XMF Reference Manual

Xactium Ltd

December 19, 2003

Contents

1 Introduction

XMF is a general engine that provides technolgies underlying a range of modelling methods and techniques. The basic XMF engine is a virtual machine that runs XMF VM code. The basic VM code runs in terms of a minimal language which is a subset of languages such as MOF. This basic language is self aware and is used as a basis for language definition. XMF is provided with a fairly extensive language, XMOF, which is both an instance and an extension of the basic machine language. XMF is not limited to this language, however, other languages can be defined in terms of the basic language and then loaded, making XMF very flexible. XMOF is also self aware and can be used as a basis for language engineering since it provides a parser, compiler and an interface to the XMF VM through various packages.

This document is the reference manual for the Xactium Modelling Framework XMF. It defines XMOF in terms of a collection of packages. This manual is not a tutorial for XMF although many of the packages and classes are documented in terms of examples. The manual has been mechanically generated from the XMF source code through the Doc package.

2 Installing and Running XMF

The XMF system is a collection of directories under a root directory XMFHOME. You can install XMF anywhere. The directory XMFHOME/bin contains scripts that control the various ways in which XMF can be executed. Each script takes a number of command line arguments, the first of which is always the path XMFHOME. Typically a script will start one of the saved images in XMFHOME/Images. A saved image is a resumable snapshot of an XMF session. You can create your own images at any stage during a session. The scripts are:

xmf This is used to run a very minimal XMF image. You must supply this script with XMFHOME and a binary file containing XMF machine instructions. The image is started, the machine instructions are loaded and executed.

xmfe This is used to run an XMF command interpreter. The evaluator image contains packages to parse and interpret commands. You must supply this script with XMFHOME. Optionally, you can supply this script with a binary file which is loaded and run.

xmfc This is used to run the XMF compiler from the command line or to start an interactive session with the XMF compiler. You must supply this script with XMFHOME. Optionally you may supply this script with an XMF source file. The file should have an '.xmf' extension, although you need not supply the extension as part of the a command line argument. If you supply a file then it is compiled and the compiler terminates. If you do not supply a file then an XMF command interpreter is started and you can interactively compile files using the Compiler package.

xmft This script starts an XMF image and runs all the tests in XMFHOME/Test. The tests should be fairly self explanatory. You must supply this script with XMFHOME.

make This is used to reconstruct the XMF images in XMFHOME/Images.

3 NameSpace `Root`

context: `Root`
overview:

> The root name space contains all XMF definitions. Adding a named element to this name space makes it globally available. XMF is initialized with a number of named elements in this name space. Typically you will add your own elements here when developing new name spaces containing your definitions. Although named elements are added to a name space using the ’add/1’ operation, the preferred (and more declarative) way of adding definitions to name spaces (including packages and classes) is to use the ’context’ construct. Typically the ’context’ construct appears at the top level of a file or equivalent collection of definitions.

example:

```
context Root
  @Operation globalOp(args)
    // Some global definition that refers to args
    // and other global definitions or other names in
    // scope.
  end
```

4 Package `Aspects`

context: `Root`
overview:

This package defines classes that support XMF aspects. An aspect is a collection of definitions that are added to model elements. An aspect allows you to group structure and behaviour that relate to a collection of (possibly otherwise unrelated) classes. An aspect contains a collection of aspect components. Each aspect component is a name space and a collection of named elements for that name space. By adding an aspect component to an aspect, each component causes the named elements to be added to the component name space. For example, consider defining an aspect that allows certain types of EMOF element to be translated to HTML. The HTML aspect will contain class definitions for just the attributes and operations that need to be added to the existing EMOF classes.

4.1 Class `Aspect`

context: `Root::Aspects`
overview:

An aspect contains a collection of definitions that are added to existing model elements. The definitions take the form of class definitions where the names are paths to existing classes and the component definitions are attributes and operations.

constructor: `Aspect(name,components)`

Each element of the sequence components is a ComponentDef.

syntax:

```
Aspect ::= '@Aspect' Name Component* 'end'.
Component ::= ClassDef.
ClassDef ::= '@Class' Path Exp 'end'.
```

interface:

```
@Class Aspect extends Sugar
  @Attribute components : Seq(ComponentDef) end
  @Attribute name : Name end
  @Operation desugar end
  @Operation pprint end
  @Operation toString end
end
```

4.2 Class `Class`

context: `Root::Aspects`
overview:

A class component definition in an aspect allows an existing class to be extended with new elements such as attributes and operations. The name of the class can be a full path to allow the existing class to be referenced.

constructor: `Class(path,elements)`

syntax:

```
       Class ::= '@Class' Path Exp* 'end'.
```

interface:

```
@Class Class extends ComponentDef
  @Attribute elements : Seq(Performable) end
  @Attribute path : Path end
  @Operation desugar end
  @Operation pprint end
  @Operation toString end
end
```

4.3 Class `ComponentDef`

context: `Root::Aspects`
overview:

> ComponentDef is the abstract superclass of all aspect component definitions.

interface:

```
@Class ComponentDef isabstract extends Sugar
end
```

5 Package `Semantics`

context: `Root::Aspects`

5.1 Class `Aspect`

context: `Root::Aspects::Semantics`

interface:

```
@Class Aspect extends Container,NamedElement
  @Attribute components : Set(Component) end
  @Operation add end
  @Operation toString end
  @Operation test end
end
```

5.2 Class `Component`

context: `Root::Aspects::Semantics`

interface:

```
@Class Component extends Object
  @Attribute contents : Set(NamedElement) end
  @Attribute nameSpace : NameSpace end
  @Operation add end
  @Operation perform end
  @Operation toString end
end
```

6 NameSpace `AllAspects`

context: `Root`

7 Package `Assembler`

context: `Root`
overview:

> Once compiled, the resulting instructions must be assembled to produce a code box. The code box can then be linearized to produce a binary stream that will re-create the code box at load time in the machine.

operation: `assemble(resourceName,instrs,locals,source)`

> Produces a code box from a sequence of instructions, the required number of locals and the source code represented as a string. The source code is optional and may be supplied as the empty string.

7.1 Class `CodeBox`

context: `Root::Assembler`
overview:

> A code box is a modelled eqivalent of the machine data structure that contains executable machine instructions. A code box can be serialized to a file which will cause the machine data structure to be constructed when the file is loaded or can be transformed directly into a machine function of 0 arguments.

constructor: `CodeBox(name)`

> Constructs a code box with the given name.

interface:

```
@Class CodeBox extends Resource
  @Attribute code : Seq(Element) end
  @Attribute constants : Seq(Element) end
  @Attribute locals : Integer end
  @Attribute name : String end
  @Attribute source : String end
  @Operation addInstr end
  @Operation constOffset end
  @Operation codeBoxes end
  @Operation init end
  @Operation pprint end
  @Operation ppSource end
  @Operation ppConstants end
  @Operation ppCode end
  @Operation symbolOffset end
  @Operation setResourceName end
  @Operation setCodeBoxNames end
  @Operation toMachineFun end
  @Operation toMachineCodeBox end
  @Operation toMachineCode end
  @Operation toMachineValue end
  @Operation toMachineArray end
  @Operation writeValue end
  @Operation writeSymbol end
  @Operation writeString end
  @Operation writeInteger end
  @Operation writeInstrs end
  @Operation writeFile end
  @Operation writeCode end
  @Operation writeArray end
end
```

7.1.1 Operation `addInstr`

context: `Root::Assembler::CodeBox`
overview:

> Instructions are added to the head of the code stream which must be reversed before use.

parameter: `instr:Instr`

> The instruction to add to the code box.

7.1.2 Operation `constOffset`

context: `Root::Assembler::CodeBox`
overview:

> Find the offset in the constants are of the code box for a given value. A given value should not occur more than once in the constants area of a code box.

7.1.3 Operation `codeBoxes`

context: `Root::Assembler::CodeBox`
overview:

> A code box may contain sub-code-boxes as part of its constants area. typically the sub-code-boxes contain instructions for functions that are created when the parent is executed. This operation returns all the code boxes in a parent.

7.1.4 Operation `ppSource`

context: `Root::Assembler::CodeBox`
overview:

> Code boxes contains source code strings. The string may be empty if the compiler has not dumped source code into the code box. You can request the compiler save source code which will be used in displaying diagnostics.

7.1.5 Operation `ppConstants`

context: `Root::Assembler::CodeBox`
overview:

> The constants of a code box pretty printed to an output channel. Constants live at particular offets in the constants area.

7.1.6 Operation `ppCode`

context: `Root::Assembler::CodeBox`
overview:

> The instructions of a code box written to an output channel.

7.1.7 Operation `symbolOffset`

context: `Root::Assembler::CodeBox`
overview:

> Symbols are objects are are therefore not = to each other even if they have the same name. This operation finds symbols in the constants area based on the equality of their names.

parameter: `symbol:Symbol`

> The symbol that is to be added to the constants area of the code box.

7.1.8 Operation `setCodeBoxNames`

context: `Root::Assembler::CodeBox`
overview:

> Some code boxes are named. For example because they correspond to named class operations. If a code box is not given a name then it it is allocated a unique name here.

7.1.9 Operation `toMachineFun`

context: `Root::Assembler::CodeBox`
overview:

> Translates a code box to a machine function suitable for applying to the appropriate number of arguments. The function can then be called or serialized as appropriate. It is assumed that the correct number of locals has been set in the code box to support at least the required number arguments.

parameter: `arity:Integer`

> The number of arguments required by the function.

parameter: `dynamics:Seq(Element)`

> The dynamics of the function. The dynamics define the non-local variables that can be referenced by the function body. The dynamics are supplied as a sequence of pairs of the form:

```
Seq{<INTEGER> | <ELEMENT>}
```

> where the first element is a type code and the second is a dynamic value. If the first element is 1 then the second element should be a pair containing a symbol and a value. If the forst element is 2 then the second element should be a hashtable that associates symbols (the names) with values.

7.1.10 Operation `toMachineArray`

context: `Root::Assembler::CodeBox`
overview:

> Translates a sequence of values to an array. This is necessary to represent code box constants (for example) as a machine data structure.

parameter: `values:Seq(Element)`

> A sequence of values to be translated to an array.

7.2 Operation `assemble`

context: `Root::Assembler`

8 Package `Clients`

context: `Root`
overview:

> This package contains definitions relating to XMF clients. An XMF client is a (possibly external) program that communicates with the XMF engine.

9 Package `ClassDiagrams`

context: `Root::Clients`
overview:

> This package extends Clients::Diagrams with features that model class diagrams.

9.1 Class `AttributeEdge`

context: `Root::Clients::ClassDiagrams`

interface:

```
@Class AttributeEdge extends Edge
  @Attribute attribute : Attribute end
end
```

9.2 Class `ClassNode`

context: `Root::Clients::ClassDiagrams`

interface:

```
@Class ClassNode extends Node
  @Attribute class : Class end
  @Attribute name : Text end
  @Operation attBox end
  @Operation addRightClickMenuItems end
  @Operation addNameDaemon end
  @Operation addDaemons end
  @Operation class end
  @Operation calculateBoxHeight end
  @Operation firePropertyChanged end
  @Operation fireNameChanged end
  @Operation fire end
  @Operation nameBox end
  @Operation opsBox end
end
```

9.3 Class `ClassDiagram`

context: `Root::Clients::ClassDiagrams`

interface:

```
@Class ClassDiagram extends Diagram
  @Attribute nodeTable : Table end
  @Attribute package : Package end
  @Operation addNode end
  @Operation classNameExists end
  @Operation defineNodeToolGroups end
  @Operation defineEdgeToolGroups end
  @Operation findNode end
  @Operation initPackage end
  @Operation initNodes end
  @Operation initInheritanceEdges end
  @Operation initEdges end
  @Operation initAttributeEdges end
  @Operation incClassNameCounter end
  @Operation newNode end
  @Operation newClassNode end
  @Operation newClassName end
  @Operation newEdge end
  @Operation newInheritanceEdge end
  @Operation newAttributeEdge end
end
```

9.4 Class `InheritanceEdge`

context: `Root::Clients::ClassDiagrams`

interface:

```
@Class InheritanceEdge extends Edge
end
```

10 Package `Diagrams`

context: `Root::Clients`
overview:

This package defines a model for a diagram client.

10.1 Class `Box`

context: `Root::Clients::Diagrams`

interface:

```
@Class Box extends Display
  @Attribute cornerCurve : Integer end
  @Attribute displays : Seq(Display) end
  @Attribute height : Integer end
  @Attribute id : Element end
  @Attribute width : Integer end
  @Attribute x : Integer end
  @Attribute y : Integer end
  @Operation addToDisplays end
  @Operation add end
  @Operation cornerCurve end
  @Operation delete end
  @Operation height end
  @Operation id end
  @Operation new end
  @Operation resize end
  @Operation setId end
  @Operation toString end
  @Operation width end
  @Operation x end
  @Operation y end
end
```

10.2 Class `ChangeEvent`

context: `Root::Clients::Diagrams`

interface:

```
@Class ChangeEvent isabstract extends Object
  @Attribute diagram : Diagram end
  @Attribute mapping : Mapping end
  @Attribute target : Element end
  @Operation diagram end
  @Operation mapping end
  @Operation target end
end
```

10.3 Class `Circle`

context: `Root::Clients::Diagrams`

interface:

```
@Class Circle extends Display
end
```

10.4 Class `Diagram`

context: `Root::Clients::Diagrams`

interface:

```
@Class Diagram extends Object
  @Attribute graph : Graph end
  @Attribute id : Element end
  @Attribute name : String end
  @Operation addNode end
  @Operation addEdge end
  @Operation delete end
  @Operation defineToolGroups end
  @Operation defineNodeToolGroups end
  @Operation defineEdgeToolGroups end
  @Operation fire end
  @Operation findElement end
  @Operation id end
  @Operation name end
  @Operation newNode end
  @Operation newEdge end
  @Operation new end
  @Operation setId end
  @Operation toString end
end
```

10.5 Class `Display`

context: `Root::Clients::Diagrams`

interface:

```
@Class Display isabstract extends Object
end
```

10.6 Class `DefaultNode`

context: `Root::Clients::Diagrams`

interface:

```
@Class DefaultNode extends Node
end
```

10.7 Class `DiagramIdError`

context: `Root::Clients::Diagrams`

interface:

```
@Class DiagramIdError extends Exception
  @Attribute id : Element end
  @Attribute value : Element end
end
```

10.8 Class `DefaultEdge`

context: `Root::Clients::Diagrams`

interface:

```
@Class DefaultEdge extends Edge
end
```

10.9 Class `Edge`

context: `Root::Clients::Diagrams`

interface:

```
@Class Edge isabstract extends Object
  @Attribute id : Element end
  @Attribute labels : Set(Label) end
  @Attribute sourceHead : Integer end
  @Attribute syoff : Integer end
  @Attribute sxoff : Integer end
  @Attribute source : Port end
  @Attribute targetHead : Integer end
  @Attribute tyoff : Integer end
  @Attribute txoff : Integer end
  @Attribute target : Port end
  @Attribute wayPoints : Set(WayPoint) end
  @Operation addToWayPoints end
  @Operation addToLabels end
  @Operation add end
  @Operation delete end
  @Operation findElement end
  @Operation fire end
  @Operation id end
  @Operation newWayPoint end
  @Operation new end
  @Operation repositionEdgeEnds end
  @Operation sourceHead end
  @Operation syoff end
  @Operation sxoff end
  @Operation source end
  @Operation setId end
  @Operation sourceReconnected end
  @Operation toString end
  @Operation targetHead end
  @Operation tyoff end
  @Operation txoff end
  @Operation target end
  @Operation targetReconnected end
end
```

10.10 Class `Group`

context: `Root::Clients::Diagrams`

interface:

```
@Class Group extends Display
end
```

10.11 Class `Graph`

context: `Root::Clients::Diagrams`

interface:

```
@Class Graph extends Object
  @Attribute edges : Set(Edge) end
  @Attribute nodes : Set(Node) end
  @Operation addNode end
  @Operation addEdge end
  @Operation delete end
  @Operation fire end
  @Operation findNodeElement end
  @Operation findEdgeElement end
  @Operation findElement end
  @Operation new end
  @Operation toString end
end
```

10.12 Class `Label`

context: `Root::Clients::Diagrams`

interface:

```
@Class Label extends Object
  @Attribute attachedTo : String end
  @Attribute editable : Boolean end
  @Attribute id : Element end
  @Attribute rely : Integer end
  @Attribute relx : Integer end
  @Attribute text : String end
  @Operation attachedTo end
  @Operation editable end
  @Operation id end
  @Operation move end
  @Operation new end
  @Operation rely end
  @Operation relx end
  @Operation setId end
  @Operation toString end
  @Operation text end
end
```

10.13 Class `Line`

context: `Root::Clients::Diagrams`

interface:

```
@Class Line extends Display
end
```

10.14 Class `MenuItem`

context: `Root::Clients::Diagrams`

interface:

```
@Class MenuItem extends Object
  @Attribute action : Operation end
  @Attribute name : String end
  @Operation new end
  @Operation select end
  @Operation toString end
end
```

10.15 Class `Mapping`

context: `Root::Clients::Diagrams`
overview:

This class defines a command interpreter over diagrams. It can be used as the basis for defining event driven mappings. The mapping handles the following output commands by sending them in the appropriate format to the client connected to the output channel:

delete(diagramId,elementId)

moveCommand(diagramId,elementId,x,y)

newBoxCommand(diagramId,nodeId,box)

newDiagramCommand(diagram)

newEdgeCommand(diagramId,edge)

newEdgeTextCommand(diagramId,edgeId,label)

newNodeCommand(diagramId,parentId,node)

newPortCommand(diagramId,nodeId,port)

newRightClickMenu(diagramId,ownerId,menuId,option)

newTextCommand(diagramId,parentId,text,x,y,editable)

newToolCommand(diagramId,name)

newToolGroupCommand(diagramId,groupId,toolName,isEdge)

newWayPointCommand(diagramId,wayPoint)

resizeCommand(diagramId,elementId,width,height)

repositionEdgeEndsCommand(diagramId,edgeId,sxoff,syoff,txoff,tyoff)

setEdgeSourceCommand(diagramId,edgeId,portId,sxoff,syoff,txoff,tyoff)

setEdgeTargetCommand(diagramid,edgeId,portId,sxoff,syoff,txoff,tyoff)

setTextCommand(diagramId,textId,newText)

The mapping defines abstract operations to handle the following incoming events from the client. In each case the event is a sequence of strings and integers. Each event has a fixed arity. Raising an event in the mapping causes the appropriate diagram element (usually the containing element) to be sent a message with the supplied arguments + the mapping. The diagram element is then responsible for updating the diagram model and using the commands listed above to update the client. Users of the mapping should always raise events and not perform the commands directly. New types of diagram element can handle the events in different ways. New types of diagram mapping can extend the interface of events that can be raised.

deleteWaypoint(diagramId,elementId)

edgeSourceReconnected(diagramId,edgeId,portId,sxoff,syoff,txoff,tyoff)

edgeTargetReconnected(diagramId,edgeId,portId,sxoff,syoff,txoff,tyoff)

moveEdgeText(diagramId,textId,x,y)
moveNode(diagramId,nodeId,x,y)
moveWaypoint(diagramId,wayPointId,x,y)
newDiagram(name)
newEdge(diagramId,edgeType,sourcePort,sxoff,syoff,targetPort,txoff,tyoff)

newNode(diagramId,nodeType,x,y)
newWaypoint(diagramId,edgeId,index,x,y)
repositionEdgeEnds(diagramId,edgeId,sxoff,syoff,txoff,tyoff)
resizeNode(diagramId,nodeId,width,height)
rightClickMenuSelected(diagramId,ownerId,option)
textChanged(diagramId,textId,newText)

interface:

```
@Class Mapping extends Object
  @Attribute debug : Boolean end
  @Attribute diagrams : Set(Diagram) end
  @Attribute idTable : Table end
  @Attribute input : TokenInputChannel end
  @Attribute output : OutputChannel end
  @Operation delete end
  @Operation deleteCommand end
  @Operation debug end
  @Operation edgeTargetReconnected end
  @Operation edgeSourceReconnected end
  @Operation find end
  @Operation moveWayPoint end
  @Operation moveNode end
  @Operation moveEdgeText end
  @Operation moveCommand end
  @Operation newWayPoint end
  @Operation newNode end
  @Operation newEdge end
  @Operation newWayPointCommand end
  @Operation newToolGroupCommand end
  @Operation newToolCommand end
  @Operation newTextCommand end
  @Operation newRightClickMenuCommand end
  @Operation newPortCommand end
  @Operation newNodeCommand end
  @Operation newEdgeTextCommand end
  @Operation newEdgeCommand end
  @Operation newDiagramCommand end
  @Operation newBoxCommand end
  @Operation newId end
  @Operation newDiagram end
  @Operation readTextChanged end
  @Operation readRightClickMenuSelected end
  @Operation readResizeNode end
  @Operation readRepositionEdgeEnds end
  @Operation readNewWayPoint end
  @Operation readNewEdge end
  @Operation readNewNode end
  @Operation readMoveWayPoint end
  @Operation readMoveNode end
  @Operation readMoveEdgeText end
  @Operation readEdgeTargetReconnected end
  @Operation readEdgeSourceReconnected end
  @Operation readDeleteWayPoint end
  @Operation readEvent end
  @Operation rightClickMenuSelected end
  @Operation resizeNode end
  @Operation repositionEdgeEnds end
  @Operation resizeCommand end
  @Operation repositionEdgeEndsCommand end
  @Operation setTextCommand end
  @Operation setEdgeTargetCommand end
  @Operation setEdgeSourceCommand end
  @Operation toString end
  @Operation textChanged end
  @Operation writeCommand end
end
```

10.16 Class `Node`

context: `Root::Clients::Diagrams`

interface:

```
@Class Node isabstract extends Object
  @Attribute displays : Seq(Display) end
  @Attribute height : Integer end
  @Attribute id : Element end
  @Attribute ports : Set(Port) end
  @Attribute rightClickMenu : PopupMenu end
  @Attribute width : Integer end
  @Attribute x : Integer end
  @Attribute y : Integer end
  @Operation addToPorts end
  @Operation addToDisplays end
  @Operation addRightClickMenuItem end
  @Operation addDaemons end
  @Operation add end
  @Operation delete end
  @Operation fire end
  @Operation findElement end
  @Operation height end
  @Operation id end
  @Operation move end
  @Operation new end
  @Operation ports end
  @Operation rightClickMenuSelected end
  @Operation resize end
  @Operation setHeight end
  @Operation setWidth end
  @Operation setId end
  @Operation setY end
  @Operation setX end
  @Operation toString end
  @Operation width end
  @Operation x end
  @Operation y end
end
```

10.17 Class `PropertyChanged`

context: `Root::Clients::Diagrams`

interface:

```
@Class PropertyChanged extends ChangeEvent
  @Attribute newValue : Element end
  @Attribute oldValue : Element end
  @Attribute property : String end
  @Operation newValue end
  @Operation oldValue end
  @Operation property end
  @Operation toString end
end
```

10.18 Class `PopupMenu`

context: `Root::Clients::Diagrams`

interface:

```
@Class PopupMenu extends Object
  @Attribute items : Seq(MenuItem) end
  @Attribute id : Element end
  @Operation addToItems end
  @Operation add end
  @Operation itemSelected end
  @Operation new end
  @Operation setId end
  @Operation toString end
end
```

10.19 Class `Port`

context: `Root::Clients::Diagrams`

interface:

```
@Class Port extends Object
  @Attribute height : Integer end
  @Attribute id : Element end
  @Attribute width : Integer end
  @Attribute x : Integer end
  @Attribute y : Integer end
  @Operation height end
  @Operation id end
  @Operation new end
  @Operation resize end
  @Operation setId end
  @Operation toString end
  @Operation width end
  @Operation x end
  @Operation y end
end
```

10.20 Class `Text`

context: `Root::Clients::Diagrams`

interface:

```
@Class Text extends Display
  @Attribute editable : Boolean end
  @Attribute id : Element end
  @Attribute text : String end
  @Attribute x : Integer end
  @Attribute y : Integer end
  @Operation delete end
  @Operation editable end
  @Operation id end
  @Operation new end
  @Operation setId end
  @Operation toString end
  @Operation text end
  @Operation textChanged end
  @Operation x end
  @Operation y end
end
```

10.21 Class `WayPoint`

context: `Root::Clients::Diagrams`

interface:

```
@Class WayPoint extends Object
  @Attribute index : Integer end
  @Attribute id : Element end
  @Attribute x : Integer end
  @Attribute y : Integer end
  @Operation index end
  @Operation id end
  @Operation move end
  @Operation new end
  @Operation setId end
  @Operation toString end
  @Operation x end
  @Operation y end
end
```

11 Package Comms

context: Root

11.1 Class Client

context: Root::Comms

interface:

```
@Class Client extends Object
  @Attribute input : InputChannel end
  @Attribute output : OutputChannel end
  @Operation connect end
end
```

11.2 Class Server

context: Root::Comms

interface:

```
@Class Server extends Object
  @Attribute input : InputChannel end
  @Attribute output : OutputChannel end
  @Operation listen end
end
```

12 Package `Compiler`

context: `Root`
overview:

> The compiler package defines operations and classes that are used for compiling. Loading the compiler package also extends the OCL abstract syntax classes with operations that support the compilation of OCL. Use the operations defined in this package to compile files and compile abstract syntax directly to core.

12.1 Class `CompilationUnit`

context: `Root::Compiler`
overview:

> A compilation unit is produced by the parser when we load a file for compiling. Send the compilation unit a 'compile' message to compile the contents.

constructor: `CompilationUnit(imports,exps)`

> The imports are those packages that are imported at run-time and are referenced by the program code. The expressions are evaluated in the environment created by the imports.

interface:

```
@Class CompilationUnit extends Object
  @Attribute exps : Seq(Performable) end
  @Attribute imports : Seq(Import) end
  @Operation compileFile end
  @Operation clearInstrs end
  @Operation env end
  @Operation foldImports end
  @Operation foldExps end
  @Operation init end
  @Operation writeInstrs end
end
```

12.2 Class `Dynamic`

context: `Root::Compiler`

interface:

```
@Class Dynamic extends Var
  @Operation ref end
end
```

12.3 Class `Env`

context: `Root::Compiler`

interface:

```
@Class Env extends Object
  @Attribute vars : Seq(Var) end
  @Operation allocateLocals end
  @Operation consNameSpaceRef end
  @Operation consLocal end
  @Operation deleteLocals end
  @Operation globalOffset end
  @Operation globalFrame end
  @Operation globals end
  @Operation isLocal end
  @Operation isGlobal end
  @Operation incNameSpaceRefs end
  @Operation init end
  @Operation locals end
  @Operation localIndex end
  @Operation maxLocal end
  @Operation newLocals end
  @Operation newGlobals end
  @Operation newFrame end
  @Operation ref end
  @Operation refs end
  @Operation setVarCode end
  @Operation setLocalCode end
  @Operation setLocalsCode end
  @Operation toString end
end
```

12.4 Class `Global`

context: `Root::Compiler`

interface:

```
@Class Global extends Var
  @Attribute frame : Integer end
  @Attribute offset : Integer end
  @Operation ref end
  @Operation toString end
end
```

12.5 Class `Import`

context: `Root::Compiler`

interface:

```
@Class Import extends OCL
  @Attribute nameSpace : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation init end
  @Operation maxLocals end
  @Operation pprint end
end
```

12.6 Class `Local`

context: `Root::Compiler`

interface:

```
@Class Local extends Var
  @Attribute offset : Integer end
  @Operation ref end
  @Operation toString end
end
```

12.7 Class `NameSpaceRef`

context: `Root::Compiler`

interface:

```
@Class NameSpaceRef extends Var
  @Attribute contour : Integer end
  @Operation init end
  @Operation inc end
  @Operation ref end
end
```

12.8 Class `Operations`

context: `Root::Compiler`
overview:

> A collection of operations that are defined in the same scope and which should be merged into a single operation via pattern maching. The order of the operation elements is important since they will be each tried in turn when pattern matching occurs.

constructor: `Operations(name,ops)`

> Ops is a sequence of operations.

interface:

```
@Class Operations extends Object
  @Attribute name : String end
  @Attribute operations : Seq(Operation) end
  @Operation arity end
  @Operation bindVars end
  @Operation compile end
  @Operation desugarBody end
  @Operation matchCode end
end
```

12.9 Class `Symbol`

context: `Root::Compiler`

interface:

```
@Class Symbol extends NamedElement
  @Operation init end
end
```

12.10 Class `Var`

context: `Root::Compiler`

interface:

```
@Class Var extends NamedElement
  @Operation toString end
end
```

12.11 Operation `compileExp`

context: `Root::Compiler`

12.12 Operation `compileFile`

context: `Root::Compiler`
overview:

Use compileFile/3 to compile a file. If the compilation succeeds then the source file is translated to machine code and an object file and an instruction file is created. The object file contains a binary format machine code and the instructions file contains a human readable version of the object file.

parameter: `name:String`

A pathname designating the file to compile.

parameter: `isLast:Boolean`

A boolean value that defines whteher this is the last in a sequence of expressions (should be ’true’).

parameter: `saveSource:Boolean`

A boolean value that determines whether or not the source code is saved to the object file. In most cases you will want to save the source code as this aids debugging and is reachable from the stack frame when a particular operation body fails.

12.13 Operation `compileToFun`

context: `Root::Compiler`
overview:

This operation is used to translate an expression directly to a function. Compilation is provided with a sequence of argument names and a collection of imported dynamic variables and namespaces.

parameter: `exp:Performable`

The expression to be compiled.

parameter: `args:Seq(String)`

A sequence of the arguments to the function in the order that they will be supplied.

parameter: `dynamics:Seq(Element)`

The dynamics of a function is a sequence of pairs. The first element of the pair is an integer defining whether the dynamic is a binding (1) or a namespace (2). If the pair is desigated as a binding then the tail of the pair should be a pair whose head is a symbol (the name of the binding) and the tail is the value of the binding. If the pair is designated as a namespace then the tail of the pair must be a table. All elements of the table are then visible to the code being compiled.

parameter: `saveSource:String`

A boolean that determines whether or not the source code being compiled is saved in the function. Saving source code is useful for debugging as it is available to the debugger through the run-time stack. Saving source code increases the size of the run-time system and imposes a penalty since it must be garbage collected).

12.14 Operation `expandFileName`

context: `Root::Compiler`

13 Package `Dialogs`

context: `Root`
overview:

> The Dialogs package defines a collection of classes that model user dialogs (i.e. questions, requests for information etc.) in a way that is independent of the representation used to display the dialog.

13.1 Class `Commands`

context: `Root::Dialogs`

interface:

```
@Class Commands extends Sugar
  @Attribute commands : Seq(Performable) end
  @Operation desugar end
  @Operation pprint end
end
```

13.2 Class `Choose`

context: `Root::Dialogs`

interface:

```
@Class Choose extends Sugar
  @Attribute choices : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.3 Class `Dialog`

context: `Root::Dialogs`
overview:

constructor: `Dialog(name,args)`

syntax:

```
       Dialog ::= ’@Dialog’ [ Name ] [ ’(’ Args ’)’ ] (Exp:Dialog)* Exp end
```

interface:

```
@Class Dialog extends Sugar
  @Attribute args : Seq(String) end
  @Attribute body : Performable end
  @Attribute dialogs : Seq(Dialog) end
  @Attribute name : String end
  @Operation desugarOperation end
  @Operation desugarDialogs end
  @Operation desugar end
  @Operation pprint end
end
```

13.4 Class `Display`

context: `Root::Dialogs`

interface:

```
@Class Display isabstract extends Object
  @Operation choose end
  @Operation option end
  @Operation reset end
  @Operation show end
  @Operation separator end
  @Operation value end
end
```

13.5 Class `Let`

context: `Root::Dialogs`

interface:

```
@Class Let extends Sugar
  @Attribute body : Performable end
  @Attribute name : String end
  @Attribute value : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.6 Class `Option`

context: `Root::Dialogs`

interface:

```
@Class Option extends Sugar
  @Attribute dialog : Performable end
  @Attribute message : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.7 Class `Options`

context: `Root::Dialogs`

interface:

```
@Class Options extends Sugar
  @Attribute options : Seq(Option) end
  @Operation desugar end
  @Operation pprint end
end
```

13.8 Class `Quit`

context: `Root::Dialogs`

interface:

```
@Class Quit extends Sugar
  @Attribute exp : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.9 Class `Skip`

context: `Root::Dialogs`

interface:

```
@Class Skip extends Sugar
  @Operation desugar end
  @Operation pprint end
end
```

13.10 Class `Separator`

context: `Root::Dialogs`

interface:

```
@Class Separator extends Sugar
  @Operation desugar end
  @Operation pprint end
end
```

13.11 Class `Show`

context: `Root::Dialogs`

interface:

```
@Class Show extends Sugar
  @Attribute label : Performable end
  @Attribute value : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.12 Class `StandardDialog`

context: `Root::Dialogs`

interface:

```
@Class StandardDialog extends Sugar
  @Attribute dialog : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

13.13 Class `TextDisplay`

context: `Root::Dialogs`
overview:

The TextDisplay class implements the Display operations to produce a command line interface for dialogs.

interface:

```
@Class TextDisplay extends Display
  @Attribute dialogs : Seq(Operation) end
  @Attribute input : InputChannel end
  @Attribute messages : Seq(String) end
  @Attribute output : OutputChannel end
  @Operation choose end
  @Operation getOption end
  @Operation option end
  @Operation printOptions end
  @Operation reset end
  @Operation readSymbol end
  @Operation readString end
  @Operation readElement end
  @Operation show end
  @Operation separator end
  @Operation toString end
  @Operation value end
end
```

13.14 Class `Value`

context: `Root::Dialogs`

interface:

```
@Class Value extends Sugar
  @Attribute type : Seq(String) end
  @Operation desugar end
  @Operation pprint end
end
```

14 Package `Env`

context: `Root`
overview:

The Env package implements environments. An environment is a lookup table associating keys and values. Environment update produces a new environment, making them suitable for systems that backtrack for example.

class: `Env`

An abstract class that is the super class of all environment classes. Provides an interface of environment operations.

class: `NullEnv()`

Create a new environment.

operation: `Env::bind(key,value)`

Associate the key with the value and return a new environment.

operation: `Env::binds(key)`

Test whether the environment associates the given key.

operation: `Env::lookup(key)`

Return the value associated with the given key. Causes an error if the environment does not associate the key.

example:

```
let e = NullEnv().bind("x",1).bind("y",2)
in if e.binds("x") and e.binds("y")
   then e.lookup("x") + e.lookup("y")
   else 0
   end
end
```

14.1 Class `Binding`

context: `Root::Env`
overview:

An internal class to the package Env. Used to represent bindings. Always use the 'bind' operation to add bindings to an environment.

interface:

```
@Class Binding extends Env
  @Attribute name : String end
  @Attribute value : Element end
  @Operation binds end
  @Operation binding end
  @Operation lookup end
  @Operation setValue end
  @Operation toString end
end
```

14.2 Class `Env`

context: `Root::Env`
overview:

The class Env is abstract and is the superclass of all environment implementation classes. Use Env as the type for all environments.

interface:

```
@Class Env isabstract extends Object
  @Operation add end
  @Operation binding end
  @Operation bind end
  @Operation binds end
  @Operation lookup end
  @Operation setValue end
end
```

14.2.1 Operation `bind`

context: `Root::Env::Env`
parameter: `key:Element`

An environment key.

parameter: `value:Element`

A value for the key.

overview:

Adds an association between the key and the value. Returns a new environment which is a copy of the receiver extended with the association.

14.2.2 Operation `binds`

context: `Root::Env::Env`
parameter: `key:Element`

A key possibly associated by the environment.

overview:

Returns true when the key is associated in the environment otherwise returns false.

14.2.3 Operation `lookup`

context: `Root::Env::Env`
parameter: `key:Element`

A key associated in the environment.

overview:

lookup(key) returns the value of the key in the environment. Causes an error is the key is not associated in the environment.

14.2.4 Operation `setValue`

context: `Root::Env::Env`
overview:

> Set will update the current value of a key by side effect.

14.3 Class `NullEnv`

context: `Root::Env`
constructor: `NullEnv()`

> Creates an empty environment.

overview:

> Represents the empty environment. Create an instance of this class then add associations using ’binds’.

interface:

```
@Class NullEnv extends Env
  @Operation binds end
  @Operation binding end
  @Operation toString end
end
```

14.4 Class `Pair`

context: `Root::Env`
constructor: `Pair(e1,e2)`

> Constructs the concatenation of environments e1 and e2. Associations in e1 shadow those from e2.

overview:

> Used to represent the concatenation of two environments.

interface:

```
@Class Pair extends Env
  @Attribute left : Env end
  @Attribute right : Env end
  @Operation binds end
  @Operation binding end
  @Operation lookup end
  @Operation toString end
end
```

15 Package `EMOF`

context: `Root`

15.1 Class `Attribute`

context: `Root::EMOF`
overview:

> An attribute is a structural feature of a class. It defines the name and type of a slot of the instances of the class. When the class is instantiated, a new object is created and a slot is added for each attribute defined and inherited by the class. Each slot is initialised to contain the default value for the type of the corresponding attribute.

constructor: `Attribute(name,type)`

> The name is a string and the type is a classifier.

interface:

```
@Class Attribute extends StructuralFeature
  @Operation init end
  @Operation underlyingType end
end
```

15.1.1 Operation `underlyingType`

context: `Root::EMOF::Attribute`
overview:

> The underlying type of an attribute is the type as would appear at the end of a directed association on a class diagram. All occurrences of Set(..) and Seq(..) are stripped off.

15.2 DataType `Boolean`

overview:

> The data type for boolean values. A boolean value is either true or false. The default value is false.

interface:

```
@DataType Boolean extends Element
  @Operation toString end
end
```

15.3 Class `BehaviouralFeature`

context: `Root::EMOF`
overview:

> A behavioural feature is a typed element that can be invoked. Typically a behavioural feature is an operation.

interface:

```
@Class BehaviouralFeature isabstract extends TypedElement
  @Attribute documentation : String end
end
```

15.4 Class `Bind`

context: `Root::EMOF`
overview:

A binding is a named value.

constructor: `Bind(name,value)`

Constructs a binding, the name is a string and the value is any element.

interface:

```
@Class Bind extends NamedElement
  @Attribute value : Element end
  @Operation init end
  @Operation init end
end
```

15.5 Class `CompiledOperation`

context: `Root::EMOF`
overview:

CompiledOperation is the type of all XMF compiled operations. A compiled operation can be invoked using ’invoke/2’ or by applying it to its arguments. A compiled operation consists of machine code instructions. A compiled operation may be associated with its source code to aid debugging.

interface:

```
@Class CompiledOperation extends Operation
  @Operation arity end
  @Operation setSupers end
  @Operation toLatex end
end
```

15.6 Class `Contained`

context: `Root::EMOF`
overview:

A contained element has an owner. The owner is set when the contained element is added to a container. Removing an owned element from a container and adding it to another container will change the value of ’owner’ in the contained element.

interface:

```
@Class Contained isabstract extends Object
  @Attribute owner : Container end
  @Operation setOwner end
  @Operation setOwner end
end
```

15.7 Class Container

context: Root::EMOF
overview:

> A container has a slot 'contents' that is a table. The table maintains the contained elements indexed by keys. By default the leys for the elements in the table are the elements themselves, but sub-classes of container will modify this feature accordingly. Container provides operations for accessing and managing its contents.

interface:

```
@Class Container extends Object
  @Attribute contents : Table end
  @Operation add end
  @Operation contents end
  @Operation initContents end
  @Operation init end
  @Operation remove end
end
```

15.8 Class Constraint

context: Root::EMOF
overview:

> A constraint is a named boolean expression owned by a classifier. Constraints are defined by classifiers to be performed with respect to their instances and as such any occurrences of 'self' in a constraint will refer to the instance that is being checked.

interface:

```
@Class Constraint extends NamedElement
  @Attribute body : Operation end
  @Attribute reason : Operation end
  @Operation invoke end
end
```

15.9 Class Constructor

context: Root::EMOF

interface:

```
@Class Constructor extends Object
  @Attribute body : Element end
  @Attribute names : Seq(String) end
  @Operation invoke end
  @Operation init end
  @Operation ref end
  @Operation setOwner end
end
```

15.10 Class `Classifier`

context: `Root::EMOF`
overview:

> A classifier is a name space for operations and constraints. A classifier is generalizable and has parents from which it inherits operations and constraints. A classifier can be instantiated via 'new/0' and 'new/1'. In both cases the default behaviour is to return a default value as an instance. If the classifier is a datatype then the basic value for the datatype is returned otherwise 'null' is returned as the default value. A classifier can also be applied to arguments (0 or more) in order to instantiate it. Typically you will not create a Classifier directly, but create a class or an instance of a sub-class of Class.

interface:

```
@Class Classifier extends NameSpace
  @Attribute constraints : Set(Constraint) end
  @Attribute default : Element end
  @Attribute grammar : Grammar end
  @Attribute operations : Set(Operation) end
  @Attribute parents : Set(Classifier) end
  @Operation addGrammar end
  @Operation add end
  @Operation allParents end
  @Operation allOperations end
  @Operation allConstraints end
  @Operation addOperation end
  @Operation addConstraint end
  @Operation add end
  @Operation add end
  @Operation addOperation end
  @Operation classify end
  @Operation defaultParents end
  @Operation default end
  @Operation default end
  @Operation grammar end
  @Operation getOperation end
  @Operation getConstraint end
  @Operation getOperation end
  @Operation hasOperation end
  @Operation invoke end
  @Operation initParents end
  @Operation initOperations end
  @Operation init end
  @Operation inheritsFrom end
  @Operation inheritsFrom end
  @Operation new end
  @Operation new end
  @Operation removeOperation end
  @Operation removeConstraint end
  @Operation remove end
  @Operation removeOperation end
  @Operation shadowOperation end
end
```

15.10.1 Operation `allParents`

context: `Root::EMOF::Classifier`
overview:

> The set of all parents of a classifier.

15.10.2 Operation `allOperations`

context: `Root::EMOF::Classifier`
overview:

> Get All the operations defined and inherited by the receiver. The order of the operations is very important because this is the order in which message lookup occurs. To calculate 'allOperations/0' we use an Operator Precedence Ordering. This is a depth first left to right traversal of the classification type lattice up to a join. Stopping at join points means that operations that would otherwise appear multiple times in the OPO are promoted to their last occurrence.

15.10.3 Operation `allConstraints`

context: `Root::EMOF::Classifier`
overview:

> Get all the constraints defined and inherited by the receiver.

15.10.4 Operation `addOperation`

context: `Root::EMOF::Classifier`
overview:

> Adds an operation to a classifier. Ise 'add/1' in preference to this operation.

15.10.5 Operation `addConstraint`

context: `Root::EMOF::Classifier`
overview:

> Adds a constraint to a classifier. Use 'add/1' in preference to this operation.

15.10.6 Operation `add`

context: `Root::EMOF::Classifier`
overview:

> Adds a named element to a classifier. If the named element is an operation or constraint then it is added to the appropriate attributes of the classifier. Sub-classes of Classifier can extend this as appropriate but should also call thi via 'super'.

15.10.7 Operation `classify`

context: `Root::EMOF::Classifier`
overview:

> A classifier classifies its instances by running constraints against them. The result of classification is a set of constraint reports that describe whether the classification succeeded and why the classification fails. To classify a candidate supply it to 'classify/1' to produce the set of reports.

15.10.8 Operation `defaultParents`

context: `Root::EMOF::Classifier`
overview:

> When creating a classifier it is possible to state the default parents so that if no parents are specified when creating an instance of the classifier the set returned by this operation is used. The default parent for a classifier is Element.

15.10.9 Operation `default`

context: `Root::EMOF::Classifier`
overview:

> When a classifier is used as an attribute type the corresponding slot values are initialised to the default value returned by this operation.

15.10.10 Operation `getOperation`

context: `Root::EMOF::Classifier`
overview:

> Index an operation by name.

15.10.11 Operation `getConstraint`

context: `Root::EMOF::Classifier`
overview:

> Index a constraint by name.

15.10.12 Operation `invoke`

context: `Root::EMOF::Classifier`
overview:

> A classifier is invoked to (by default) initialise itself with respect to some arguments. This is the preferred way for classifiers to be instantiated.

15.10.13 Operation `initParents`

context: `Root::EMOF::Classifier`
overview:

> Initialise the parents of a classifier. When the compiler encounters a package of definitions it turns references into operations that return the references when they are called with no arguments. This is the mechanism by which mutual recursion is implemented. The dereferencing occurs by delaying the mutual recursion in the body of the operations; the compiler will have compiled the references in the body of the operation as an appropriate lookup in the containing namespace. It is therefore important that namespaces such as classifiers and classes, and elements that reference things in

namespaces such as attributes and operations are initialised. The initialisation is fairly simple since the compiler has done all the work: initialisation just calls the operation which then performs the delayed reference.

15.10.14 Operation `initOperations`

context: `Root::EMOF::Classifier`
overview:

Initialise all the operations.

15.10.15 Operation `init`

context: `Root::EMOF::Classifier`
overview:

To initialise a classifier, initialise the operations, the parents and run super.

15.10.16 Operation `inheritsFrom`

context: `Root::EMOF::Classifier`
overview:

A classifier inherits from another if they are the same or we can ue the ’parents’ relation transitively to link the two. Supply the super-classifier to this operation which will return true when the receiver inherits from the argument.

15.10.17 Operation `new`

context: `Root::EMOF::Classifier`
overview:

’new/1’ takes a sequence of initialisation arguments. It calls ’new/0’ to create the new instance and then calls ’init/1’ with the initialisation arguments.

15.10.18 Operation `new`

context: `Root::EMOF::Classifier`
overview:

’new/0’ is defined by classifiers to create a new instance. The default behaviour returns the default value.

15.10.19 Operation `remove`

context: `Root::EMOF::Classifier`
overview:

’remove/1’ removes a named element from a classifier. This is the preferred way of removing an element. It causes the named element to be removed from the namespace of the element (i.e. the contents table) and also removes operations and constraints from the appropriate slots. Subclasses of Classifier should extend this as appropriate, but should call this via ’super’.

15.11 Class `Class`

context: `Root::EMOF`
overview:

A class is a classifier with structural features (i.e. attributes). Instances of classes are always objects with slots for the attributes of the class. A class is instantiated using the ’new/0’ and ’new/1’ operations (inherited from Classifier). The former takes no initialization arguments whereas the latter takes a sequence of initialization arguments. The preferred way of instantiating a class is by applying it as an operator to the initialization arguments, as in C() or C(1,2,3). This instantiates the class and calls ’init/1’ on the resulting instance. Typically classes will redefine ’init/1’ to initialize new instances on a class-by-class basis. Typically you will create a class using the @Class ... end notation.

interface:

```
@Class Class extends Classifier
  @Attribute attributes : Set(Attribute) end
  @Attribute constructors : Seq(Constructor) end
  @Attribute isAbstract : Boolean end
  @Operation allConstructors end
  @Operation allAttributes end
  @Operation addOperation end
  @Operation addConstructor end
  @Operation addAttribute end
  @Operation add end
  @Operation constraintsToLatex end
  @Operation defaultParents end
  @Operation getConstructor end
  @Operation getAttribute end
  @Operation interface end
  @Operation init end
  @Operation new end
  @Operation operationsToLatex end
  @Operation removeAttribute end
  @Operation remove end
  @Operation toLatex end
end
```

15.11.1 Operation `allConstructors`

context: `Root::EMOF::Class`
overview:

Calculate a sequence of constructors in most specific to least specific order.

15.11.2 Operation `allAttributes`

context: `Root::EMOF::Class`
overview:

Get all the attributes that are defined and inherited by the class. Refer to the ’attributes’ attribute of a class to get the locally defined attributes of a class.

15.11.3 **Operation** `addOperation`

context: `Root::EMOF::Class`
overview:

Obsolete.

15.11.4 **Operation** `addConstructor`

context: `Root::EMOF::Class`
overview:

Add a constructor to a class. Use Class::add in preference to this.

15.11.5 **Operation** `addAttribute`

context: `Root::EMOF::Class`
overview:

Adds an attribute to a class and sets the owner of the attribute to be the class. Use Class::add in preference to this since that will also add the attribute to the contents table of the class.

15.11.6 **Operation** `add`

context: `Root::EMOF::Class`
overview:

Extend the behaviour for ’add’ inherited from Classifier by taking attributes and constructors into account.

15.11.7 **Operation** `defaultParents`

context: `Root::EMOF::Class`
overview:

When a class is created its parents may not be specified as part of the definition. The meta-class that is instantiated can specify the default superclasses of the new class using this operation. It is automatically called when the new class is initialised. It is a useful way of stating that all classes of a given type must inherit from a collection of super-classes. The default is Object.

15.11.8 **Operation** `getConstructor`

context: `Root::EMOF::Class`
overview:

Return the most specific constructor with the given arity or null if no constructor exists.

15.11.9 Operation `getAttribute`

context: `Root::EMOF::Class`
overview:

> Get an attribute using its name. Be aware that atttibutes, like all instances of NamedElement use symbols for names. The name argument supplied to 'getAttribute' can be a string or a symbol.

15.11.10 Operation `init`

context: `Root::EMOF::Class`
overview:

> Initialise a class by initialising the attributes and then initialising as a classifier.

15.11.11 Operation `new`

context: `Root::EMOF::Class`
overview:

> Create a new instance of a class. Each attribute becomes a slot in the new instance and the values of the slots are the default values of the corresponding attributes. Once the new object has been created, it is sent an 'init/0' message. Therefore, objects are initialised on a type-by-type basis. Note that there are two versions of 'new': this one 'new/0' that takes no arguments and uses 'init/0' to initialise the object; 'new/1' that takes initialisation arguments. In general, avoid using 'new' to instantiate classes and apply the class to initialisation arguments instead.

15.11.12 Operation `removeAttribute`

context: `Root::EMOF::Class`
overview:

> Remove an attribute supplied as an argument. Use 'remove/1' in preference to this operation.

15.11.13 Operation `remove`

context: `Root::EMOF::Class`
overview:

> Extend the behaviour inherited from Classifier by taking attributes into account. The argument is a named element to be removed from the receiver.

15.11.14 Operation `toLatex`

context: `Root::EMOF::Class`
overview:

A class is flattened into a sequence of latex definitions for the class interface, the class operations and the class constraints. The class interface is complete. The detailed descriptions that follow occur only if the definition has a documentation string.

parameter: `out:OutputChannel`

The output channel to send the latex source to.

15.12 Class `ConstraintReport`

context: `Root::EMOF`

interface:

```
@Class ConstraintReport extends Object
  @Attribute constraint : Constraint end
  @Attribute candidate : Element end
  @Attribute reason : String end
  @Attribute satisfied : Boolean end
  @Operation toString end
end
```

15.13 Class `DataType`

context: `Root::EMOF`
overview:

DataType is a sub-class of Classifier that designates the non-object classifiers that are basic to the XMF system. An instance of DataType is a classifier for values (the instances of the data type). For example Boolean is an instance of DataType - it classifies the values ’true’ and ’false’. For example Integer is an instance of DataType - it classifies the values 1, 2, etc.

interface:

```
@Class DataType isabstract extends Classifier
  @Operation interface end
  @Operation toLatex end
end
```

15.14 Class `Element`

context: `Root::EMOF`
overview:

Element is the root class of the XMF type hierarchy. It has no superclasses. Everything is an instance of Element. Operations defined on Element are available to every data value in XMF. Use this as a type when you want to use heterogeneous values (for example as the type of an attribute). An element always has a classifier which is the value of the ’of/0’ message. An element can always be sent messages using ’send/2’ where the first argument is the name of the message and the second argument is a sequence of message arguments.

interface:

```
@Class Element isabstract extends
  @Operation error end
  @Operation init end
  @Operation init end
  @Operation isKindOf end
  @Operation init end
  @Operation of end
  @Operation println end
  @Operation print end
  @Operation setOf end
  @Operation send end
  @Operation save end
  @Operation toLatex end
  @Operation toString end
  @Operation yield end
end
```

15.14.1 Operation `error`

context: `Root::EMOF::Element`
overview:

> You should really throw something specific. As a last resort you can call error and a general exception will be raised.

15.14.2 Operation `init`

context: `Root::EMOF::Element`
overview:

> Initialisation with respect to some arguments. Specialize this operation as necessary. Note that ’init/1’ is used only when a class is instatiated directly with some initialisation args. It must arrange for ’init/0’ to be called in order to perform any default initialisation.

15.14.3 Operation `init`

context: `Root::EMOF::Element`
overview:

> All elements can be initialised. The default behaviour is to do nothing. Init always returns the receiver. All elements should be initialised. ’init/0’ is the operation where default initialisation should take place. All other initialisation operations should call ’init/0’ where appropriate.

15.14.4 Operation `println`

context: `Root::EMOF::Element`
overview:

> Any element can be printed (with newline). This uses ’toString/0’ to turn the receiver into a string.

15.14.5 Operation `print`

context: `Root::EMOF::Element`
overview:

> Any element can be printed (no newline). This uses 'toString/0' to turn the receiver into a string.

15.14.6 Operation `setOf`

context: `Root::EMOF::Element`
overview:

> It is possible to dynamically change the classifier of an element. This is not, however, recommended.

15.14.7 Operation `save`

context: `Root::EMOF::Element`
overview:

> Serialize any element to a file.

15.14.8 Operation `toString`

context: `Root::EMOF::Element`
overview:

> All elements can transform themselves into strings. This is used to display the element. Subclasses of Element should redefine this appropriately. The system uses 'toString/0' to transform values into strings prior to printing them.

15.14.9 Operation `yield`

context: `Root::EMOF::Element`
overview:

> Give up control to any pending thread.

15.15 Class `Exception`

context: `Root::EMOF`
overview:

> An exception is raised when something goes wrong. An exception contains a message that reports what went wrong. An exception also contains a sequence of stack frames that defines the history of computation at the point the exception was raised. An exception may optionally contain information about where in the source file the error occurred. This is encoded as the lineCount and charCount.

interface:

```
@Class Exception extends Object
  @Attribute backtrace : Seq(Element) end
  @Attribute charCount : Integer end
  @Attribute lineCount : Integer end
  @Attribute message : String end
  @Attribute resourceName : String end
  @Operation printFrame end
  @Operation printBacktrace end
  @Operation setBacktrace end
  @Operation toString end
end
```

15.16 Class `Enum`

context: `Root::EMOF`

interface:

```
@Class Enum extends DataType
  @Attribute values : Seq(String) end
end
```

15.17 Class `ForeignOperation`

context: `Root::EMOF`

interface:

```
@Class ForeignOperation extends Operation
  @Operation getStructuralFeatureNames end
  @Operation toLatex end
end
```

15.18 DataType `Integer`

overview:

The data type for integer values The default value is 0.

interface:

```
@DataType Integer extends Element
  @Operation isWhiteSpaceChar end
  @Operation isUpperCaseChar end
  @Operation isNumericChar end
  @Operation isNewLineChar end
  @Operation isLowerCaseChar end
  @Operation lsh end
  @Operation min end
  @Operation max end
  @Operation rsh end
  @Operation toString end
  @Operation to end
end
```

15.19 Class `InterpretedOperation`

context: `Root::EMOF`
overview:

An interpreted operation is created when we evaluate an operation definition.

interface:

```
@Class InterpretedOperation extends Operation
  @Attribute body : Performable end
  @Attribute env : Seq(Element) end
  @Attribute imports : Seq(NameSpace) end
  @Attribute parameters : Seq(Pattern) end
  @Operation arity end
  @Operation bindParams end
  @Operation invoke end
  @Operation setSupers end
end
```

15.20 Class `InitialisedAttribute`

context: `Root::EMOF`

interface:

```
@Class InitialisedAttribute extends Attribute
  @Attribute value : Performable end
end
```

15.21 Class `LazySeq`

context: `Root::EMOF`

interface:

```
@Class LazySeq extends Object
  @Attribute head : Element end
  @Attribute tail : Element end
  @Operation expand end
  @Operation head end
  @Operation isExpanded end
  @Operation isEmpty end
  @Operation init end
  @Operation tail end
end
```

15.22 Class `MachineException`

context: `Root::EMOF`

interface:

```
@Class MachineException extends Exception
  @Attribute id : Integer end
end
```

15.23 Class `NameSpace`

context: `Root::EMOF`
overview:

> A name space is a container of named elenents. A name space defines two operations ’getElement/1’ and ’hasElement/1’ that are used to get an element by name and check for an element by name. Typically a name space will contain different categories of elements in which case the name

space will place the contained elements in its contents table and in a type specific collection. For example, a class is a container for operations, attributes and constraints. Each of these elements are placed in the contents table for the class and in a slot containing a collection with the names 'operations', 'attributes; and 'constraints' respectively. The special syntax '::' is used to invoke the 'getElement/1' operation on a name space.

interface:

```
@Class NameSpace extends NamedElement,Container
  @Attribute bindings : Set(Bind) end
  @Attribute documentation : String end
  @Attribute imports : Seq(NameSpace) end
  @Operation addBinding end
  @Operation add end
  @Operation add end
  @Operation getContents end
  @Operation getElement end
  @Operation hasElement end
  @Operation initBindings end
  @Operation init end
  @Operation init end
  @Operation names end
  @Operation nameChanged end
  @Operation putElement end
  @Operation toLatex end
end
```

15.24 Class `NamedElement`

context: `Root::EMOF`
overview:

A named element is an owned element with a name. The name may be a srring or a symbol. typically we use symbols where the lookup of the name needs to be efficient.

interface:

```
@Class NamedElement isabstract extends Contained
  @Attribute name : String end
  @Operation getNamedElement end
  @Operation pathSeq end
  @Operation path end
  @Operation setName end
  @Operation toString end
end
```

15.25 DataType `Null`

overview:

Null is the data type for the special value 'null'. The special value 'null' is an instance of all classifiers. It is the default value for all instances of Class.

interface:

```
@DataType Null extends Element
  @Operation toString end
end
```

15.26 Class `Operation`

context: `Root::EMOF`
overview:

> Operation is the abstract super-class of all operations in XMF. An operation can be compiled or interpreted. All operations have parameters, a return type and a body. The body must be performable. An operation is invoked using 'invoke/2' where the forst argument is the value of 'self' in the operation body and the secon dargument is a sequence of parameter vaolues.

interface:

```
@Class Operation extends BehaviouralFeature
  @Operation arity end
  @Operation fork end
  @Operation invoke end
  @Operation setSupers end
  @Operation toLatex end
end
```

15.27 Class `Object`

context: `Root::EMOF`
overview:

> Object is the super-class of all classes with structural features oin XMF. Object provides access to slots via the 'get/1' and 'set/2' operations. Object is the default super-class for a class definition - if you do not specify a super-class then Object is assumed.

interface:

```
@Class Object extends Element
  @Operation addStructuralFeature end
  @Operation addDaemon end
  @Operation daemons end
  @Operation fire end
  @Operation get end
  @Operation getStructuralFeatureNames end
  @Operation hasStructuralFeature end
  @Operation init end
  @Operation removeDaemon end
  @Operation setDaemons end
  @Operation set end
end
```

15.27.1 Operation `addDaemon`

context: `Root::EMOF::Object`
overview:

> All objects have a collection of daemons. A daemon is an operation that is invoked whenever a slot of the object is updated. The operation is any invokable value (either an operation or an object that implements the 'invoke/2' operation. The invocation occurs after the slot has been updated and is supplied with 3 values as the second argument: the slot (symbol) that has been updated, the new value and the old value.

15.27.2 Operation `fire`

context: `Root::EMOF::Object`
overview:

When the slot of an object is updated, its daemons are fired by calling this operation. The operation is supplied with 3 arguments: the name of the slot that was changed (a symbol), the new value of the slot and the old value of the slot.

15.27.3 Operation `init`

context: `Root::EMOF::Object`
overview:

When an object is initialised, by default we look for a constructor that has the same arity as the supplied arguments. If we find one then it is invoked.

15.28 Class `Parameter`

context: `Root::EMOF`
overview:

A parameter is a typed element that occurs in operations.

interface:

```
@Class Parameter extends TypedElement
  @Operation init end
  @Operation lift end
end
```

15.29 Class `Performable`

context: `Root::EMOF`
overview:

A performable element can be executed on the XMF VM. It must provide a collection of operations that support its evaluation or its translation into VM machine instructions. In particular it must support ’compile/4’, ’FV/0’, ’maxLocals/0’ and ’eval/3’. Performable is the root class for all extensions to executable XMF. For example OCL is a sub-class of Performable. If you intend to define your own languages in XMF then they should extend Performable.

interface:

```
@Class Performable isabstract extends Object
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation pprint end
  @Operation pprint end
end
```

15.30 Class `Package`

context: `Root::EMOF`
overview:

> A package is a classifier that has instances. Currently we don't support instantiating packages therefore you should view a package as a name space.

interface:

```
@Class Package extends Resource,Classifier
  @Attribute metaPackage : Package end
  @Operation addPackage end
  @Operation addClass end
  @Operation add end
  @Operation contentsToLatex end
  @Operation defaultParents end
  @Operation init end
  @Operation remove end
  @Operation toLatex end
end
```

15.30.1 Operation `toLatex`

context: `Root::EMOF::Package`
overview:

> A package is flattened into a sequence of latex definitions A simple package definition and a sequence of content definitions.

parameter: `out:OutputChannel`

> Supply the output channel to send the latex source code to.

15.31 Class `Resource`

context: `Root::EMOF`
overview:

> A resource records where the resource originated via a resource name. For example a definition is a resource that records the file where it was loaded from.

interface:

```
@Class Resource isabstract extends Object
  @Attribute resourceName : String end
end
```

15.32 Class `Seq`

context: `Root::EMOF`
overview:

> Seq is a sub-class of DataType. All sequence data types are an instance of Seq. Seq defines an attribute 'elementType' that is used to record the type of the elements in a sequence data type.

interface:

```
@Class Seq extends DataType
  @Attribute elementType : Classifier end
  @Operation init end
  @Operation seqType end
  @Operation setElementType end
end
```

15.33 Class `Set`

context: `Root::EMOF`
overview:

> Set is a sub-class of DataType. All set data types are an instance of Set. Set defines an attribute 'elementType' that is used to record the type of the elements in a set data type.

interface:

```
@Class Set extends DataType
  @Attribute elementType : Classifier end
  @Operation init end
  @Operation setType end
  @Operation setElementType end
end
```

15.34 Class `StructuralFeature`

context: `Root::EMOF`
overview:

> This is an abstract class that is the super-class of all classes that describe structural features. For example Attribute is a sub-class of StructuralFeature. Other types of structural feature are possible by managing the internal structure of objects via a MOP.

interface:

```
@Class StructuralFeature isabstract extends TypedElement
end
```

15.35 DataType `Symbol`

overview:

> Symbol is a sub-class of String. Whereas there may be two different strings with the same sequence of characters, there can only be one symbol with the same sequence of characters. This is useful when using names as the basis for lookup (in tables). For example XMF ensures that classes, packages, operations, slots are named using symbols so that the lookup of these features by name is as efficient as possible. If strings were used the lookup would necessarily involve a character by character comparison. Using symbols the lookup can use the symbol's identity as the comparison operator. You can reference a symbol by constructing an instance: Symbol(name).

interface:

```
@DataType Symbol extends String
  @Operation init end
  @Operation toString end
end
```

15.36 DataType `Seq(Element)`

overview:

An instance of Seq where the element type is Element. This is the supertype of all sequence data types.

interface:

```
@DataType Seq(Element) extends Element
  @Operation at end
  @Operation asString end
  @Operation asSeq end
  @Operation asSet end
  @Operation append end
  @Operation butLast end
  @Operation contains end
  @Operation collect end
  @Operation drop end
  @Operation dot end
  @Operation default end
  @Operation drop end
  @Operation exists end
  @Operation excluding end
  @Operation forAll end
  @Operation flatten end
  @Operation head end
  @Operation hasSuffix end
  @Operation hasPrefix end
  @Operation iter end
  @Operation isProperSequence end
  @Operation isEmpty end
  @Operation includes end
  @Operation isKindOf end
  @Operation insertAt end
  @Operation indexOf end
  @Operation including end
  @Operation lookup end
  @Operation last end
  @Operation max end
  @Operation prepend end
  @Operation reverse end
  @Operation reject end
  @Operation subSequence end
  @Operation sortNames end
  @Operation sortNamedElements end
  @Operation sort end
  @Operation separateWith end
  @Operation select end
  @Operation size end
  @Operation toString end
  @Operation tail end
  @Operation take end
end
```

15.37 DataType `Seq(Element)`

overview:

An instance of Seq where the element type is Element. This is the supertype of all sequence data types.

interface:

```
@DataType Seq(Element) extends Element
  @Operation at end
  @Operation asString end
  @Operation asSeq end
  @Operation asSet end
  @Operation append end
  @Operation butLast end
  @Operation contains end
  @Operation collect end
  @Operation drop end
  @Operation dot end
  @Operation default end
  @Operation drop end
  @Operation exists end
  @Operation excluding end
  @Operation forAll end
  @Operation flatten end
  @Operation head end
  @Operation hasSuffix end
  @Operation hasPrefix end
  @Operation iter end
  @Operation isProperSequence end
  @Operation isEmpty end
  @Operation includes end
  @Operation isKindOf end
  @Operation insertAt end
  @Operation indexOf end
  @Operation including end
  @Operation lookup end
  @Operation last end
  @Operation max end
  @Operation prepend end
  @Operation reverse end
  @Operation reject end
  @Operation subSequence end
  @Operation sortNames end
  @Operation sortNamedElements end
  @Operation sort end
  @Operation separateWith end
  @Operation select end
  @Operation size end
  @Operation toString end
  @Operation tail end
  @Operation take end
end
```

15.38 DataType `Set(Element)`

overview:

An instance of Set where the element type is Element. This is the super-type of all set data types

interface:

```
@DataType Set(Element) extends Element
  @Operation asSet end
  @Operation asSeq end
  @Operation contains end
  @Operation collect end
  @Operation dot end
  @Operation default end
  @Operation exists end
  @Operation excluding end
  @Operation exists end
  @Operation flatten end
  @Operation iter end
  @Operation isKindOf end
  @Operation isEmpty end
  @Operation intersection end
  @Operation includes end
  @Operation including end
  @Operation max end
  @Operation reject end
  @Operation select end
  @Operation sel end
  @Operation select end
  @Operation size end
  @Operation toString end
  @Operation union end
end
```

15.39 DataType `Set(Element)`

overview:

An instance of Set where the element type is Element. This is the supertype of all set data types

interface:

```
@DataType Set(Element) extends Element
  @Operation asSet end
  @Operation asSeq end
  @Operation contains end
  @Operation collect end
  @Operation dot end
  @Operation default end
  @Operation exists end
  @Operation excluding end
  @Operation exists end
  @Operation flatten end
  @Operation iter end
  @Operation isKindOf end
  @Operation isEmpty end
  @Operation intersection end
  @Operation includes end
  @Operation including end
  @Operation max end
  @Operation reject end
  @Operation select end
  @Operation sel end
  @Operation select end
  @Operation size end
  @Operation toString end
  @Operation union end
end
```

15.40 DataType `String`

overview:

The data type for strings. The defualt value is the empty string.

interface:

```
@DataType String extends Element
  @Operation asSet end
  @Operation asSeq end
  @Operation asInt end
  @Operation asBool end
  @Operation default end
  @Operation hasSuffix end
  @Operation hasPrefix end
  @Operation isOlder end
  @Operation lookup end
  @Operation loadBin end
  @Operation padTo end
  @Operation repeat end
  @Operation subString end
  @Operation size end
  @Operation truncate end
  @Operation toString end
  @Operation upperCaseInitialLetter end
end
```

15.41 Class `Table`

context: `Root::EMOF`
overview:

> A table associates keys with values. Any element can be used as a key. A table has an initial size and can support any numbr of values. Use ’hesKey/1’ to determine whether a table contains a key. Use ’get/1’ to access a table via a key and ’put/2’ to update a table given a key and a value. Use ’keys/0’ to access the set of keys for a table.

interface:

```
@Class Table isabstract extends Element
  @Operation clear end
  @Operation get end
  @Operation get end
  @Operation hasKey end
  @Operation hasKey end
  @Operation init end
  @Operation keys end
  @Operation put end
  @Operation put end
  @Operation remove end
  @Operation remove end
  @Operation toString end
  @Operation values end
end
```

15.42 Class `TypedElement`

context: `Root::EMOF`
overview:

> A typed element is a named element with an associated type. The type is a classifier. This is an abstract class and is used (for example) to define Attribute.

interface:

```
@Class TypedElement isabstract extends NamedElement
  @Attribute type : Classifier end
end
```

16 Package `Exceptions`

context: `Root`
overview:

> This package defines a collection of exception classes. Instances of these classes are created when an exception occurs in XMF code. The exception is raised by throwing it to the most recently established catch.

16.1 Class `Error`

context: `Root::Exceptions`
overview:

> A general error exception. Use this when other exception classes are not provided for the specific exception you want to raise.

constructor: `Error(message)`

interface:

```
@Class Error extends Exception
  @Operation toString end
end
```

16.2 Class `NoSlot`

context: `Root::Exceptions`
overview:

> This exception is raised when a slot reference is performed on an object which does not define a slot of that name.

interface:

```
@Class NoSlot extends Exception
  @Attribute name : String end
  @Attribute object : Object end
  @Operation toString end
end
```

16.3 Class `NameSpaceRef`

context: `Root::Exceptions`
overview:

> This exception is raisd when a name space does not contain an element with a given name.

constructor: `NameSpaceRef(nameSpace,name)`

interface:

```
@Class NameSpaceRef extends Exception
  @Attribute name : String end
  @Attribute nameSpace : NameSpace end
  @Operation toString end
end
```

16.4 Class `PathNotFound`

context: `Root::Exceptions`
overview:

> This exception is raisd when a path cannot be de-referenced with respect to the currently imported name spaces.

constructor: `PathNotFound(path,imports)`

interface:

```
@Class PathNotFound extends Exception
  @Attribute imports : Seq(NameSpace) end
  @Attribute path : Element end
  @Operation toString end
end
```

16.5 Class `TypeError`

context: `Root::Exceptions`
overview:

> This exception is thrown when a value is expected to be of a particular type and is not.

constructor: `TypeError(value,type)`

interface:

```
@Class TypeError extends Exception
  @Attribute type : Classifier end
  @Attribute value : Element end
  @Operation toString end
end
```

16.6 Class `UnboundVar`

context: `Root::Exceptions`

interface:

```
@Class UnboundVar extends Exception
  @Attribute name : String end
  @Operation toString end
end
```

17 Package Grammars

context: Root

17.1 Class PathRef

context: Root::Grammars

interface:

```
@Class PathRef extends Object
  @Attribute path : Seq(String) end
end
```

18 Package `Instrs`

context: `Root`
overview:

> The package Instrs contains the definition of all XMF machine instructions. Each machine instruction has an 8-bit code used by the XMF machine. The codes are defined in the operation Instr::code/0. Each instruction defines a collection of operations that are used by the compiler and assembler.

18.1 Class `AsSeq`

context: `Root::Instrs`

interface:

```
@Class AsSeq extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.2 Class `At`

context: `Root::Instrs`

interface:

```
@Class At extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.3 Class `And`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs an 'and' operation. The machine knows how to perform 'and' for boolean operands. If the machine does not understand the operand types then 'and/1' is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class And extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.4 Class `Add`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs an + operation. Many + operations are builtin to the machine. If the machine does not understand the operand types then ’add/1’ is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Add extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.5 Class `Cons`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two operands at the head of the stack. Both operands are popped. The top element is a sequence ’t’ and the element below is ’h’. The instruction pushes the sequence Seqh — t back on the stack.

interface:

```
@Class Cons extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.6 Class `Dynamic`

context: `Root::Instrs`
overview:

> A machine instruction that looks up the value of a dynamic variable. The offset into the constants array in the currently executing code box is in the instruction word. This indexes a symbol that is used to lookup a value in the current dynamics list starting in the current machine stack frame. This will lookup the symbol name in all currently imported name spaces (actually hash tables in the dynamics list). The instruction leaves the value of the dyanamic on the stack. An error will occur of a dynamic with the appropriate name cannot be found.

interface:

```
@Class Dynamic extends Instr
  @Attribute nameOffset : Integer end
  @Attribute name : Symbol end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.7 Class `Dot`

context: `Root::Instrs`
overview:

> A machine instruction that performs object navigation. The instruction has an operand in the machine word that is the offset in the constants array of the current code box indexing a symbol. The instruction expects a value at the head of the stack. The value is popped. If the type of the element is known by the machine (typically an object) then the slot value indexed by the symbol is pushed onto the stack. Otherwise the instruction causes a 'dot/1' message to be sent to the element. A value is left at the head of the stack.

interface:

```
@Class Dot extends Instr
  @Attribute constOffset : Integer end
  @Attribute name : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.8 Class `Div`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs a / operation. Many / operations are builtin to the machine. If the machine does not understand the operand types then 'div/1' is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Div extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.9 Class `Drop`

context: `Root::Instrs`

interface:

```
@Class Drop extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.10 Class `Excluding`

context: `Root::Instrs`

interface:

```
@Class Excluding extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.11 Class `Enter`

context: `Root::Instrs`
overview:

> A machine instruction that closes a currently open call frame and switches machine context to start to execute the instructions in the new call frame in the context of the locals in that frame. When an operation is called, a new frame is opened, the arguments are pushed, the operation is pushed and then the frame is entered. Entering the frame switches context by updating the machine registers such as CurrentFrame to point to the newly constructed frame. When the operation returns the frame is discarded and control passes to the previously open frame.

interface:

```
@Class Enter extends Instr
  @Attribute arity : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.12 Class `Eql`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs a = operation. The '=' operation is implemented by the machine. Objects are compared by identity, sequences are compared by identity (since they can be updated by side effect), sets are compared by structure (recursively applying '=' to elements), symbols are copared by identity, strings are compared by structure and all atomic data items are compared by value. The boolean result is left at the head of the stack.

interface:

```
@Class Eql extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.13 Class `Greater`

context: `Root::Instrs`
overview:

> A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs a ¿ operation. If the elements are numbers then ¿ will push the appropriate boolean value. If the machine does not understand the operand types then ’gre/1’ is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Greater extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.14 Class `GlobalRef`

context: `Root::Instrs`
overview:

> A global ref occurs when a local variable is referenced by an operation where the variable is declared outside the operation. When the operation is created, all local variables from the enclosing scope are added to the dynamics array of the new operation. Dynamics arrays are linked, so that a newly created dynamics array is linked to the dynamics array of the enclosing scope. When a dynamic variable is referenced in the body of an operation, the value is found by chaining back through the dynamic array linked list that is in the current stack frame and then indexing into the resulting array. This instruction has machine word operands that statically determine the ’frame’ (i.e. how many links to traverse) and the ’offset’ (i.e. the index into the resulting array). The instruction leaves a value on the stack.

interface:

```
@Class GlobalRef extends Instr
  @Attribute frame : Integer end
  @Attribute name : String end
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.15 Class `GetElement`

context: `Root::Instrs`

interface:

```
@Class GetElement extends Instr
  @Attribute nameOffset : Integer end
  @Attribute name : Symbol end
  @Operation assemble end
  @Operation operands end
  @Operation toString end
end
```

18.16 Class `Head`

context: `Root::Instrs`
overview:

> This machine instruction expects a sequence at the head of the stack. It pops the sequence and pushes the head of the sequence back on the stack. If the value is not a sequence then the instruction sends a 'head/0' message to the value.

interface:

```
@Class Head extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.17 Class `Including`

context: `Root::Instrs`

interface:

```
@Class Including extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.18 Class `Instr`

context: `Root::Instrs`
overview:

> A machine instruction is an instance of the abstract class Instr. Each machine instruction has a code in the range 1 - 255. Machine instructions are of the form:
>
> `CODE DATA1 DATA2 DATA3`

where each element is an 8 bit value. The high byte is the machine byte code. The rest of the machine instruction word may contain up to 3 operands encoded as bytes. Typically these may be constants or indices into parts of the current machine stack frame. The class Instr allocates the instruction byte codes. These should correspond to the byte codes used by the underlying machine implementation.

interface:

```
@Class Instr isabstract extends Object
  @Operation assemble end
  @Operation code end
  @Operation operands end
end
```

18.19 Class `IsEmpty`

context: `Root::Instrs`

interface:

```
@Class IsEmpty extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.20 Class `Implies`

context: `Root::Instrs`
overview:

A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs an implies operation. If the operands are boolean then the machine performs the operation directly, pushing the value on the stack. If the machine does not understand the operand types then ’implies/1’ is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Implies extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.21 Class `Includes`

context: `Root::Instrs`

interface:

```
@Class Includes extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.22 Class `IsKindOf`

context: `Root::Instrs`

interface:

```
@Class IsKindOf extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.23 Class `Less`

context: `Root::Instrs`
overview:

A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs a ¡ operation. If the elements are numbers then ¡ will push the appropriate boolean value. If the machine does not understand the operand types then 'less/1' is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Less extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.24 Class `LocalRef`

context: `Root::Instrs`
overview:

This machine instruction references a local variable location in the current stack frame. The index of the local variab;e location is in the machine instruction word. The instruction leaves the local value on the top of the stack.

interface:

```
@Class LocalRef extends Instr
  @Attribute name : String end
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.25 Class `MkSeq`

context: `Root::Instrs`
overview:

A machine instruction that expects a collection of values on the stack. The machine instruction word has a single operand that is the number of elements to pop off the stack. the values are formed into a sequence which is then pushed onto the stack.

interface:

```
@Class MkSeq extends Instr
  @Attribute size : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.26 Class `Mul`

context: `Root::Instrs`
overview:

A machine instruction. Expects two elements at the head of the stack. Pops the elements and performs a * operation. Many * operations are builtin to the machine. If the machine does not understand the operand types then 'mul/1' is sent to the left hand operand. The result is left at the head of the stack.

interface:

```
@Class Mul extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.27 Class `MkOp`

context: `Root::Instrs`
overview:

Thismachine instruction constructs an operation. The machine instruction word has two operands being the indices in the constants array of the current machine stack frame for the name of the operation and the code box of the function. The instruction expects a number of values on the stack corresponding to the globals that are popped into a freshly allocated array. The array is linked to the globals array in the current stack frame and becomes the globals array for the new operation. The new operation is pushed onto the stack.

interface:

```
@Class MkOp extends Instr
  @Attribute arity : Integer end
  @Attribute codeBoxOffset : Integer end
  @Attribute code : Seq(Element) end
  @Attribute free : Integer end
  @Attribute locals : Integer end
  @Attribute nameOffset : Integer end
  @Attribute name : String end
  @Attribute source : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.28 Class `MkSet`

context: `Root::Instrs`
overview:

> A machine instruction that expects a collection of values on the stack. The machine instruction word has a single operand that is the number of elements to pop off the stack. the values are formed into a set which is then pushed onto the stack.

interface:

```
@Class MkSet extends Instr
  @Attribute size : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.29 Class `Negate`

context: `Root::Instrs`

interface:

```
@Class Negate extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.30 Class `NameSpaceRef`

context: `Root::Instrs`
overview:

> A name space contains definitions. When a name space and its definitions are statically defined, it is possible for the definitions to be mutually recursive. Each name is scoped over all the other definitions in the name space. Since name spaces can contain nested name spaces, names are scoped over nested definitions. This machine instruction is used to index a name that is statically declared in an enclosing name space. Name space contents are linked to their enclosing name space using their ’owner’ slot. The instruction contains two operands in the instruction word. The contour defines how many ’owner’ slots must be traversed to get to the appropriate name space. The name to reference is given by the second operand which is an offset into the current constants array in the stack frame. The instruction

either signals an error if the name is not found or pushes the value of the name space reference onto the stack.

interface:

```
@Class NameSpaceRef extends Instr
  @Attribute contour : Integer end
  @Attribute nameOffset : Integer end
  @Attribute name : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.31 Class `Null`

context: `Root::Instrs`

interface:

```
@Class Null extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.32 Class `Or`

context: `Root::Instrs`

interface:

```
@Class Or extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.33 Class `Of`

context: `Root::Instrs`

interface:

```
@Class Of extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.34 Class `PushFalse`

context: `Root::Instrs`
overview:

A machine instruction that pushes ’false’ onto the stack.

interface:

```
@Class PushFalse extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.35 Class `PushInteger`

context: `Root::Instrs`
overview:

> A machine instruction that pushes an integer value onto the stack. The integer is encoded in the machine word as an instruction operand.

interface:

```
@Class PushInteger extends Instr
  @Attribute value : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.36 Class `PushTrue`

context: `Root::Instrs`
overview:

> A machine instruction that pushes ’true’ onto the stack.

interface:

```
@Class PushTrue extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.37 Class `Pop`

context: `Root::Instrs`
overview:

> A machine instruction that pops the stack.

interface:

```
@Class Pop extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.38 Class `PushString`

context: `Root::Instrs`
overview:

> A machine instruction that pushes a string onto the stack. The string is indexed by the instruction word operand in the currentstack frame constants array.

interface:

```
@Class PushString extends Instr
  @Attribute valueOffset : Integer end
  @Attribute value : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.39 Class `Return`

context: `Root::Instrs`
overview:

> A machine instruction that returns from the current operation call. The value at the top of the stack is popped and the current stack frame is discarded (returning to the most recently pushed stack frame). The value is then pushed onto the top of the stack. This must be the last instruction executed in the body of an operation.

interface:

```
@Class Return extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.40 Class `Read`

context: `Root::Instrs`
overview:

> A machine instruction that performs an non blocking read on the input channel found at the head of the stack.

interface:

```
@Class Read extends Instr
  @Operation assemble end
  @Operation toString end
end
```

18.41 Class `Skip`

context: `Root::Instrs`

interface:

```
@Class Skip extends Instr
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.42 Class `Self`

context: `Root::Instrs`
overview:

References the current value of self. This is held in the current stack frame. The value is pushed onto the stack.

interface:

```
@Class Self extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.43 Class `SetGlobal`

context: `Root::Instrs`

interface:

```
@Class SetGlobal extends Instr
  @Attribute frame : Integer end
  @Attribute name : String end
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation operands end
  @Operation toString end
end
```

18.44 Class `SourcePos`

context: `Root::Instrs`

interface:

```
@Class SourcePos extends Instr
  @Attribute charCount : Integer end
  @Attribute lineCount : Integer end
  @Operation assemble end
  @Operation operands end
  @Operation toString end
end
```

18.45 Class `SetHead`

context: `Root::Instrs`

interface:

```
@Class SetHead extends Instr
  @Operation assemble end
  @Operation toString end
end
```

18.46 Class `SetLocal`

context: `Root::Instrs`

interface:

```
@Class SetLocal extends Instr
  @Attribute name : String end
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.47 Class `Super`

context: `Root::Instrs`

interface:

```
@Class Super extends Instr
  @Attribute arity : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
end
```

18.48 Class `Sub`

context: `Root::Instrs`

interface:

```
@Class Sub extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.49 Class `SkipBack`

context: `Root::Instrs`

interface:

```
@Class SkipBack extends Instr
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.50 Class `Size`

context: `Root::Instrs`

interface:

```
@Class Size extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.51 Class `Sel`

context: `Root::Instrs`

interface:

```
@Class Sel extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.52 Class `Send`

context: `Root::Instrs`
overview:

> This machine inbstruction sends a message to a target. The message has a name and some arguments. The target is at the top of the stack above the arguments. The instruction word contains the arity and the offset of the message in the current constants array. The message name must be a symbol. The instruction is performed after a new stack frame has been opened and the arguments and target have been pushed. The instruction then enters the stack frame after finding the operation appropriate to the target. If no operation is defined then an error is signaled.

interface:

```
@Class Send extends Instr
  @Attribute arity : Integer end
  @Attribute messageOffset : Integer end
  @Attribute message : Symbol end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.53 Class `SkipFalse`

context: `Root::Instrs`

interface:

```
@Class SkipFalse extends Instr
  @Attribute offset : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.54 Class `SetTail`

context: `Root::Instrs`

interface:

```
@Class SetTail extends Instr
  @Operation assemble end
  @Operation toString end
end
```

18.55 Class `StartCall`

context: `Root::Instrs`

interface:

```
@Class StartCall extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.56 Class `SetSlot`

context: `Root::Instrs`

interface:

```
@Class SetSlot extends Instr
  @Attribute nameOffset : Integer end
  @Attribute name : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.57 Class `TailSend`

context: `Root::Instrs`

interface:

```
@Class TailSend extends Instr
  @Attribute arity : Integer end
  @Attribute messageOffset : Integer end
  @Attribute message : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.58 Class `TailSuper`

context: `Root::Instrs`

interface:

```
@Class TailSuper extends Instr
  @Attribute arity : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.59 Class `Try`

context: `Root::Instrs`

interface:

```
@Class Try extends Instr
  @Attribute body : Seq(Instr) end
  @Attribute codeBoxOffset : Integer end
  @Attribute freeVars : Integer end
  @Attribute locals : Integer end
  @Attribute source : String end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
end
```

18.60 Class `Tail`

context: `Root::Instrs`

interface:

```
@Class Tail extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

18.61 Class `Throw`

context: `Root::Instrs`

interface:

```
@Class Throw extends Instr
  @Operation assemble end
  @Operation toString end
end
```

18.62 Class `TailEnter`

context: `Root::Instrs`

interface:

```
@Class TailEnter extends Instr
  @Attribute arity : Integer end
  @Operation assemble end
  @Operation init end
  @Operation operands end
  @Operation toString end
  @Operation write end
end
```

18.63 Class `Union`

context: `Root::Instrs`

interface:

```
@Class Union extends Instr
  @Operation assemble end
  @Operation toString end
  @Operation write end
end
```

19 Package `IO`

context: `Root`
overview:

> The IO package provides classes and operations that support input and output. XMF uses input and output channels that can be used as the sources and sinks of characters. The channel stdout is used to write to the standard output. The operation 'format' is provided to control formatted output.

19.1 Class `ElementOutputChannel`

context: `Root::IO`

interface:

```
@Class ElementOutputChannel extends OutputChannel
  @Attribute out : OutputChannel end
  @Operation close end
  @Operation channel end
  @Operation flush end
  @Operation toString end
  @Operation write end
end
```

19.2 Class `EvalInputChannel`

context: `Root::IO`
overview:

> This channel is initialised with a grammar, an input channel and a sequence of imports. When requested for a value via 'readValue', the channel will parse the next input from the channel and evaluate it.

constructor: `EvalInputChannel(grammar,start,channel,imports)`

interface:

```
@Class EvalInputChannel extends InputChannel
  @Attribute channel : InputChannel end
  @Attribute env : Element end
  @Attribute grammar : Element end
  @Attribute imports : Seq(NameSpace) end
  @Attribute state : Element end
  @Attribute start : String end
  @Operation asString end
  @Operation close end
  @Operation channel end
  @Operation eof end
  @Operation peek end
  @Operation readValue end
  @Operation read end
end
```

19.3 Class `ElementInputChannel`

context: `Root::IO`

interface:

```
@Class ElementInputChannel extends InputChannel
  @Attribute input : InputChannel end
  @Operation close end
  @Operation peek end
  @Operation read end
  @Operation toString end
end
```

19.4 Class `FileOutputChannel`

context: `Root::IO`
overview:

A file output channel is used to write chars to a file.

constructor: `FileOutputStream(fileName:String)`

Creates and returns a file output stream to the named file. The file should be passed as a pathname to the file. The file will be created if it does not already exist.

interface:

```
@Class FileOutputChannel extends OutputChannel
  @Attribute channel : Element end
  @Operation close end
  @Operation channel end
  @Operation init end
  @Operation writeChar end
end
```

19.5 Class `Format`

context: `Root::IO`
overview:

This cass implements a formatter. A formatter is applied to an output channel, a control string and an optional sequence of argments. The formatter prints out the chars of the control string with respect to the arguments. Control chars are introduced via ’ ’ and invoke the appropriate control char handler.

interface:

```
@Class Format extends Object
  @Attribute handlers : Table end
  @Operation dispatch end
  @Operation defineHandler end
  @Operation format end
  @Operation invoke end
  @Operation init end
end
```

19.5.1 Operation `format`

context: `Root::IO::Format`
overview:

> This is the formatting loop. While there are control chars: if the char is ' ' then jump to the handler otherwise just print the char and go round again.

19.6 Class `FileInputChannel`

context: `Root::IO`
overview:

> A FileInputChannel generates a sequence of characters from a file.

constructor: `FileInputChannel(path)`

> Creates and returns a file input channel from the given file. The path should be string.

interface:

```
@Class FileInputChannel extends InputChannel
  @Attribute channel : Element end
  @Operation asString end
  @Operation close end
  @Operation channel end
  @Operation eof end
  @Operation init end
  @Operation peek end
  @Operation read end
end
```

19.7 Class `FormatContext`

context: `Root::IO`
overview:

> A formatter uses a context to control formatting. The context is created on each call to a formatter and is updated during formatting. The format context is supplied to format handlers so that thay can consume arbitrary values and control string characters.

constructor: `FormatContext(control,values,index)`

> The control is a control string, the values area sequence of values controlled by the control string and the index is an integer index into the control string.

interface:

```
@Class FormatContext extends Object
  @Attribute args : Seq(Element) end
  @Attribute control : String end
  @Attribute index : Integer end
  @Attribute values : Seq(Element) end
  @Operation appendValues end
  @Operation appendControl end
  @Operation complete end
  @Operation controlTo end
  @Operation init end
  @Operation lift end
  @Operation nextValue end
  @Operation nextChar end
  @Operation peekChar end
  @Operation parseNumeric end
  @Operation parseArgs end
  @Operation parseArg end
end
```

19.7.1 Operation `controlTo`

context: `Root::IO::FormatContext`
overview:

> Returns the string up to and including the supplied control char. Removes the control chars from the head of the control string. Note that this does not take into account nested control strings.

19.8 Class `InputChannel`

context: `Root::IO`
overview:

> An input channel is a character source. Use ’read/1’ to get the next character code from the channel and use ’eof/0’ to test whether the channel is exhausted.

interface:

```
@Class InputChannel isabstract extends Object
  @Operation close end
  @Operation channel end
  @Operation eof end
  @Operation peek end
  @Operation read end
  @Operation readNonWhiteSpace end
end
```

19.8.1 Operation `readNonWhiteSpace`

context: `Root::IO::InputChannel`
overview:

> Returns the next non-whitespace character on the input channel.

19.9 Class `OutputChannel`

context: `Root::IO`
overview:

> An output channel is a sink of characters for output. Use output channels with an output formatter such as ’format’.

interface:

```
@Class OutputChannel isabstract extends Object
  @Operation close end
  @Operation channel end
  @Operation flush end
  @Operation writeChar end
  @Operation writeString end
end
```

19.10 Class `StringOutputChannel`

context: `Root::IO`
overview:

> An output channel that buffers its chars up in a string. Use 'getString/0' to get the string from the channel.

constructor: `StringOutputChannel()`

interface:

```
@Class StringOutputChannel extends OutputChannel
  @Attribute chars : Seq(Integer) end
  @Operation close end
  @Operation getString end
  @Operation writeChar end
end
```

19.10.1 Operation `close`

context: `Root::IO::StringOutputChannel`
overview:

> Call this operation when use of the channel is complete.

19.10.2 Operation `getString`

context: `Root::IO::StringOutputChannel`
overview:

> Get the string managed by a string output channel.

19.11 Class `StandardOutputChannel`

context: `Root::IO`

interface:

```
@Class StandardOutputChannel extends OutputChannel
  @Attribute channel : Element end
  @Operation close end
  @Operation channel end
  @Operation flush end
  @Operation writeChar end
end
```

19.12 Class `StandardInputChannel`

context: `Root::IO`

interface:

```
@Class StandardInputChannel extends InputChannel
  @Attribute channel : Element end
  @Operation asString end
  @Operation close end
  @Operation channel end
  @Operation eof end
  @Operation peek end
  @Operation read end
end
```

19.13 Class `TokenInputChannel`

context: `Root::IO`

interface:

```
@Class TokenInputChannel extends InputChannel
  @Attribute input : InputChannel end
  @Operation close end
  @Operation channel end
  @Operation eof end
  @Operation peek end
  @Operation readToken end
  @Operation readSymbol end
  @Operation readString end
  @Operation readNumber end
  @Operation read end
end
```

19.14 Class `XMLInputChannel`

context: `Root::IO`

interface:

```
@Class XMLInputChannel extends InputChannel
  @Attribute input : InputChannel end
  @Operation close end
  @Operation read end
  @Operation toString end
end
```

19.15 Class `XMLOutputChannel`

context: `Root::IO`
overview:

An XML output channel writes an XML encoding of any element to a supplied base output channel. The encoding conforms to XMF.dtd.

constructor: `XMLOutputChannel(out)`

The argument out is an output channel to which the XML characters are written when an element is written to the XML output channel.

interface:

```
@Class XMLOutputChannel extends OutputChannel
  @Attribute out : OutputChannel end
  @Attribute walker : XML end
  @Operation close end
  @Operation toString end
  @Operation writeChar end
  @Operation write end
end
```

19.16 Operation `Kernel_tableKeys`

context: `Root`

19.17 Operation `Kernel_codeBoxSetResourceName`

context: `Root`

19.18 Operation `Kernel_of`

context: `Root`

19.19 Operation `Kernel_setDaemons`

context: `Root`

19.20 Operation `Kernel_mk24bit`

context: `Root`

19.21 Operation `Kernel_asString`

context: `Root`

19.22 Operation `Kernel_funCodeBox`

context: `Root`

19.23 Operation `Kernel_daemonsOff`

context: `Root`

19.24 Operation `Kernel_nextToken`

context: `Root`

19.25 Operation `Kernel_tableGet`

context: `Root`

19.26 Operation `Kernel_yield`

context: `Root`

19.27 Operation `Kernel_codeBoxSetConstants`

context: `Root`

19.28 Operation `Kernel_load`

context: `Root`

19.29 Operation `Kernel_server_listen`

context: `Root`

19.30 Operation `Kernel_asSet`

context: `Root`

19.31 Operation `Kernel_fileOutputChannel`

context: `Root`

19.32 Operation `Kernel_daemons`

context: `Root`

19.33 Operation `Kernel_mkSymbol`

context: `Root`

19.34 Operation `Kernel_stackFrames`

context: `Root`

19.35 Operation `Kernel_writeChar`

context: `Root`

19.36 Operation `Kernel_isOlder`

context: `Root`

19.37 Operation `Kernel_asSeq`

context: `Root`

19.38 Operation `Kernel_save`

context: `Root`

19.39 Operation `Kernel_codeBoxName`

context: `Root`

19.40 Operation `Kernel_codeSet`

context: `Root`

19.41 Operation `Kernel_fileInputChannel`

context: `Root`

19.42 Operation `Kernel_mkObj`

context: `Root`

19.43 Operation `Kernel_stats`

context: `Root`

19.44 Operation `Kernel_valueToString`

context: `Root`

19.45 Operation `Kernel_invoke`

context: `Root`

19.46 Operation `Kernel_readXML`

context: `Root`

19.47 Operation `Kernel_arraySet`

context: `Root`

19.48 Operation `Kernel_codeBoxConstants`

context: `Root`

19.49 Operation `Kernel_codeBoxToFun`

context: `Root`

19.50 Operation `Kernel_mkTable`

context: `Root`

19.51 Operation `Kernel_slotNames`

context: `Root`

19.52 Operation `Kernel_import`

context: `Root`

19.53 Operation `Kernel_traceFrames`

context: `Root`

19.54 Operation `Kernel_readString`

context: `Root`

19.55 Operation `Kernel_addAtt`

context: `Root`

19.56 Operation `Kernel_codeBoxSetCode`

context: `Root`

19.57 Operation `Kernel_mkString`

context: `Root`

19.58 Operation `Kernel_codeBoxSource`

context: `Root`

19.59 Operation `Kernel_letVar`

context: `Root`

19.60 Operation `Kernel_size`

context: `Root`

19.61 Operation `Kernel_hasVar`

context: `Root`

19.62 Operation `Kernel_tokenChannelTextTo`

context: `Root`

19.63 Operation `Kernel_read`

context: `Root`

(c) 2003 Xactium Ltd.

19.64 Operation `Kernel_mkCodeBox`

context: `Root`

19.65 Operation `Kernel_flush`

context: `Root`

19.66 Operation `Kernel_setTypes`

context: `Root`

19.67 Operation `Kernel_close`

context: `Root`

19.68 Operation `Kernel_codeBoxSetSource`

context: `Root`

19.69 Operation `Kernel_getVar`

context: `Root`

19.70 Operation `Kernel_peek`

context: `Root`

19.71 Operation `Kernel_tableHasKey`

context: `Root`

19.72 Operation `Kernel_loadbin`

context: `Root`

19.73 Operation `Kernel_fork`

context: `Root`

19.74 Operation `Kernel_mkCode`

context: `Root`

19.75 Operation `Kernel_setSlotValue`

context: `Root`

19.76 Operation `Kernel_client_connect`

context: `Root`

19.77 Operation `Kernel_getSlotValue`

context: `Root`

19.78 Operation `Kernel_operatorPrecedenceList`

context: `Root`

19.79 Operation `Kernel_codeBoxSetName`

context: `Root`

19.80 Operation `Kernel_tableRemove`

context: `Root`

19.81 Operation `Kernel_exit`

context: `Root`

19.82 Operation `Kernel_mkBasicTokenChannel`

context: `Root`

19.83 Operation `Kernel_setCharAt`

context: `Root`

19.84 Operation `Kernel_backtrace`

context: `Root`

19.85 Operation `Kernel_funSetSupers`

context: `Root`

19.86 Operation `Kernel_codeBoxResourceName`

context: `Root`

19.87 Operation `Kernel_operationCodeBox`

context: `Root`

19.88 Operation `Kernel_eof`

context: `Root`

19.89 Operation `Kernel_tablePut`

context: `Root`

19.90 Operation `Kernel_setOf`

context: `Root`

19.91 Operation `Kernel_mkArray`

context: `Root`

19.92 Operation `Kernel_available`

context: `Root`

19.93 Operation `Kernel_funGlobals`

context: `Root`

19.94 Operation `Kernel_daemonsOn`

context: `Root`

20 Package `OCL`

context: `Root`
overview:

> The OCL package contains the concrete and abstract syntax definitions for the OCL language as supplied by XMF. The OCL package is intended to be consistent with the OCL standard where this is practical. OCL is not the only language supported by XMF but it is used as the basis for the implementation of XMF. As such it has a number of differences from and extensions to standard OCL. Each of the classes in OCL define how they are compiled and interpreted on the XMF VM. The oCL package provides many of the features that are useful for a wide variety of domain specific languages. It is recommended that you base new languages on OCL either as extensions or as the target of a desugaring.

20.1 Class `Apply`

context: `Root::OCL`
overview:

> An application expression consists of an operator and some operands, all of which are expressions. Each expression is performed and the operator expression should result in a value that supports the operation 'invoke/2'. Typically, the operator will be an instance of the class EMOF::Operation whose invoke operation is supported natively by the XMF machine. The effect of the application is to invoke the operator on the operands by supplying them as arguments.

constructor: `Apply(operator,operands)`

> The operator is an expression and the operands are a sequence of expressions.

syntax:

```
Apply ::= Exp '(' [ Exp (',' Exp)* ')'
```

example:

```
let f = @Operator(x,y) x + y end
in f(10,20)
end
```

interface:

```
@Class Apply extends OCL
  @Attribute args : Seq(OCL) end
  @Attribute operator : OCL end
  @Operation FV end
  @Operation FV end
  @Operation compileSuper end
  @Operation compileCall end
  @Operation compileBuiltin end
  @Operation compile end
  @Operation compileSuper end
  @Operation compileCall end
  @Operation compileBuiltin end
  @Operation compile end
  @Operation eval end
  @Operation isSuper end
  @Operation isBuiltin end
  @Operation isSuper end
  @Operation isBuiltin end
  @Operation lift end
  @Operation lift end
  @Operation maxLocals end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.2 Class `BoolExp`

context: `Root::OCL`
overview:

A boolean constant expression: either t true or t false.

constructor: `BoolExp(boolValue)`

The operand is the boolean constant.

syntax:

```
        BoolExp ::= ’true’ | ’false’.
```

interface:

```
@Class BoolExp extends OCL
  @Attribute value : Boolean end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.3 Class `Binding`

context: `Root::OCL`
overview:

Binding is an abstract class that represents an association between a name and a value. Bindings occur in syntax expressions such as let.

interface:

```
@Class Binding extends OCL
  @Attribute name : String end
end
```

20.4 Class `BinExp`

context: `Root::OCL`
overview:

> A binary expression consists of two sub-expressions and a binary operator. XMT support extensible operators. For example if '+' is supplied with integer operands then the XMT machine handles this natively. Otherwise, if the machine does not understand the operand types, the left handl operand is send a message including the right operand. This allows the user to overload the binary operators for their own types.

constructor: `BinExp(left,binOp,right)`

> The left and right operands are expressions. The binary operator is a string that names the operator. This should be one of the following:

```
+,-,*,/,and,or,implies,xor,implies,>,<,>=,<=,=
```

syntax:

```
 BinExp ::= Exp BinOp Exp.
```

example:

```
10 < 20 or x = "hello"
```

interface:

```
@Class BinExp extends OCL
  @Attribute binOp : String end
  @Attribute left : OCL end
  @Attribute right : OCL end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.5 Class `CmpEntry`

context: `Root::OCL`

interface:

```
@Class CmpEntry extends Object
end
```

20.6 Class `ConsExp`

context: `Root::OCL`
overview:

A cons expression constructs a sequence from a head and tail.

constructor: `ConExp(head,tail)`

The head and tail arguments are expressions. The head expression denotes an arbitrary value and the tail expression denotes a sequence (possibly the empty sequence).

syntax:

```
ConsExp ::= 'Seq{' Exp '|' Exp '}'.
```

example:

```
Seq{1 | 2}
Seq{1 | Seq{2 | Seq{}}}
```

interface:

```
@Class ConsExp extends OCL
  @Attribute head : OCL end
  @Attribute tail : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.7 Class `Case`

context: `Root::OCL`

interface:

```
@Class Case extends OCL
  @Attribute clauses : Seq(Element) end
  @Attribute value : OCL end
  @Operation init end
end
```

20.8 Class `Consp`

context: `Root::OCL`
overview:

A cons pattern consists of a head and a tail pattern. A value matches a cons pattern when the value is a cons pair and the head of the pair matches the head of the pattern and the tail of the pair matches the tail of the pattern. Cons patterns can be sugared so that they look like conventional sequences.

constructor: `Consp(head,tail)`

The head and tail are patterns.

syntax:

```
Consp ::= Pairp | Seqp.
Pairp ::= 'Seq' '{' Pattern '|' Pattern '}'.
Seqp ::= 'Seq' '{' Pattern (',' Pattern)* '}'.
```

interface:

```
@Class Consp extends Pattern
  @Attribute head : Pattern end
  @Attribute tail : Pattern end
  @Operation bind end
  @Operation lift end
  @Operation matchCode end
  @Operation pprint end
end
```

20.9 Class `CmpCondition`

context: `Root::OCL`

interface:

```
@Class CmpCondition extends CmpEntry
  @Attribute condition : OCL end
end
```

20.10 Class `ContextDef`

context: `Root::OCL`
overview:

Definitions are evaluated and (typically) added to a container. The 'context' construct is a declarative way of specifying that a given definition should be added to a particular container. The context construct should occur at the top level in files. If the context is a name space then the context expression does not cause the name space to be imported.

constructor: `ContextDef(path,def)`

The path is an expression that results in a name space and the def is an expression denoting a named element.

syntax:

```
ContextDef ::= 'context' Exp Exp.
```

example:

```
context Root
  @Operation myOp()
    format(stdout,"a global operation")
  end
```

interface:

```
@Class ContextDef extends OCL
  @Attribute element : OCL end
  @Attribute path : OCL end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation init end
  @Operation maxLocals end
  @Operation pprint end
end
```

20.11 Class `CmpBinding`

context: `Root::OCL`

interface:

```
@Class CmpBinding extends CmpEntry
  @Attribute name : String end
  @Attribute value : OCL end
end
```

20.12 Class `Constp`

context: `Root::OCL`
overview:

> A constant pattern is either an integer, a string a boolean or an empty collection.

constructor: `Constp(const)`

> The expression defines the constant.

syntax:

```
       Constp ::= Int | Str | Bool | ’Seq’ ’{’ ’}’ | ’Set’ ’{’ ’}’.
```

interface:

```
@Class Constp extends Pattern
  @Attribute const : Performable end
  @Operation bind end
  @Operation lift end
  @Operation matchCode end
  @Operation pprint end
end
```

20.13 Class `Comprehension`

context: `Root::OCL`

interface:

```
@Class Comprehension extends Object
  @Attribute body : OCL end
  @Attribute entries : Seq(CmpEntry) end
  @Attribute name : String end
end
```

20.14 Class `Const`

context: `Root::OCL`

interface:

```
@Class Const extends OCL
  @Attribute value : Element end
  @Operation FV end
  @Operation compile end
  @Operation init end
  @Operation maxLocals end
end
```

20.15 Class `CollExp`

context: `Root::OCL`
overview:

> A collection expression represents sending a message to a collection. The compiler knows about certain collection operations that get compiled to machine instructions. Otherwise a collection expression just sends a message to the collection. Note that when the message has no arguments, the parentheses can be omitted.

constructor: `CollExp(coll,collOp,operands`

> The coll argument is an expression whose value is a collection. the collOp is a string naming the message to send, and the operands is a sequence of expressions.

syntax:

```
       CollExp ::= Exp '->' Name [ '(' Exp (',' Exp)* ')' ].
```

example:

```
       Set{1,2,3}->size
       Set{1,2,3}->including(4)
```

interface:

```
@Class CollExp extends OCL
  @Attribute args : Seq(OCL) end
  @Attribute collOp : String end
  @Attribute collection : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.16 Class `Dot`

context: `Root::OCL`
overview:

> A dot expression is a slot navigation. The target of the dot can be any value. If the value is a collection then the dot operator is applied to each element of the collection and the resulting values are flattened if necessary. If the value is an object then the corresponding slot value is returned or an error is signaled. If the value is not an object then either the machine knows how to resolve the slot reference or the message 'slot/1' is sent to the value.

constructor: `Dot(target,name)`

> The target is an expression and the name is a symbol.

syntax:

```
 Dot ::= Exp '.' Name.
```

example:

```
o.x
Set{o1,o2,o3}.x
@Operation(x) x + 1 end.arity
```

interface:

```
@Class Dot extends OCL
  @Attribute name : String end
  @Attribute target : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.17 Class `DropCollection`

context: `Root::OCL`

interface:

```
@Class DropCollection extends OCL
  @Attribute exp : OCL end
  @Operation init end
end
```

20.18 Class `Drop`

context: `Root::OCL`
overview:

A drop expression may occur within a lift expression [— ... —]. A lift expression is shorthand for the construction of an expression that will build the AST for the expression contained in the [— ... —]. Drop expressions occur between ¡ ... ¿ in lift expressions, the syntax is not constructed for a drop expression, the expression is evaluated to produce a syntax value that is used at that point in the lift expression value being constructed.

constructor: `Drop(exp)`

syntax:

```
Drop ::= ’<’ Exp ’>’
```

example:

```
[| let x = <y> in x + 1 end |]
```

interface:

```
@Class Drop extends OCL
  @Attribute exp : OCL end
  @Operation lift end
  @Operation pprint end
  @Operation toString end
end
```

20.19 Class `ForSeq`

context: `Root::OCL`

interface:

```
@Class ForSeq extends Object
end
```

20.20 Class `FunBinding`

context: `Root::OCL`

interface:

```
@Class FunBinding extends Binding
  @Attribute args : Seq(Parameter) end
  @Attribute body : Performable end
  @Attribute type : Classifier end
end
```

20.21 Class `For`

context: `Root::OCL`

interface:

```
@Class For extends Object
end
```

20.22 Class `HeadUpdate`

context: `Root::OCL`

interface:

```
@Class HeadUpdate extends OCL
  @Attribute seq : Performable end
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.23 Class `Iterate`

context: `Root::OCL`
overview:

An iteration expression is used to accumulate a value while processing each individual element from a collection in turn. Iteration expressions capture the common pattern that occurs where we wish to incrementally build a value up, for example adding all the elements of a set. An iteration expression is the basis for all expressions such as ’collect’, and ’forAll’. Iteration expressions are part of OCl and are slightly limited in the sense that there can only be one collection and one accumulation value. In general, XMF provides recursive operations to deal with more complex cases.

constructor: `Iterate(coll,name,acc,value,body)`

Constructs an iterate expression from a collection valued expression, two strings naming the element variable and the accumulation variable, an initial value expression and a body expression.

syntax:

```
       Iterate ::= Exp ’->’ ’iterate’ ’(’ Name Name ’=’ Exp ’|’ Exp ’)’.
```

example:

```
       Set{1,2,3,4}->iterate(x sum = 0 | sum + x)
```

interface:

```
@Class Iterate extends OCL
  @Attribute accumulator : String end
  @Attribute body : OCL end
  @Attribute collection : OCL end
  @Attribute name : String end
  @Attribute value : OCL end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.24 Class `If`

context: `Root::OCL`
overview:

A conditional expression evaluates a test expression and then performs one of two possible expressions depending on whether the test produces 'true' or 'false'.

constructor: `If(test,then,else)`

Constructs a conditional expression.

syntax:

```
If ::= 'if' Exp 'then' Exp [ IfTail ] 'end'.
IfTail ::= 'else' Exp | 'elseif' Exp 'then' Exp IfTail.
```

example:

```
if x > y then format(stdout,"x > y") else format(stdout,"y >= x") end
if x = y
then format(stdout,"~S = ~S~%",Seq{x,y})
elseif x > y
then format(stdout,"~S > ~S~%",Seq{x,y})
else format(stdout,"~S > ~S~%",Seq{y,x})
end
```

interface:

```
@Class If extends OCL
  @Attribute else : OCL end
  @Attribute then : OCL end
  @Attribute test : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation init end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.25 Class `ImportIn`

context: `Root::OCL`
overview:

A namespace can be imported for the scope of an expression. The names in the name space are available in the expression. Note that if the names are the same as lexically bound names currently in scope then the lexically bound names take precedence.

constructor: `ImportIn(nameSpace,body)`

The name space is an expressionwhose value is a name space and the body is an arbitrary expression.

syntax:

```
ImportIn ::= ’import’ Exp ’in’ Exp ’end’.
```

example:

```
import Compiler in compileFile(fileName,true,true) end
```

interface:

```
@Class ImportIn extends OCL
  @Attribute body : Performable end
  @Attribute nameSpace : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation eval end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.26 Class `IterExp`

context: `Root::OCL`
overview:

An iter-exp selects values from a collection and performs a task. Tasks include mappings a collection to another collection by applying a function to each element or filtering a collection by applying a predicate to each element in turn. Iteration expressions have a general form:

```
C->i(x | e)
```

where C is a collection, i is the name of the iteration expression, x is a variable bound to elements of C and e is an expression. The variable x is scoped over e. XMF knows about the following iteration names:

collect transforms a collection; each element is transformed to a new element produced by the corresponding evaluation of the body e. The resulting collection is of the same type as C.

select filters a collection; the body of the iter-exp is a boolean values expression that controls whether or not the value is added to the resulting collection. The resulting collection is of the same type as C.

reject is the opposite of select.

forAll returns true when the boolean valued expression e is true for all if the items in the collection. Otherwise returns false.

exists returns true when the boolean valued expreession e is true for at least one of the items in the collection. Otherwise returns false.

constructor: `IterExp(C,i,x,e)`

Returns a collection expression where C and e are expressions, i are x are strings.

syntax:

```
IterExp ::= Exp '->' Name '(' Name '|' Exp ')'.
```

example:

```
S->collect(x | x + 1)
S->select(x | x > 100)
S->reject(x | x <= 100)
S->forAll(x | x > 100)
S->exists(x | x > 100)
```

interface:

```
@Class IterExp extends OCL
  @Attribute body : OCL end
  @Attribute collection : OCL end
  @Attribute iterOp : String end
  @Attribute name : String end
  @Operation FV end
  @Operation compile end
  @Operation desugarSelect end
  @Operation desugar end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.27 Class `IntExp`

context: `Root::OCL`
overview:

An integer constant.

constructor: `IntExp(value)`

syntax:

```
Int
```

example:

```
100
```

interface:

```
@Class IntExp extends OCL
  @Attribute value : Integer end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.28 Class `Letrec`

context: `Root::OCL`

interface:

```
@Class Letrec extends OCL
  @Attribute body : Performable end
  @Attribute bindings : Seq(FunBinding) end
end
```

20.29 Class `Let`

context: `Root::OCL`
overview:

A ’let’ expression allows local variables to be defined. The scope of any ’let’ introduced variables is the body of the ’let’ expression. Once thay are introduced, the variable sare local to the body and may be referenced or updated. Let allows more than one ’binding’ before the body in which case the bindings are separated using ’;’. Note that the bindings are established in parallel (i.e. the values cannot refer to the names in other bindings).

constructor: `Let(bindings,body)`

Constructs a ’let’ expression. The bindings are a sequence of value bindings. The body is an expression.

syntax:

```
Let ::= ’let’ Bindings ’in’ Exp ’end’.
Bindings ::= Binding (’;’ Binding)*.
Binding ::= Name ’=’ Exp.
```

example:

```
let x = o.findX() in x.println() end
let x = 1; y = 2 in x + y end
```

interface:

```
@Class Let extends OCL
  @Attribute body : OCL end
  @Attribute bindings : Seq(ValueBinding) end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.30 Class `Lift`

context: `Root::OCL`

interface:

```
@Class Lift extends OCL
  @Attribute exp : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation init end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
end
```

20.31 Class `Negate`

context: `Root::OCL`

interface:

```
@Class Negate extends OCL
  @Attribute exp : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.32 Class `Order`

context: `Root::OCL`

interface:

```
@Class Order extends OCL
  @Attribute first : OCL end
  @Attribute second : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.33 Class `Operation`

context: `Root::OCL`
overview:

> An operation is a parameterized performable expression. An operation has a name, some parameters, a return type, a performable body and a documentation string. The class OCL::Operation defines the abstract syntax of an operation. When an instance of OCL::Operation is performed it is translated into an instance of EMOF::Operation (typically either a CompiledOperation or an InterpretedOperation). Typically you will not create an instance of OCL::Operation directly, rather you will use the concrete syntax @Operation ... end.

constructor: `Operation(name,parameters,type,body)`

The name should be a string. The name ’anonymous’ is used when we don’t care what the name of the operation is. The parameters are supplied as a sequence of instances of EMOF::Parameter. The type is n expression that will return a classifier when it is performed. The body is a performable expression.

syntax:

```
Operation ::= @Operation [ Name ] Args [ ’:’ Exp ] [ Str ] Exp ’end’.
Args ::= ’(’ [ Parameter (’,’ Parameter)* ] ’)’.
Parameter ::= Name ’:’ Exp.
```

example:

```
@Operation add1(x:Integer):Integer
  "Adds one to x"
  x + 1
end
```

interface:

```
@Class Operation extends OCL
  @Attribute documentation : String end
  @Attribute name : String end
  @Attribute performable : Performable end
  @Attribute parameters : Seq(Pattern) end
  @Attribute type : Performable end
  @Operation FV end
  @Operation arity end
  @Operation compile end
  @Operation eval end
  @Operation liftParameters end
  @Operation lift end
  @Operation matchCode end
  @Operation maxLocals end
  @Operation newParams end
  @Operation paramList end
  @Operation pprint end
  @Operation toString end
end
```

20.34 Class `Objectp`

context: `Root::OCL`
overview:

An object pattern consists of a class path and a sequence of patterns. The general form is C(p1,..,pn). A value matches an object pattern when it is an instance of C and when its slots match the patterns p1 to pn. In order for the slots to match, the class C must define a constructor of arity n. this constructor defines a sequence of attribute names which in turn define the mapping from the slots of the value to the patterns.

constructor: `Objectp(nameSpace,names,slots)`

The name space is a string, the names are a sequence of names and slots is a sequence of expressions.

syntax:

```
       Objectp ::= Name ('::' Name)* '(' Pattern* ')'.
```

interface:

```
@Class Objectp extends Pattern
  @Attribute class : String end
  @Attribute names : Seq(String) end
  @Attribute slots : Seq(Pattern) end
  @Operation bind end
  @Operation lift end
  @Operation matchSlots end
  @Operation matchCode end
  @Operation pprint end
end
```

20.35 Class `OCL`

context: `Root::OCL`
overview:

> The OCL class is the root of the OCL AST class hierarchy. Note that this class is not the root of the XMF class hierarchy for abstract syntax constructs (i.e. things that can be performed). That class is EMOF::Performable; OCL is a sub-class of Performable.

interface:

```
@Class OCL isabstract extends Performable
  @Operation FV end
  @Operation compile end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation pprint end
  @Operation pprint end
  @Operation typeExp end
end
```

20.36 Class `Pattern`

context: `Root::OCL`
overview:

> A pattern can occur in a binding position and is used to conditionally match values and bind variables to component values. A pattern is replaced with a variable, which it can be asked to generate and can be requested to translate itself into pattern matching code.

interface:

```
@Class Pattern isabstract extends Object
  @Operation lift end
  @Operation matchCode end
  @Operation newVar end
  @Operation newParam end
  @Operation pprint end
  @Operation pprint end
  @Operation typeExp end
end
```

20.36.1 Operation `newParam`

context: `Root::OCL::Pattern`
overview:

> By default a pattern will generate a new paremeter with a new var name. The parameter is used to contain the value that is matched against the pattern.

20.37 Class `PathUpdate`

context: `Root::OCL`

interface:

```
@Class PathUpdate extends OCL
  @Attribute path : Path end
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.38 Class `Path`

context: `Root::OCL`

interface:

```
@Class Path extends OCL
  @Attribute names : Seq(Element) end
  @Attribute root : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.39 Class `SlotUpdate`

context: `Root::OCL`

interface:

```
@Class SlotUpdate extends OCL
  @Attribute name : String end
  @Attribute target : OCL end
  @Attribute value : OCL end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.40 Class `Self`

context: `Root::OCL`

interface:

```
@Class Self extends OCL
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
end
```

20.41 Class `StrExp`

context: `Root::OCL`

interface:

```
@Class StrExp extends OCL
  @Attribute value : String end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.42 Class `SetExp`

context: `Root::OCL`

interface:

```
@Class SetExp extends OCL
  @Attribute collType : String end
  @Attribute elements : Seq(OCL) end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.43 Class `Send`

context: `Root::OCL`

interface:

```
@Class Send extends OCL
  @Attribute args : Seq(OCL) end
  @Attribute message : String end
  @Attribute target : OCL end
  @Operation FV end
  @Operation compileSend end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.44 Class `Try`

context: `Root::OCL`
overview:

A try expression is used to capture exceptional events that occur during evaluation. The body of the try expression is performed and, if no exceptions are thrown during its evaluation, the body produces the value of the try. Otherwise a value is thrown by the body in which case the value is bound to the catch variable and a catch handler is performed. The catch handler may attempt to resolve the reason for the exception being raised in which case it evaluates normally and produces tghe value of the try expression. Otherwise the catch handler throws an exception value which is passed to the next most recently performed try.

syntax:

```
       Try ::= ’try’ Exp ’catch’ ’(’ Name ’)’ Exp ’end’.
```

interface:

```
@Class Try extends OCL
  @Attribute body : Performable end
  @Attribute handler : Performable end
  @Attribute name : String end
  @Operation FV end
  @Operation compile end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.45 Class `Throw`

context: `Root::OCL`
overview:

A throw expression throws a value to the most recently performed try expression. The value is bound to the catch-variable in the try and the body of the catch is performed.

syntax:

```
       Throw ::= ’throw’ Exp.
```

interface:

```
@Class Throw extends OCL
  @Attribute exp : Performable end
  @Operation FV end
  @Operation compile end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.46 Class `TailUpdate`

context: `Root::OCL`

interface:

```
@Class TailUpdate extends OCL
  @Attribute seq : Performable end
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.47 Class `ValueBinding`

context: `Root::OCL`

interface:

```
@Class ValueBinding extends Binding
  @Attribute value : OCL end
  @Operation lift end
  @Operation pprint end
  @Operation toString end
end
```

20.48 Class `VarUpdate`

context: `Root::OCL`

interface:

```
@Class VarUpdate extends OCL
  @Attribute name : String end
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.49 Class `Varp`

context: `Root::OCL`
overview:

A variable pattern is just a name and an optional type.

constructor: `Varp(name,type)`

The name is a string and the type is an expression producing a classifier.

syntax:

```
       Varp ::= Name [ ':' Exp ].
```

interface:

```
@Class Varp extends Pattern
  @Attribute name : String end
  @Attribute type : Performable end
  @Operation bind end
  @Operation lift end
  @Operation matchCode end
  @Operation newVar end
  @Operation newParam end
  @Operation pprint end
end
```

20.50 Class `Var`

context: `Root::OCL`

interface:

```
@Class Var extends OCL
  @Attribute charCount : Integer end
  @Attribute lineCount : Integer end
  @Attribute name : String end
  @Attribute sourceSet : Boolean end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

20.51 Operation `update`

context: `Root::OCL`

21 Package Parser

context: `Root`

22 Package BNF

context: Root::Parser

22.1 Class At

context: Root::Parser::BNF

interface:

```
@Class At extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.2 Class And

context: Root::Parser::BNF

interface:

```
@Class And extends Recognizer
  @Attribute left : Recognizer end
  @Attribute right : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.3 Class Action

context: Root::Parser::BNF

interface:

```
@Class Action extends Recognizer
  @Attribute action : Element end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
end
```

22.4 Class `BindGlobal`

context: `Root::Parser::BNF`

interface:

```
@Class BindGlobal extends Recognizer
  @Attribute name : String end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.5 Class `Bind`

context: `Root::Parser::BNF`

interface:

```
@Class Bind extends Recognizer
  @Attribute name : String end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.6 Class `Call`

context: `Root::Parser::BNF`

interface:

```
@Class Call extends Recognizer
  @Attribute clause : Clause end
  @Attribute name : String end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.7 Class `Char`

context: `Root::Parser::BNF`

interface:

```
@Class Char extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.8 Class `Cut`

context: `Root::Parser::BNF`

interface:

```
@Class Cut extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.9 Class `Clause`

context: `Root::Parser::BNF`

interface:

```
@Class Clause extends NamedElement
  @Attribute body : Recognizer end
  @Operation code end
  @Operation call end
  @Operation init end
  @Operation lift end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation terminals end
end
```

22.10 Class `Const`

context: `Root::Parser::BNF`

interface:

```
@Class Const extends Recognizer
  @Attribute value : Element end
  @Operation FV end
  @Operation init end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.11 Class `EndAt`

context: `Root::Parser::BNF`

interface:

```
@Class EndAt extends Recognizer
  @Attribute grammar : Element end
  @Attribute owner : Classifier end
  @Operation init end
  @Operation parse end
end
```

22.12 Class `EOF`

context: `Root::Parser::BNF`

interface:

```
@Class EOF extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.13 Class `Fail`

context: `Root::Parser::BNF`

interface:

```
@Class Fail extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.14 Class `Grammar`

context: `Root::Parser::BNF`
overview:

> A grammar is an owned element that describes how to recognise a sequence of characters and to synthesize values. grammars are extensible elements where clauses are inherited from the parents. Clauses with the same name are merged using ’or’. A grammar is performable because we want to define grammars as part of performable definitions. Therefore a grammar can be compiled into an expression that will create the grammar when it is performed.

constructor: `Grammar(parents,clauses,startingSymbol,imports)`

interface:

```
@Class Grammar extends NameSpace,Performable
  @Attribute clauses : Set(Clause) end
  @Attribute dynamics : Seq(Element) end
  @Attribute debug : Boolean end
  @Attribute owner : Element end
  @Attribute parents : Set(Grammar) end
  @Attribute startingSymbol : String end
  @Attribute terminals : Set(String) end
  @Operation FV end
  @Operation allClauses end
  @Operation addClause end
  @Operation add end
  @Operation compile end
  @Operation call end
  @Operation cacheTerminals end
  @Operation getClause end
  @Operation init end
  @Operation inheritedTerminals end
  @Operation inheritedClauses end
  @Operation localTerminals end
  @Operation localClauses end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation pprint end
  @Operation transform end
  @Operation terminals end
end
```

22.14.1 Operation `call`

context: `Root::Parser::BNF::Grammar`
overview:

> Call the nonterminal with respect to the current machine state. The grammar will have a single clause wih the given name or 'null' if no clause exists. Calling the clause is like calling a procedure. We get a new stack frame, we push a debugging fail choice that prints out diagnostics if we ever fail past this point, we push a success continuation that prints out diagnostics if we ever succeed in completing the clause, we record the current fail state in case we ever invoke '!'. Finally we invoke the parser for the clause.

22.15 Class `ImportAt`

context: `Root::Parser::BNF`

interface:

```
@Class ImportAt extends Recognizer
  @Operation code end
  @Operation parse end
  @Operation toExp end
  @Operation transform end
end
```

22.16 Class `Int`

context: `Root::Parser::BNF`

interface:

```
@Class Int extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.17 Class `Loop`

context: `Root::Parser::BNF`

interface:

```
@Class Loop extends Recognizer
  @Attribute parser : Recognizer end
  @Attribute values : Seq(Element) end
  @Operation init end
  @Operation parse end
  @Operation setOwner end
end
```

22.18 Class `NewStack`

context: `Root::Parser::BNF`

interface:

```
@Class NewStack extends Recognizer
  @Attribute locals : Element end
  @Attribute stack : Seq(Element) end
  @Operation init end
  @Operation parse end
end
```

22.19 Class `Name`

context: `Root::Parser::BNF`

interface:

```
@Class Name extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation toExp end
  @Operation transform end
end
```

22.20 Class `Opt`

context: `Root::Parser::BNF`

interface:

```
@Class Opt extends Recognizer
  @Attribute parser : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
end
```

22.21 Class `Or`

context: `Root::Parser::BNF`

interface:

```
@Class Or extends Recognizer
  @Attribute left : Recognizer end
  @Attribute right : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.22 Class `Plus`

context: `Root::Parser::BNF`

interface:

```
@Class Plus extends Recognizer
  @Attribute parser : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.23 Class `PreAction`

context: `Root::Parser::BNF`

interface:

```
@Class PreAction extends Recognizer
  @Attribute action : Element end
  @Operation FV end
  @Operation body end
  @Operation code end
  @Operation init end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.24 Class `PlusCons`

context: `Root::Parser::BNF`

interface:

```
@Class PlusCons extends Recognizer
  @Attribute parser : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.25 Class `Recognizer`

context: `Root::Parser::BNF`

interface:

```
@Class Recognizer isabstract extends Object
  @Operation FV end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.26 Class `Star`

context: `Root::Parser::BNF`

interface:

```
@Class Star extends Recognizer
  @Attribute parser : Recognizer end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
end
```

22.27 Class `StarCons`

context: `Root::Parser::BNF`

interface:

```
@Class StarCons extends Recognizer
  @Attribute parser : Recognizer end
  @Attribute values : Seq(Element) end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation setOwner end
  @Operation transform end
  @Operation toExp end
end
```

22.28 Class `Str`

context: `Root::Parser::BNF`

interface:

```
@Class Str extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

22.29 Class `Terminal`

context: `Root::Parser::BNF`

interface:

```
@Class Terminal extends Recognizer
  @Attribute terminal : String end
  @Operation FV end
  @Operation code end
  @Operation init end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
  @Operation terminals end
  @Operation toString end
end
```

22.30 Class `Term`

context: `Root::Parser::BNF`

interface:

```
@Class Term extends Str
  @Operation parse end
  @Operation toExp end
end
```

22.31 Class `TraceSuccess`

context: `Root::Parser::BNF`

interface:

```
@Class TraceSuccess extends Recognizer
  @Attribute name : String end
  @Operation init end
  @Operation parse end
end
```

22.32 Class `TypeCheck`

context: `Root::Parser::BNF`

interface:

```
@Class TypeCheck extends Recognizer
  @Attribute type : Seq(String) end
  @Operation FV end
  @Operation code end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
  @Operation toString end
end
```

22.33 Class `Token`

context: `Root::Parser::BNF`

interface:

```
@Class Token extends Object
  @Attribute charCount : Integer end
  @Attribute charPos : Integer end
  @Attribute data : Element end
  @Attribute lineCount : Integer end
  @Attribute type : Integer end
  @Operation toString end
end
```

22.34 Class `Tok`

context: `Root::Parser::BNF`

interface:

```
@Class Tok extends Recognizer
  @Operation FV end
  @Operation code end
  @Operation nextSet end
  @Operation parse end
  @Operation transform end
  @Operation toExp end
end
```

23 Package `Machine`

context: `Root::Parser`
overview:

> This package defines the elements necessary to run the parsing machine. Create an instance of the class State and then invoke 'run/0' or 'run/1' to parse the chars from an input channel.

23.1 Class `BasicState`

context: `Root::Parser::Machine`
overview:

> A basic state defines the essential information contained in a machine state. It is abstract.

constructor: `BasicState(grammar,inputChannel)`

> Creates an initial machine state that uses grammar to parse the characters from the input stream.

interface:

```
@Class BasicState isabstract extends Object
  @Attribute consumedChars : Integer end
  @Attribute cut : Seq(Fail) end
  @Attribute fail : Fail end
  @Attribute globals : Env end
  @Attribute grammar : Grammar end
  @Attribute imports : Seq(NameSpace) end
  @Attribute indent : Integer end
  @Attribute inputChannel : InputChannel end
  @Attribute locals : Env end
  @Attribute owner : Element end
  @Attribute stack : Seq(Element) end
  @Attribute successes : Seq(Recognizer) end
  @Attribute token : Seq(Element) end
  @Attribute tokenChannel : Element end
  @Attribute tokens : Element end
  @Operation atNameSpace end
  @Operation atClassifier end
  @Operation at end
  @Operation init end
  @Operation importAt end
  @Operation resetTokens end
  @Operation resetInputChannel end
  @Operation tokenStream end
  @Operation tokens end
end
```

23.2 Class `Fail`

context: `Root::Parser::Machine`

interface:

```
@Class Fail isabstract extends Object
  @Operation fail end
end
```

23.3 Class `InitialFail`

context: `Root::Parser::Machine`

interface:

```
@Class InitialFail extends Fail
  @Operation fail end
  @Operation init end
end
```

23.4 Class `MachineFail`

context: `Root::Parser::Machine`

interface:

```
@Class MachineFail extends Fail,BasicState
  @Operation fail end
  @Operation init end
end
```

23.5 Class `State`

context: `Root::Parser::Machine`
overview:

> A machine state contains all information to execute a parse. The parse proceeds by making machine state transitions.

interface:

```
@Class State extends BasicState
  @Attribute failed : Boolean end
  @Attribute parses : Integer end
  @Operation bindsGlobal end
  @Operation bindGlobal end
  @Operation binds end
  @Operation bind end
  @Operation cut end
  @Operation choice end
  @Operation choice end
  @Operation call end
  @Operation decIndent end
  @Operation debugs end
  @Operation debug end
  @Operation debug end
  @Operation eof end
  @Operation env end
  @Operation fail end
  @Operation getClause end
  @Operation isTerminal end
  @Operation incIndent end
  @Operation lookupGlobal end
  @Operation lookup end
  @Operation nextSet end
  @Operation next end
  @Operation newStack end
  @Operation pushTraceSuccess end
  @Operation pushTraceFail end
  @Operation pushSuccess end
  @Operation pushStack end
  @Operation pushEndAt end
  @Operation popCut end
  @Operation pushCut end
  @Operation printConsumedChars end
  @Operation popSuccess end
  @Operation popStack end
  @Operation parseError end
  @Operation runToCompletion end
  @Operation run end
  @Operation run end
  @Operation reset end
  @Operation startAt end
  @Operation stackTop end
  @Operation terminal end
  @Operation typeCheck end
end
```

23.6 Class `TraceFail`

context: `Root::Parser::Machine`

interface:

```
@Class TraceFail extends Fail
  @Attribute fail : Fail end
  @Attribute name : String end
  @Operation fail end
  @Operation init end
end
```

24 Package `TopLevel`

context: `Root`
overview:

> The top level package defines data and operations that control the top level loop for XMF.

24.1 Class `Loop`

context: `Root::TopLevel`
overview:

> Create an instance of Loop and supply an input channel. This returns a command interpreter for the input channel.

constructor: `Loop(inputChannel)`

> Creates andreturns a new command interpreter.

interface:

```
@Class Loop extends Object
  @Attribute channel : InputChannel end
  @Operation handleException end
  @Operation init end
  @Operation loop end
  @Operation printBanner end
end
```

24.2 Class `Version`

context: `Root`
overview:

> This class defines operations that manage the version information maintained in the file Boot/verison.xml.

interface:

```
@Class Version extends Object
  @Attribute XMF : Element end
  @Attribute delta : Element end
  @Attribute document : Document end
  @Attribute new : Boolean end
  @Attribute release : Element end
  @Attribute silent : Boolean end
  @Attribute version : Element end
  @Operation dialog end
  @Operation deltaId end
  @Operation incDelta end
  @Operation loadVersion end
  @Operation newVersion end
  @Operation newRelease end
  @Operation newVersionDialog end
  @Operation newReleaseDialog end
  @Operation newDeltaDialog end
  @Operation newDelta end
  @Operation releaseId end
  @Operation setXMF end
  @Operation setVersion end
  @Operation setRelease end
  @Operation setDelta end
  @Operation versionId end
  @Operation version end
  @Operation writeVersion end
end
```

25 Package Walkers

context: Root
overview:

> The Walkers package defines the abstract class Walker and a number of useful instances and extensions. An element walker is used to walk an arbitrary value and perform a task. The walker will handle cycles in the structure of the element. For example, a walker may be used to search a value, or collect information about its contents.

25.1 Class TypeCheckReport

context: Root::Walkers

interface:

```
@Class TypeCheckReport extends Object
  @Attribute name : String end
  @Attribute ok : Boolean end
  @Attribute object : Object end
  @Attribute type : Classifier end
  @Attribute value : Element end
  @Operation toString end
end
```

25.2 Class TypeCheckError

context: Root::Walkers

interface:

```
@Class TypeCheckError extends Exception
  @Operation toString end
end
```

25.3 Class TypeChecker

context: Root::Walkers

interface:

```
@Class TypeChecker extends Walker
  @Attribute reports : Set(TypeCheckReport) end
  @Operation addToReports end
  @Operation defaultWalk end
  @Operation reWalk end
  @Operation walkTable end
  @Operation walkString end
  @Operation walkSlot end
  @Operation walkSet end
  @Operation walkSeq end
  @Operation walkPostObject end
  @Operation walkPreObject end
  @Operation walkOperation end
  @Operation walkNull end
  @Operation walkInteger end
  @Operation walkBoolean end
end
```

25.4 Class `Walker`

context: `Root::Walkers`
overview:

> An element walker is used to perform some arbitrary task over the structure of a value. A walker recursively descends into an element's structure and dispatches to appropriate operations depending on the values of component elements. A walker handles cycles in a value by dispatching to a special operation after an element has been encountered for the first time. To create your own walker you should specialise this class or one of its descendants and implement the abstract operations or override the provided operations.

interface:

```
@Class Walker isabstract extends Object
  @Attribute refCount : Integer end
  @Attribute table : Table end
  @Operation defaultWalk end
  @Operation encountered end
  @Operation encounter end
  @Operation encounter end
  @Operation getRef end
  @Operation newRef end
  @Operation reWalk end
  @Operation reset end
  @Operation walkTable end
  @Operation walkString end
  @Operation walkSlot end
  @Operation walkSet end
  @Operation walkSeq end
  @Operation walkOperation end
  @Operation walkPostObject end
  @Operation walkPreObject end
  @Operation walkNull end
  @Operation walkInteger end
  @Operation walkBoolean end
  @Operation walkObject end
  @Operation walk end
end
```

25.5 Class `XML`

context: `Root::Walkers`

interface:

```
@Class XML extends Walker
  @Operation defaultWalk end
  @Operation reWalk end
  @Operation walkTable end
  @Operation walkString end
  @Operation walkSlot end
  @Operation walkSet end
  @Operation walkSeq end
  @Operation walkRef end
  @Operation walkPostObject end
  @Operation walkPreObject end
  @Operation walkOperation end
  @Operation walkNull end
  @Operation walkInteger end
  @Operation walkBoolean end
end
```

26 Package `XOCL`

context: `Root`
overview:

> The XOCL package contains definitions that extend the basic features of the OCL package. Many of the classes in this package are abstract syntax structure types that are instantiated by parsers. The syntax structures define how to perform themselves, for example by turning themselves into machine instructions.

26.1 Class `AbstractOp`

context: `Root::XOCL`
overview:

> An abstract operation defines the signature of an operation without specifying the body. If it is ever called, an abstract operation will signal an error. The intention is that an abstract operation is defined in an abstract class and that it is redefined in each concrete sub-class.

constructor: `AbstractOp(name,args,type)`

> The name is a string, the args is a sequence of patterns, the type is an expression producing a classifier.

syntax:

```
AbstractOp ::= ’@AbstractOp’ Name ’(’ Pattern* ’)’ [ ’:’ Exp ]
```

interface:

```
@Class AbstractOp extends Sugar
  @Attribute args : Seq(Element) end
  @Attribute name : String end
  @Attribute type : Performable end
  @Operation desugar end
end
```

26.2 Class `AttributeModifiers`

context: `Root::XOCL`

interface:

```
@Class AttributeModifiers extends Object
  @Attribute accessor : Boolean end
  @Attribute extender : Boolean end
  @Attribute reducer : Boolean end
  @Attribute updater : Boolean end
  @Operation defineModifier end
  @Operation emptyModifier end
  @Operation pprint end
end
```

26.3 Class `Attribute`

context: `Root::XOCL`
overview:

> This class represents an attribute definition. Performing an attribute definition produces an attribute.

constructor: `Attribute(name,type,modifiers)`

> The name should be a symbol and the type should be an expression that evaluates to produce a classifier. The modfiers control the automatic generation of the appropriate operations in a class. The modifiers are: ? generating an accessor with the same name as the attribute, ! generating an updater with the name setNAME, + generating an extender with the name addNAME and - generating a reducer with the name deleteNAME.

syntax:

```
    Attribute ::= '@Attribute' Name ':' Exp [ '(' Modifier (',' Modifier)* ')' ] 'end'.
    Modifier ::= ? | ! | + | -.
```

interface:

```
@Class Attribute extends Def
  @Attribute modifiers : AttributeModifiers end
  @Attribute type : Performable end
  @Operation accessor end
  @Operation compile end
  @Operation extender end
  @Operation eval end
  @Operation lift end
  @Operation operations end
  @Operation pprint end
  @Operation reducer end
  @Operation toString end
  @Operation updater end
end
```

26.4 Class `Bind`

context: `Root::XOCL`
overview:

> A value binding is an association between a name ans a value. A value binding can be added to a name space. Once added, the value can be referenced using its name via the name space using the '::' notation.

constructor: `Bind(name,value`

>) The name should be a string or a symbol and the value is an expression. The name will be coerced to be a symbol.

interface:

```
@Class Bind extends Def
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation init end
  @Operation maxLocals end
  @Operation pprint end
end
```

26.5 Class `Case`

context: `Root::XOCL`
overview:

A case expression captures the often used pattern of dispatching on a value to do several different tasks.

constructor: `Case(value,arms,default)`

The value is an expression. Default may be either an expression or null if not specified. The arms are a sequence of case arms.

syntax:

```
Case ::= '@Case' Exp 'of' Arm* [ Default ] 'end'.
Arm ::= Exp 'do' Exp ';'.
Default ::= 'else' Exp.
```

example:

```
@Case x of
  1 do one();
  2 do two();
  "three" do three();
  true do four();
  else do error()
end
```

interface:

```
@Class Case extends Performable
  @Attribute arms : Seq(CaseArm) end
  @Attribute default : Performable end
  @Attribute value : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation init end
  @Operation maxLocals end
end
```

26.6 Class `CaseArm`

context: `Root::XOCL`

interface:

```
@Class CaseArm extends Object
  @Attribute action : Performable end
  @Attribute value : Performable end
  @Operation init end
end
```

26.7 Class `Constraint`

context: `Root::XOCL`

interface:

```
@Class Constraint extends Sugar
  @Attribute body : Performable end
  @Attribute name : String end
  @Attribute reason : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

26.8 Class `Constructor`

context: `Root::XOCL`
overview:

A constructor describes how to initialise a newly created instance of a class. Each class may define a number of constructors that are used when an instance of the class is sent an ’init/1’ message. The constructor is a sequence of attribute names and an optional body. When an instance is initialised, the matching constructor is selected from the class. A constructor matches the initialisation arguments when it defines the same number of attribute names as the length of the initialization arguments. The effect of using the selected constructor is to set the slots with the appropriate names in the constructor from the initialization arguments and then to perform the body of the constructor. If defined, the body of the constructor must return ’self’. A constructor may optionally specify that it is a template for transforming an object into a string, it does so by including the optional modifier ’!’ after the sequence of names. If present, the body of the constructor is evaluated in a scope where the names of the object slots specified in the name list are bound to the supplied values.

constructor: `Constructor(names,body)`

The names are a sequence of strings and the body is an expression.

syntax:

```
       Constructor ::= ’@Constructor’ ’(’ [ Name (’,’ Name)* ] ’)’ [ ’!’ ] Exp ’end’.
```

interface:

```
@Class Constructor extends Sugar
  @Attribute body : Performable end
  @Attribute names : Seq(String) end
  @Attribute toString : Boolean end
  @Operation desugar end
  @Operation namesToString end
  @Operation operations end
  @Operation pprint end
  @Operation toStringOperation end
end
```

26.9 Class `Class`

context: `Root::XOCL`
overview:

The class XOCL::Class is a class definition. When a class definition is performed, it produces a class. Classes (typically) live in name spaces and should be added to a name space. Either define a class using a ’context ...’ construct or add a class directly to a name space and then initialise the class via its ’init/1’ operation.

constructor: `Class(name,isabstract,parents,doc,defs)`

The name should be a string. A class is either abstract or not depending on the value of the boolean. The parents in a class definition are expressions. The documentation for a class is a string. The definitions contained in a class are a sequence of expressions.

syntax:

```
Class ::= ’@Class’ Name [ isabstract ] [ extends Exp (, Exp)* Exp* ’end’
```

example:

```
@Class Record
  @Attribute field1 : String end
  @Attribute field2 : Integer end
  @Constructor(field1,field2) end
  @Operation add(i:Integer)
    self.field2 := field2 + i
  end
end
```

interface:

```
@Class Class extends Def
  @Attribute defs : Seq(Performable) end
  @Attribute doc : String end
  @Attribute isAbstract : Boolean end
  @Attribute name : Symbol end
  @Attribute parents : Seq(Performable) end
  @Operation attributes end
  @Operation constructors end
  @Operation compileClassPopulation end
  @Operation compileClassCreation end
  @Operation compile end
  @Operation foldDefs end
  @Operation init end
  @Operation maxLocals end
  @Operation pprint end
end
```

26.10 Class `Def`

context: `Root::XOCL`
overview:

A definition is a named performable syntactic element. The class Def is abstract and requires that all sub-classes have a name.

interface:

```
@Class Def isabstract extends Performable
  @Attribute name : String end
end
```

26.11 Class `Fork`

context: `Root::XOCL`
overview:

The fork construct creates a new execution thread. The body of the Fork construct is the expression to be evaluated by the thread when it starts up. A forked thread does not start immediately. At any given time there is a current thread that can choose to YIELD control. At this point in time, if there are any other threads waiting to run then one is (re)started. The order in which threads are activated via yield is in the order of thread creation.

constructor: `Fork(body)`

syntax:

```
Fork ::= '@Fork' Exp 'end'.
```

interface:

```
@Class Fork extends Sugar
  @Attribute body : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

26.12 Class `For`

context: `Root::XOCL`
overview:

A for loop selects elements from a sequence in turn and performs an action.

constructor: `For(name,coll,body)`

The name is a string and names each successive element in the collection. Coll is a sequence values expression and body is an arbitrary expression that will be evaluated for its side effect.

syntax:

```
For ::= '@For' Name 'in' Exp 'do' Exp 'end'
```

example:

```
@For x in 0.to(100)
  x.println()
end
```

interface:

```
@Class For extends Sugar
  @Attribute body : Performable end
  @Attribute coll : Performable end
  @Attribute name : String end
  @Operation desugar end
  @Operation lift end
  @Operation pprint end
  @Operation toString end
end
```

26.13 Class `Find`

context: `Root::XOCL`
overview:

> A find construct is used to perform an action in terms of an element of a collection. Typically we want to find the first element in a collection that satisfies a given predicate and to perform an action. If no value exists that satisfies the predicate then we optionally want to perform some other action. This construct captures the pattern.

constructor: `Find(collection,name,test,action,alternative)`

syntax:

```
       Find ::= '@Find' (Exp,Name) 'when' Exp 'do' Exp [ 'else' Exp ] 'end'.
```

interface:

```
@Class Find extends Sugar
  @Attribute alternative : Performable end
  @Attribute action : Performable end
  @Attribute collection : Performable end
  @Attribute name : String end
  @Attribute test : Performable end
  @Operation desugar end
  @Operation pprint end
  @Operation toString end
end
```

26.14 Class `Letrec`

context: `Root::XOCL`
overview:

> A letrec construct allows bindings to be established for a local scope. The bindings are mutually recursive. Since XMF has eager evaluation, this means that the values of the bindings should be operations.

constructor: `Letrec(bindings,body)`

syntax:

```
       Letrec ::= '@Letrec' Binding (';' Binding)* 'in' Exp 'end'.
       Binding ::= Name '=' Exp
```

interface:

```
@Class Letrec extends Sugar
  @Attribute body : Performable end
  @Attribute bindings : Seq(ValueBinding) end
  @Operation desugar end
  @Operation pprint end
end
```

26.15 Class `NameSpace`

context: `Root::XOCL`
overview:

> A name space is a named container of named elements. This construct allows a name space to be defined statically together with some bindings. Once created, new named elements can be added or existing ones removed.

constructor: `NameSpace(bindings)`

> Each binding is an instance of OCL::ValueBinding.

syntax:

```
       NameSpace ::= '@NameSpace' Name Binding* 'end'
```

interface:

```
@Class NameSpace extends Sugar
  @Attribute bindings : Seq(ValueBinding) end
  @Attribute name : String end
  @Operation desugar end
  @Operation pprint end
end
```

26.16 Class `Package`

context: `Root::XOCL`
overview:

> A package definition is represented as an instance of this class. When the instance is performed it creates a new package.

constructor: `Package(name,doc,defs)`

> The name is a string and the definitions are expressions. The definitions are mutually recursive if they have been defined statically in the same package.

syntax:

```
       Package ::= '@Package' Name Exp* end
```

interface:

```
@Class Package extends Def
  @Attribute defs : Seq(Performable) end
  @Attribute doc : String end
  @Attribute name : String end
  @Operation compile end
  @Operation init end
end
```

26.17 Class `Sugar`

context: `Root::XOCL`
overview:

> A sugared expression is one that is new syntax masquerading as old syntax. A sugared expression defines an operation 'desugar/0' that translates the receiver into an expression at a higher level of detail. It is not necessary to define how a sugared expression is compiled or interpreted since these mechanisms simply call desugar and then try again.

interface:

```
@Class Sugar isabstract extends Performable
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation typeExp end
end
```

26.18 Class `TypeCase`

context: `Root::XOCL`
overview:

> A typecase defines a dispatch table for an element based on its type. The type case value expression is evaluated and each type case arm is tried in turn. A typecase arm consists of a type and a body. If the value is of the specified type then the arm body is performed and produces the value of the typecase. If no arm matches then there may be an optional default expression which is evaluated and produces the value of the typecase. If no arm matches and no default is specified then an error is signaled.

constructor: `TypeCase(value,arms,default)`

> The value is an expression, the arms is a sequence of type case arms. The default is an expression.

syntax:

```
       TypeCase ::= '@TypeCase' '(' Exp ')' Arm* [ 'else' Exp ] 'end'.
       Arm ::= Exp 'do' Exp 'end'
```

interface:

```
@Class TypeCase extends Sugar
  @Attribute arms : Seq(TypeCaseArm) end
  @Attribute default : Performable end
  @Attribute value : Performable end
  @Operation desugarArms end
  @Operation desugar end
  @Operation pprint end
end
```

26.19 Class `TypeCaseArm`

context: `Root::XOCL`

interface:

```
@Class TypeCaseArm extends Object
  @Attribute body : Performable end
  @Attribute type : Performable end
  @Operation desugar end
  @Operation pprint end
end
```

26.20 Class `While`

context: `Root::XOCL`
overview:

A basic loop with a test. The test is performed at least once before the loop is entered each time. When the loop test returnes false, the loop terminates. The return value of a while expression is undefined.

constructor: `While(test,body)`

Both args should be expressions. The test should return a boolean value.

syntax:

```
       While ::= '@While' Exp 'do' Exp 'end'
```

interface:

```
@Class While extends Performable
  @Attribute body : Performable end
  @Attribute test : Performable end
  @Operation FV end
  @Operation compile end
  @Operation eval end
  @Operation lift end
  @Operation maxLocals end
  @Operation pprint end
  @Operation toString end
end
```

26.21 Class `WithoutDaemons`

context: `Root::XOCL`
overview:

The without daemons expression is used to turn daemons off for the evaluation of an expression in the context of an object. Often a daemon will fire and the code that handles the daemon should not cause recursive daemon invocation. This form can be used to safely turn off daemons temporarily for the scope of the handler.

constructor: `WithoutDaemons(object,body)`

The obejct and body should be expressions. Daemons are switched off for the scope of the evaluation of the body.

syntax:

```
       WithoutDaemons ::= ’@WithoutDaemons’ ’(’ Exp ’)’ Exp ’end’.
```

interface:

```
@Class WithoutDaemons extends Performable
  @Attribute body : Performable end
  @Attribute object : Performable end
  @Operation FV end
  @Operation compile end
  @Operation desugar end
  @Operation init end
  @Operation maxLocals end
  @Operation pprint end
end
```

27 Package XMap

context: Root
overview:

> This package defines various constructs that allow XMF objects to be mapped and synchronised.

27.1 Class AddRight

context: Root::XMap
overview:

> See AddLeft. this is identical except that the daemon fires when an element is added to the right hand container.

interface:

```
@Class AddRight extends AddAction
end
```

27.2 Class AddLeft

context: Root::XMap
overview:

> The AddLeft construct occurs in a SyncContainers construct and is used to define a daemon that fires when an element is added to the left container of the two being synchronized. If the left handl container is updated with one or more values then this daemon is fired for each new element. The construct has a parameter which is bound to the value which is added. In the body of the daemon the value of 'self' is the object being updated (i.e. the container). All other scoping rules apply.

constructor: `AddLeft(arg,body)`

syntax:

```
AddLeft ::= '@AddLeft' '(' Name ')' Exp 'end'.
```

example:

```
@SyncContainers(Class,constraints,myObj,sneaky)
  @AddLeft(constraint)
    // Every time anyone adds a constraint to Class
    // squirrel it away in MyClass.
    if not sneaky->includes(constraint)
    then myObj.sneaky := sneaky->including(constraint)
    end
  end
end
```

interface:

```
@Class AddLeft extends AddAction
end
```

(c) 2003 Xactium Ltd.

27.3 Class `AddAction`

context: `Root::XMap`
overview:

> An add action is a general action that can occur to a container. The AddAction construct is general and is specialized to particular forms of container synchronization and mapping.

interface:

```
@Class AddAction extends Sugar,SyncContainerAction
  @Attribute arg : String end
  @Attribute body : Performable end
  @Attribute name : String end
  @Operation desugar end
  @Operation setName end
end
```

27.4 Class `DeleteRight`

context: `Root::XMap`
overview:

> The DeleteRight construct occurs in a SyncContainers construct and is used to define a daemon that fires when an element is removed from the right container of the two being synchronized. If the right hand container is updated by deleting one or more values then this daemon is fired for each new element. The construct has a parameter which is bound to the value which is deleted. In the body of the daemon the value of ’self’ is the object being updated (i.e. the container). All other scoping rules apply.

constructor: `DeleteRight(arg,body)`

syntax:

```
       DeleteRight ::= ’@DeleteRight’ ’(’ Name ’)’ Exp ’end’.
```

example:

```
       @SyncContainers(Class,constraints,myObj,sneaky)
         @DeleteRight(constraint)
           // Every time anyone removes a constraint from myObj
           // remove the constraint from Class.
           Class.removeConstraint(constraint)
         end
       end
```

interface:

```
@Class DeleteRight extends DeleteAction
end
```

27.5 Class `DeleteLeft`

context: `Root::XMap`
overview:

> See DeleteRight. This is the same except the daemon fires when elements are removed from the left hand container.

interface:

```
@Class DeleteLeft extends DeleteAction
end
```

27.6 Class `DeleteAction`

context: `Root::XMap`
overview:

> A delete action is a general action that can occur to a container. The DeleteAction construct is general and is specialized to particular forms of container synchronization and mapping.

interface:

```
@Class DeleteAction extends Sugar,SyncContainerAction
  @Attribute arg : String end
  @Attribute body : Performable end
  @Attribute name : String end
  @Operation desugar end
  @Operation setName end
end
```

27.7 Class `SlotValueChanged`

context: `Root::XMap`
overview:

> This construct adds a daemon that fires when the given slot value changes. The construct allows the new value and old value to be bound. The body of the construct is any expression.

constructor: `SlotValueChanged(object,slot,newValue,oldValue,body`

> The object and body should be expressions. The slot is any expression whose value is a string or symbol. The new value and old value are both strings that name the variables bound for the scope of body.

syntax:

```
        SlotValueChanged ::= ’@SlotValueChanged’ ’(’ Exp ’,’ Exp ’,’ Name ’,’ Name ’)’ Exp ’end’.
```

interface:

```
@Class SlotValueChanged extends Sugar
  @Attribute body : Performable end
  @Attribute newValue : String end
  @Attribute oldValue : String end
  @Attribute object : Performable end
  @Attribute slot : Performable end
  @Operation desugar end
end
```

27.8 Class `SyncContainerAction`

context: `Root::XMap`

interface:

```
@Class SyncContainerAction isabstract extends Object
end
```

27.9 Class `Sync`

context: `Root::XMap`
overview:

This construct synchronizes two objects on respective slots. Once synchronized, the values of the two slots are the same. An update to the appropriate slot of either object will cause the change to be propagated to the other object.

constructor: `Sync(object1,name1,object2,name2)`

The objects are expressions and the names are strings.

syntax:

```
       Sync ::= @Sync' '('Exp '.' Name ',' Exp '.' Name ')' 'end'.
```

interface:

```
@Class Sync extends Sugar
  @Attribute name2 : String end
  @Attribute name1 : String end
  @Attribute object2 : Performable end
  @Attribute object1 : Performable end
  @Operation desugar end
  @Operation toString end
end
```

27.10 Class `SyncContainers`

context: `Root::XMap`
overview:

Two containers are synchronized using this construct. A container is an object with a slot whose value is a collection. Containers are synchronized with respect to a particular slot. A container may be synchronized (for containership) more than once; each time the slot may be the same or may be different. The body of this construct is a collection of daemon definitions. Special patterns of daemons for containership synchronization are provided: AddLeft; DeleteLeft; AddRight; DeleteRight. In each case daemons are defined that fire when elements are added or deleted to the left or right container with respect to the defined slot. The daemon bodies can be any expression and scoping rules apply except that 'self' is interpreted in a daemon body as the object that has changed. The value of 'self' currently in scope at the point of definition is referred to in each daemon as 'outerSelf'. In each daemon the name of the slots being synchronized are bound to the values after they have been updated.

constructor: `SyncContainers(leftContainer,leftAttribute,rightContainer,rightAttribute`

Left and right containers are expressions, attributes are strings and actions are performable elements of type SynContainerAction.

syntax:

```
SyncContainers ::= '@SyncContainers' '(' Exp ',' Name ',' Exp ',' Name ')' Exp* 'end'.
```

example:

```
@SyncContainers(company,employees,company,management)

  // A company has employees some of which are bosses.
  // Ensure that these are synchronized when one or other
  // change. Note, in this example the containers are
  // the same - they need not be.

  @AddLeft(newEmployee)

    // If we get a new employee then update the management
    // if necessary.

    if newEmployee.isABoss() and not management->includes(newEmployee)
    then company.addManager(newEmployee)
    end
  end

  @DeleteLeft(employee)

    // When an employee leaves they must be removed from
    // management if necessary.

    if employee.isABoss() and management->includes(employee)
    then company.deleteManager(employee)
    end
  end

  @AddRight(boss)

    // A new boss must be recorded as an employee.

    if not employees->includes(boss)
    then company.addEmployee(boss)
    end
  end

  @DeleteRight(boss)

    // A disgraced boss must be removed from the company
    // completely.

    if employees->includes(boss)
    then company.deleteEmployee(boss)
    end
  end
end
```

interface:

```
@Class SyncContainers extends Sugar
  @Attribute actions : Seq(Performable) end
  @Attribute leftAttribute : String end
  @Attribute leftContainer : Performable end
  @Attribute rightAttribute : String end
  @Attribute rightContainer : Performable end
  @Operation desugarActions end
  @Operation desugar end
end
```

28 Package XML

context: Root
overview:

The XML package defines classes that are used to represent XML documents.

28.1 Class Attribute

context: Root::XML
overview:

An XML element attribute. The name and the value are both strings.

interface:

```
@Class Attribute extends Object
  @Attribute name : String end
  @Attribute value : String end
  @Operation print end
  @Operation setValue end
  @Operation toString end
end
```

28.2 Class Document

context: Root::XML
overview:

An XML document. Typically an XML document has been produced from a file and the resourceName of the document should define the file. A document has a root which is an XML node. A document that conforms to XMF.dtd can be reduced to form an XMF data value. Documents can be printed to an output channel.

interface:

```
@Class Document extends Resource
  @Attribute root : Node end
  @Operation print end
  @Operation root end
  @Operation reduce end
  @Operation setRoot end
  @Operation toString end
end
```

28.3 Class ExpectingTag

context: Root::XML
overview:

This exception is raised when element processing expects an element with a given tag and when the received node does not match.

interface:

```
@Class ExpectingTag extends Exception
  @Operation init end
end
```

28.4 Class `Element`

context: `Root::XML`
overview:

> An XML element has a tag, some attributes and some children. Use 'get/1' to access the attribute values by name. If the element conforms to XMF.dtd then it can be reduced to an XMF data value using 'reduce/1' where the argument is a table of element identifiers to XMF data values.

interface:

```
@Class Element extends Node
  @Attribute attributes : Set(Attribute) end
  @Attribute children : Seq(Node) end
  @Attribute tag : String end
  @Operation addToChildren end
  @Operation children end
  @Operation deleteFromChildren end
  @Operation get end
  @Operation nameSpaceRef end
  @Operation put end
  @Operation print end
  @Operation reduceSlot end
  @Operation reduceObject end
  @Operation reduceRef end
  @Operation reduce end
  @Operation setChildren end
  @Operation setTag end
  @Operation toString end
  @Operation tag end
end
```

28.5 Class `Node`

context: `Root::XML`
overview:

> An XML node can occur as the root of a document or the child of an element.

interface:

```
@Class Node isabstract extends Object
  @Operation print end
end
```

28.6 Class `NoAttribute`

context: `Root::XML`
overview:

> This exception is raised when an attribute of an element is requested and when the attribute is not defined.

interface:

```
@Class NoAttribute extends Exception
  @Operation init end
end
```

28.7 Class `Text`

context: `Root::XML`
overview:

A text node records the text occurring in an XML document.

interface:

```
@Class Text extends Node
  @Attribute text : String end
  @Operation print end
  @Operation setText end
  @Operation toString end
  @Operation text end
end
```

28.8 Class `UnknownTag`

context: `Root::XML`
overview:

This exception is raised when an XML document is being parsed and when an element with an unexpected tag is encountered.

interface:

```
@Class UnknownTag extends Exception
  @Operation init end
end
```

28.9 Operation `callcc`

context: `Root`

28.10 Operation `anonymous`

context:

28.11 Operation `gc`

context: `Root`

28.12 Operation `print`

context: `Root`

28.13 Operation `resetOperatorTable`

context: `Root`

28.14 Operation `restoreMachineState`

context: `Root`

28.15 Operation `saveMachineState`

context: `Root`