The XMF Meta Object protocol

Tony Clark

May 11, 2004

1 Introduction

XMF is an environment for language design and deployment. Languages control the structure and behaviour of the values that they denote. In this sense, language design is a *meta-activity* and requires a meta-language that represents the structure and behaviour of the language components.

XMF provides a meta-circular object-oriented kernel language called XOCL. The meta-circular property means that XOCL is defined completely in itself. This property validates XOCL as a meta-language. Object-orientation provides a basis for application extension and reuse through inheritance and modularity through encapsulation.

XOCL is both meta-circular and object-oriented, it is suitable for language definition where languages can be easily constructed as modular extensions of the basic XOCL language. Instantiations of XOCL are languages and extensions of XOCL are meta-languages.

All languages have key semantic features that can be represented as an interface in the definition of the language. Consider a language with an operational semantics. In this case programs written in the language may be viewed as controlling a machine that contains the state of the execution at any given snapshot in time. The key semantic features of the language form the API of the machine.

Given a language L, we would like to construct a new language that is L-like. If L is defined using object-oriented principles then it is attractive to construct the new language as an extension of L using inheritance. Syntax structures and values of the new language can be defined by extending the appropriate features of L. We would like to construct the semantics of the new language using the same approach. If we have constructed the semantics of L by encapsulating the key features as an implementation of the API as described above, then the new language semantics can be defined by inheriting and extending these operations as appropriate.

Where the semantics of a language has been constructed using object-oriented principles, the resulting collection of classes and operations is referred to as a *meta-object-protocol*. This document describes the XOCL MOP.

2 Message Passing

XOCL performs computation in terms of messages between elements. A message consists of a name and some data. A message is sent from a source element to a target element. The target element receives the message, performs appropriate computation and returns a result. Messages between elements are synchronous: the source element halts computation and waits the return value from the target element.

Message passing occurs when the source element performs an expression of the form: `o.m(x,y,z,...)` where `o` is the target element, `m` is the message name and `x`, `y`, `z` etc. is the data, or *parameters*, of the message.

Message passing is defined by the MOP component referred to as the *massage passing protocol*. The protocol is defined by the meta-class of the target element and is called `sendInstance`:

```
context Element
  @Operation send(message,args):Element
    self.of().sendInstance(self,message,args)
  end
```

A default protocol is provided by the Kernel meta-class `Classifier`:

```
context Classifier
  @Operation sendInstance(element,name,args)
    // Get all the operations of element with the
    // correct name and arity. Select the most
    // specific and invoke it.
    let arity = args->size then
        operations = element.of().allOperations()
          ->asSeq
          ->select(o | o.name = name and o.arity() = arity)
    in if operations->isEmpty
       then element.error("Cannot handle " + message + "/" + arity)
       else operations->head.invoke(element,args)
       end
    end
  end
```

Since `Class` is a sub-class of `Classifier`, any sub-class of `Class` that defines a new `sendInstance` operation will provide a specialized message passing protocol for the instances of its instances. This can be used to implement specialized operation lookup mechanisms, to facilitiate debugging information and to change the basic message passing mechanisms (for example by defining a class of objects with message queues).

3 Object Creation

Objects are created by sending a `new` message to a class together with some initialization data. The preferred way of invoking the `new` operation of a class is to apply the class as an operator to the initialization arguments. This is preferred because it is succinct and because the compiler and XMF VM can handle class instantiation more efficiently in this form:

```
context Classifier
  @Operation invoke(target:Element,args:Seq(Element)):Element
    self.new(args)
  end
```

The `new` operation is defined by the meta-class of the receiving class. It constitutes the instantiation procotol for a collection of classes. The class `Classifier` defines the default instantiation protocol:

```
context Classifier
  @Operation new(args:Seq(Element)):Element
    self.new().init(args)
  end
```

where the operation `new` creates an empty new instance of the receiver and `init` initializes the new instance.

Sub-classes of `Classifier` can define their own instantiation protocol. Typically this will use `super` to create an instance using the default protocol and then perform some extra computation to initialize the new instance; however, in principle the instantiation protocol can by-pass the default protocol altogether. To create a raw instance of a class `C` and add a single slot named `"x"` with initial value `10` you can do the following:

```
let o = Kernel_mkObj()
in Kernel_setOf(o,C);
   Kernel_addAtt(o,Symbol("x",100));
   o
end
```

Using the kernel-level operations, you can create a completely bespoke instantiation protocol.

4 Slot Access

Objects have internal storage in the form of named *slots*. Access to a slot value is via the object and the name of the slot. Slots are named using symbols. Access is defined by the object's *slot access protocol*. The slot access protocol is used when an expression of the form `o.a` is performed. Access involves checking that the slot exists and then accessing the value of the slot.

The existence of a slot can be checked using the `hasSlot` operation defined by `Object`:

```
context Object
  @Operation hasSlot(name):Boolean
    self.of().hasInstanceSlot(self,name)
  end
```

The `hasSlot` operation invokes the `hasInstanceSlot` operation of the object's class. `hasInstanceSlot` forms part of the slot access protocol for the object; the operation is defined by the object's meta-class. The default definition is provided by `Class` and uses the kernel operation `Kernel_hasSlot` to directly check whether there is a machine-level slot:

```
context Class
  @Operation hasInstanceSlot(object,name)
    Kernel_hasSlot(object,name)
  end
```

Access to a slot's value is provided by the operation `get` defined by `Object`:

```
context Object
  @Operation get(name:String):Element
    self.of().getInstanceSlot(self,name)
  end
```

The operation `getInstanceSlot` is defined by an object's meta-class and describes how to access the storage associated with an object and a slot name. The default protocol is provided by `Class` and uses the kernel operation `Kernel_getSlotValue` to access the machine-level slot (as added using `Kernel_addAtt`):

```
context Class
  @Operation getInstanceSlot(object,name)
    Kernel_getSlotValue(object,name)
  end
```

Typically a new slot access protocol is required because a collection of classes implement object storage in a non-standard way (for example using a table, in a data base or distributed over a network).

5 Slot Update

The value of an object's slot can be updated using the object's *slot update protocol*. A slot is updated when an expression of the form `o.a := e` is performed. The class `Object` provides an operation used to set the value of a slot:

```
context Object
  @Operation set(name:String,value:Element):Element
    self.of().setInstanceSlot(self,name,value);
    self
  end
```

The object's meta-class defines an operation `setInstanceSlot` that forms the update protocol. The default update protocol is defined by `Class` and uses the kernel-level operation `Kernel_setSlotValue` to update the machine-level slot and to invoke any daemons that are defined on the object:

```
context Class
  @Operation setInstanceSlot(object,name,value)
    Kernel_setSlotValue(object,name,value)
  end
```

A new slot update protocol is used to circumvent the default storage. For example the storage for a slot may be in a database or accessed over a network.

6 Default Parents

A class is created as an instance of a *meta-class*. When a class is created it must have some parents. The meta-class defines an operation `defaultParents` that produces a set of classes that are the default parents for its instances. The basic definition for `defaultParents` is provided by `Classifier`:

```
context Classifier
  @Operation defaultParents():Set(Classifier)
    Set{Element}
  end
```

Most classes are instances of the class `Class`, that overrides the definition as follows:

```
context Class
  @Operation defaultParents():Set(Classifier)
    Set{Object}
  end
```

7 Example

Suppose that we want to define a class of objects that can have standard attribute defined slots in addition to *dynamic slots*. Attribute defined slots are defined at the class level. Dynamic slots are defined at the object level and can be added and removed dynamically. Both types of slot can be accessed and updated via the standard protocols using `o.a` and `o.a := e` expressions.

In order to implement these objects we require a new slot access and update protocol. The protocol is defined at the meta-level and is to be called the *Elastic* protocol. We require two new classes: `ElasticObject` that is the super-class of all user-defined elastic classes; and, `ElasticClass` that defines the elastic protocol.

The class `ElasticObject` uses a table to contain the dynamic slots:

```
context Root
  @Class ElasticObject
    @Attribute slots : Table = Table(100) end
  end
```

An elastic object provides operations to add and remove the dynamic slots:

```
context ElasticObject
  @Operation addSlot(name:String,value)
    slots.put(Symbol(name),value)
  end

context ElasticObject
  @Operation removeSlot(name)
    slots.remove(Symbol(name))
  end
```

An elastic object can remove all the dynamic slots:

```
context ElasticObject
  @Operation removeAll()
    @For key inTableKeys slots do
      self.removeSlot(key.toString())
    end
  end
```

An elastic object can increment the values of all the dynamic slots. Note that `incAll` uses the slot access and update protocol for the object to change the value of the dynamic slots:

```
context ElasticObject
  @Operation incAll()
    @For key inTableKeys slots do
      self.set(key,self.get(key) + 1)
    end
  end
```

The class `ElasticClass` defines the elastic MOP:

```
context Root
  @Class ElasticClass extends Class
  end
```

`ElsaticObject` must be a parent of any elastic class:

```
context ElasticClass
  @Operation defaultParents()
    Set{ElasticObject}
  end
```

The elastic slot access protocol inspects the `slots` table to see if the required slot is defined there. If not then the protocol uses `super` to revert to the default protocol inherited from `Class`:

```
context ElasticClass
  @Operation getInstanceSlot(object,name)
    if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
    then Kernel_getSlotValue(object,Symbol("slots")).get(name)
    else super(object,name)
    end
  end

context ElasticClass
  @Operation setInstanceSlot(object,name,value)
    if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
    then Kernel_getSlotValue(object,Symbol("slots")).put(name,value)
    else super(object,name,value)
    end
  end

context ElasticClass
  @Operation hasInstanceSlot(object,name)
    if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
    then true
    else super(object,name)
    end
  end
```

There is no specific need for `sendInstance` in the elastic protocol, however it is defined for completeness and simply prints a message before reverting to the default protocol:

```
context ElasticClass
  @Operation sendInstance(element,message,args)
    format(stdout,"Sending message ~S(~{,~;~S~})~%",Seq{message,args});
    super(element,message,args)
  end
```

There is no specific need for `new` in the elastic protocol, however it is defined for completeness and simply prints a message before reverting to the default protocol:

```
context ElasticClass
  @Operation new(args)
    format(stdout,"Creating a new instance of an elastic class~%");
    super(args)
  end
```

The following is an example class definition that specifies its meta-class as `ElasticClass`:

```
context Root
  @Class C metaclass ElasticClass
    @Attribute s : Element end
    @Operation test()
      self.addSlot("x",100);
      self.addSlot("y",200);
      self.addSlot("z",300);
      self.x := self.x + 1;
      self.y := self.y + 1;
      self.z := self.z + 1;
      self.incAll();
      self.s := self.x + self.y + self.z;
      self.removeAll();
      s
    end
  end
```