Classes, Classification and Meta-classes in XMF (Incomplete Draft)

Xactium Ltd.

July 8, 2004

1 Introduction

XMF is a class-based object-oriented modelling environment. Each value in XMF has a type or *classifier* that describes its structure and behaviour. Values in XMF are divided into objects and non-objects. Object types are called *classes* and non-object types are called *classifiers*. If a value v is of a type c then we say that v is an instance of c. XMF is provided with a large number of classes and classifiers; XMF developers can define their own classes and classifiers as extensions of those provided.

Classes and classifiers *classify* their instances by running *constraints*. A constraint is a boolean valued expression that runs in the context of the current state of the candidate instance. The outcome of constraint checking is a constraint report containing details of the constraints that were performed, the candidates, the outcome and a reason for any constraints that failed. Constraint checking is a powerful mechanism for checking whether a model or a model scenario is correctly formed.

XMF is a meta-modelling environment. Values are instances of classes and classifiers. Classes and classifiers are themselves instances of meta-classes; the meta-classes describe the structure and behaviour of their instances - including information relating to constraint checking. New types of classes can be created by extending the basic meta-classes provided by XMF. Meta-extensions are a powerful tool that allows XMF to be extended with new modelling features.

This note describes features of XMF classes and classifiers.

2 Classes

A class describes the structure and behaviour of its instances. A class has a name and lives in a name-space. The following is a basic class that lives in the name-space `Root`:

```
context Root
  @Class EmptyClass end
```

By default, the class `EmptyClass` specializes the XMF class `Object` and provides a single constructor for creating instances: `EmptyClass()`. If we perform the following expression:

```
EmptyClass().isKindOf(EmptyClass)
```

then the result is `true` since the newly created instance is directly an instance of `EmptyClasss`. In addition, the expression:

```
EmptyClass().isKindOf(Object) and EmptyClass().isKindOf(Element)
```

returns true since `EmptyClass` inherits (bu default) from `Object` and `Object` inherits from `Element` (the class `Element` does not inherit from anywhere). The class `EmptyClass` is itself a value:

```
EmptyClass.isKindOf(Class) and
EmptyClass.isKindOf(Classifier) and
EmptyClass.isKindOf(NamedElement) and
EmptyClass.isKindOf(Object) and
EmptyClass.isKindOf(Element)
```

returns `true`.

2.1 Attributes

A class typically defines some attributes that correspond to slots in the instancs of the class. Each attribute has a name and a type and may optionally have some modifiers and an initial value. The following is a simple example of a class with attributes:

```
context Root
  @Class Point
    @Attribute x : Integer end
    @Attribute y : Integer end
    @Constructor(x,y) ! end
  end
```

A new point is creates using the constructor defined by `Point`. A class may define any number of constructors; each constructor must have a different number of arguments. Each constructor argument corresponds to one of the attributes in the class. The optional modifier `!` declares that the printed representation for a point is defined by that constructor. A new `Point` is constructed:

```
Point(100,200)
```

where `100` is the value of the slot `x` and `200` is the value of the slot `y`. Accessor and updater operations are automatically produced by including attribute modifiers:

```
context Root
  @Class Point
    @Attribute x : Integer (?,!) end
    @Attribute y : Integer (?,!) end
    @Constructor(x,y) ! end
  end
```

The modifier `?` defines that operations `getX` and `getY` are automatically provided; modifier `!` defines that operations `setX` and `setY` are automatically provided:

```
let p = Point(100,200)
in p.setX(p.getX() - 1);
   p.setY(p.getY() - 1)
end
```

2.2 Inheritance

All classes are specializations or *sub-classes* of at least one other class. By default a class is a sub-class of `Object`. A class inherits constructors, attributes, operations and constraints from its parent. For example the following class specializes `Point` with a `z` co-ordinate:

```
context Root
  @Class Point3D extends Point
    @Attribute z : Integer (?,!) end
    @Constructor(x,y,z) ! end
  end
```

An instance of `Point3D` may be constructed using a two-place constructor or a three-place constructor (the three-place is probably more useful). An instance of the class `Point3D` is also an instance of the parent class `Point`:

```
let p = Point3D(1,2,3)
in p.isKindOf(Point)
end
```

returns `true`.

2.3 Operations

Object-oriented execution proceeds by message passing; a message consists of a name and some argument values. When a message is sent to an object, the name is looked up in the class of the object (and its parents); if an operation is found then it is invoked otherwise an error is reported. Operations may be defined as part of a class or a package definition or added to existing classes and packages via a `context` defintion.

The following defines a class of stacks and adds the definition to a package named `Stacks`. The example shows the complete contents of a file containing

the definition. XMF can be used in a file-based mode where files contain source code that is compiled using the XMF compiler. The compiled binary is then loaded into XMF.

```
parserImport XOCL;

context Stacks

  @Class Stack

    @Attribute elements : Seq(Element) end
    @Attribute index : Integer end

    @Operation isEmpty():Boolean
      self.elements->isEmpty
    end

    @Operation pop()
      if self.isEmpty()
      then throw StackUnderflow(self)
      else
        let head = elements->head
        in self.elements := elements->tail;
           head
        end
      end
    end

    @Operation push(e:Element)
      self.elements := Seq{e | elements}
    end

   @Operation top()
      if self.isEmpty()
      then throw StackUnderflow(self)
      else elements->head
      end
    end

  end
```

3 Constraints

4 Classifiers

5 Meta-classes