> |
<elementary-variable>

<int-operation>::=

      ‘toInteger(‘<string-expression>‘)‘ |

      ‘size(‘<string-expression>) |

      <set-expression>‘->size()‘ |

      ‘indexOf(‘<string-expression>‘,‘<string-expression>‘)‘
```

4.6 String expression

The result of the string expression is a string value. The string expression base elements are:

- string constant
- attribute specification. The attribute type must be String.
- elementary variable whose type is String.
- String functions:
  - `toString` function converts the result of an integer, an enumeration or a simple boolean expression to a string value.
  - `substring` function returns the substring of the string. The substring is specified using the starting character and ending character positions in the string. The first argument is a source string specified by the result of a string expression. The second argument is the starting character position specified by the result of an integer expression. The third argument is the ending character position specified by the result of an integer expression. If the third argument is omitted, then the last character of the string is taken as ending character.
  - `toLower` function converts all characters of the string value specified by the result of the string expression to the lowercase characters.
  - `toUpper` function converts all characters of the string value specified by the result of a string expression to the uppercase characters.

String expression base elements may be used in more complex string expressions. The concatenation (+) operation may be applied.

```
<string-expression>::=<str-factor> | <string-
expression>‘+‘<str-factor>
```

<str-factor>::=

<string-constant> | <str-operation> |

<attribute-specification> | <elementary-variable>

<str-operation>::=

'toString('(integer-expression> | <enum-expression> | <bool-expression>)**')'** |

'substring('<string-expression>**','**<integer-expression>[**','**<integer-expression>]**')'** |

'toLower('<integer-expression>**')'** | **'toUpper('**<integer-expression>**')'**

4.7 Simple boolean expression

The result of the simple boolean expression is a boolean value. The simple boolean expression may be:

- boolean constant
- attribute specification. The attribute type must be Boolean.
- elementary variable whose type is Boolean.
- `toBoolean` function converts the result of a string expression to a boolean value.
- type checking operations:
  - `isTypeOf` operation examines if the instance denoted by the pointer is of a particular type. This operation returns *true* if the instance is exactly of the specified class.
  - `isKindOf` operation examines if the instance denoted by the pointer is of a particular kind. This operation returns *true* if the instance is of specified class or its subclass.
- emptiness verification operations:
  - `isEmpty` operation examines if the particular attribute of an instance specified by an attribute specification is not set or the set specified by a set expression contains no elements. This operation returns *true* if the attribute is not set or the set is empty respectively.
  - `notEmpty` operation examines if the particular attribute of an instance specified by an attribute specification is set or the set specified by a set expression contains any element. This operation returns *true* if the attribute is set or the set is not empty respectively.

<bool-expression>::=

<boolean-constant> | <bool-operation> |

<attribute-specification> | <elementary-variable> |

<bool-operation>::=

'toBoolean('<string-expression>**')'** | <null-operation> | <type-check-operation>

<type-check-operation>::=

```
      <pointer>’.isTypeOf(’<full-class-name>’)’ |

      <pointer>’.isKindOf(’<full-class-name>’)’

<null-operation>::=

      (<attribute-specification>|<set-expression>)’->isEmpty()’
|

      (<attribute-specification>|<set-expression>)’-
>notEmpty()’
```

4.8 Simple expressions summary

Thus, there are six expression types: the integer expression, the string expression, the enumeration expression, the simple boolean expression, the pointer expression and the set expression.

```
<expression>::=

      <integer-expression> | <string-expression> | <enum-
expression> |

      <bool-expression> | <pointer-expression> | <set-
expression>
```

4.9 Simple constraints

Simple constraints may be described in MOLA using expressions described above and relational symbols = (equal), <> (unequal), < (less, strong subset), <= (less or equal, weak subset), > (greater, strong superset), >= (greater or equal, weak superset). If forms *the condition*, and the result of a simple constraint is *true* if this condition holds.

The = and <> symbols are used to compare two expressions of the same type. The condition holds if results of both expressions are equal or unequal respectively.

The <, <=, >, >= are used to compare the results of an integer expressions. The condition holds if the inequality is correct. These symbols may be used also to compare the results of two set expressions. The < symbol denotes that the set specified by the left expression is a subset of the set specified by the right expression. The <= symbol denotes that both sets may be also equal. > and >= symbols are used similar, only the set specified by the right expression is a subset of the set specified by the left expression.

The simple boolean expression also may be used as simple constraint. The condition holds if the result of a simple boolean expression is *true*.

```
<simple-constraint>::=

      <bool-expression>| <expression>(’=’|’<>’)<expression>|

      <integer-expression> (’>’|’<’|’>=’|’<=’)<integer-
expression> |
```

<set-expression>(’**>**’|’**<**’|’**>=**’|’**<=**’)<set-expression>

4.10 Constraint expression

A textual constraint is built using the constraint expression in MOLA. The constraint expression is a boolean expression. The standard operations may be used - `or`, `and`, `not`. Parentheses are also permitted. The operands of the constraint expressions are simple constraints.

<constraint-expression>::=<bool-factor> | <constraint-expression>’ **or** ’<bool-factor>

<bool-factor>::=<bool-term> | <bool-factor> ‘ **and** ’ <bool-term>

<bool-term>::=

<simple-constraint> | ’**(**’<constraint-expression>’**)**’ | ’**not** ’<constraint-expression>

5 Additional remarks on syntax and semantics

This Chapter contains the description of the semantics of patterns in MOLA and some additional remarks on semantics of the rule.

5.1 Pattern semantics

This section describes semantics of *the pattern* in MOLA. Patterns are used in graphical conditional statements - *rules*. A pattern forms the graphical part of the condition of a rule. A pattern specifies the fragment of the model that must be found. The model consists of instances, which may have links connecting them and values of attributes. Each instance corresponds to a class defined in the metamodel, as well as, each link corresponds to an association. Instances may have attribute values set. Thus, a pattern must contain information on instances, links and attribute values. Actually, a pattern describes what instances must be found. The process that is used to find the corresponding instances is called a *pattern matching*.

A pattern is defined by means of class elements (see Section 3.28) and association links (see Section 3.28) in MOLA.

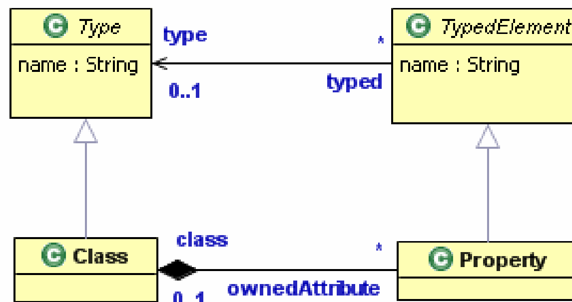

Fig. 25. The metamodel used in examples

Normal, *delete* and *loop variable* class elements (pattern elements) form a pattern. A pattern element represents an instance of a particular metamodel class. A *non-reference* pattern element specifies that the condition of the rule may be *true* only if there is an instance which matches to the features defined by it. More precisely, if there is an instance that corresponds to the specified class. See an example Fig. 26. The metamodel that is used for examples of this section is shown in Fig. 25. The pattern matches if there is an instance of the type `Class` - there is a class in the model. The condition of the rule may be expressed using *many-sorted first-order logic* (`Ec:Class`).

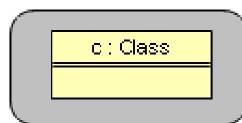

Fig. 26. Pattern example

Note that if there is more than one matching instance then first matched instance is chosen. If there is more than one pattern element in a pattern then all appropriate instances must exist.

See an example Fig. 27. The pattern matches if there is an instance of the type `Class` and an instance of the type `Property` - there is a class and a property in the model. (`Ec:Class ∧ Ep:Property`).

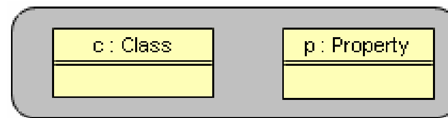

Fig. 27. Pattern example

Note that names of the pattern elements that are used in the same pattern must be unique. A pattern element may contain a constraint. The constraint is a Boolean expression (see Chapter 4 for details) that must evaluate to *true* for an instance in order to match it. A constraint may be used to constrain values of attributes. See an example Fig. 28. The pattern matches if there is an instance of the type `Class` whose "name" attribute value is "Person" and an instance of the type `Property` – there is a class *Person* and a property in the model. (`Ec:Class (name(c)='Person') ∧ Ep:Property`).

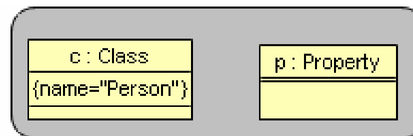

Fig. 28. Pattern example

Normal and *delete* association links (pattern links) are used in pattern to add an additional constraint to instances that are being matched. It defines that a link between appropriate instances must exist in order to match them. See an example Fig. 29. The pattern matches if there is a class *Person* which has an attribute in the model. (`Ec:Class (name(c)='Person' ∧ Ep:Property (ownedAttribute(p,c))`).

Fig. 29. Pattern example

A pattern may contain also *reference* pattern elements. A reference pattern element represents already known instance. It is used to examine particular features of that instance or to add additional constraint to instances that are being matched. See an example Fig. 30. The pattern matches if there is a type set for the already known property in the model. (`Et:Type (type(t,@p))`).

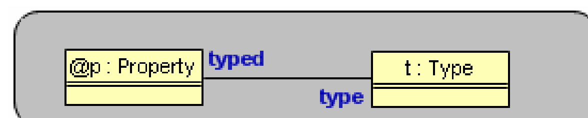

Fig. 30. Pattern example

A *normal non-reference* class element which has a *NOT* cardinality constraint (*NOT-element*) specifies that the condition of the rule may be *true* only if there is no instance which matches to the features defined by it. See an example Fig. 31. The pattern matches if there is a class which has no *name* attribute in the model. (`Ec:Class ( ⌐Ep:Property (ownedAttribute(p,c) ∧ name(p)='name'`)). Two *NOT-elements* may not be linked by an association link.

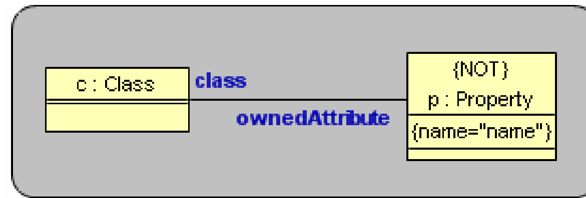

Fig. 31. Pattern example

A *normal* association link which has a *NOT* cardinality constraint (*NOT-link*) specifies that the condition of the rule may be *true* only if there is no link of the given type between instances denoted by source and target pattern elements. See an example Fig. 32. The pattern matches if there is a property which is not typed by the already known type instance in the model. (`Ep:Property (⌐typed(p,@t))`). A *NOT-link* may not be connected to a *NOT-element*.

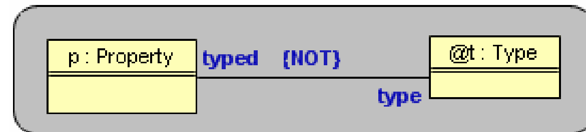

Fig. 32. Pattern example

Pattern elements that are linked by pattern links form a *pattern fragment*. Also, if two parts of a pattern fragment are linked by a *c-link* (see later) or *NOT-link* only, they are considered to be separate pattern fragments. The pattern may contain any number of pattern fragments. A pattern matches if all pattern fragments match. A pattern matching is a performance "bottleneck" of the implementation of transformation languages. Therefore it is strongly recommended to use annotations in MOLA. Each pattern fragment must contain an *initial* pattern element. It may be a *reference* class element or a *non-reference* class element that is annotated using *single* or *start* annotation to improve the performance of the pattern matching in MOLA. The *single* and *start* annotations denote that the pattern matching must start from this pattern element. Thus there can be only one annotated pattern element in a pattern fragment. The *single* annotation denotes also that only the first instance found must be examined. 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 pattern element, the constraint is evaluated on the chosen instance, and if it is *false* the pattern fails. The *start* annotation means that the pattern matching starts from the annotated pattern element and if the first found instance does not satisfy constraints then the next instance is sought, and so on. In a loophead the *start* annotation may not be used in the pattern fragment which contains the *loop variable*. If a pattern fragment contains a *reference* pattern element then the annotated element is not necessary in this fragment. The search always starts from this pattern element. If there is more than one reference pattern element, an arbitrary one is chosen as the first, such

situation should be avoided whenever possible. The pattern fragment which contains the *loop variable* may be without an annotated or reference class element. Then the *loop variable* is used as the initial pattern element.

To sum up, each pattern fragment must contain one *initial* pattern element - reference, *single* or *start*. Reference pattern elements may be more than one, if there is a *single* or *start* pattern element, then namely this annotated element is used as *initial*, but not references. If the pattern fragment contains several reference pattern elements, but no *single* or *start*, it is recommended to split up the fragment using *c-links*, e.g., to decide from which of the reference pattern elements a natural match should start and mark pattern links going to other references as *c-links* (these pattern links then mean additional constraints during the match, but other reference pattern elements themselves are treated as separate trivial pattern fragments). Hence, each pattern fragment is treated as a *rooted tree*, which is traversed from the root (*initial* pattern element) in all possible directions during the match.

A pattern link may have also an annotation - *c*. Then this link is called a *c-link*. The meaning of *c-link* is "check the presence of a link". It is used to avoid situations when a pattern fragment contains a closed loop or there are more then one potential *initial* pattern element (no annotated pattern elements and more then one reference pattern element). The main goal of *c-links* is to make a rooted tree or several rooted trees from a pattern fragment.

A *loop variable* may be not the initial pattern element in the rooted tree of the pattern fragment. Then it is reachable by association links from the initial pattern element. The order in which the corresponding instances are traversed in the loop may be determined by the appropriate association end (in the direction towards the loop variable). If it is ordered then instances are traversed in the order determined by this ordering. Otherwise, the order is undetermined.

For example, if a loop variable is linked to a reference pattern element (which serves as the initial pattern element) and this association link corresponds to an association with an ordered end at the loop variable then instances of the loop variable are traversed in the order defined by this association. In particular, normally it is the order in which the appropriate links have been created.

5.2 Action part of the rule

The action part of the rule is executed if the condition of the rule holds, actually, if the pattern matches. Matched instances may be referenced in the action part using names of corresponding pattern elements.

The order actions are performed in a rule is:

- Creation of instances. It is specified using *create* class elements.
- Creation of links. It is specified using *create* association links.
- Assignment of attribute values. It is specified using assignments in class elements.
- Deletion of links. It is specified using *delete* association links.
- Deletion of instances. It is specified using *delete* class elements.

Class 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.