EASST

OCL 09: The Pragmatics of OCL and Other Textual Specification Languages

A MOP Based DSL for Testing Java Programs using OCL

Tony Clark

15 pages

A MOP Based DSL for Testing Java Programs using OCL

Tony Clark

Thames Valley University, St Mary's Road, Ealing, UK, tony.clark@tvu.ac.uk
http://itcentre.tvu.ac.uk/~clark/

Abstract: OCL is used to specify systems by defining pre and post-conditions for class operations. Typically, the conditions refer to properties and operations that are defined in a model. When the model is implemented, various implementation decisions are made regarding properties and operations that cause the OCL conditions to be inconsistent with the implementation. This paper defines a domain specific language (DSL) for testing and shows how a meta-object-protocol for OCL can be used to dynamically run tests written in the DSL against different Java implementations of the same model.

Keywords: OCL, meta-object protocol, domain-specific language, testing.

1 Introduction

The Object-Constraint Language (OCL) is used to specify the behaviour of system operations in terms of pre and post-conditions that are defined for a UML class model. Once specified, the system is implemented. The implementation involves making many technology decisions relating to partitioning, structure, messaging mechanisms, object instantiation, distribution, persistence etc.

The system tests are derived from the original model. Ideally it should be possible to run the original OCL pre and post-conditions against the implementation. However, OCL does not provide a definition of *how* to connect to a system implementation. Furthermore, the OCL constraints are defined against the original model which is different, due to technology decisions, to the implemented system.

In order to make use of a model involving OCL pre and post-conditions in system testing two key issues must be addressed: there must be a mechanism for linking the model with an implementation; and, there must be a mechanism for bridging the difference between the original model and the implementation.

Our hypothesis is that the language used to express tests in terms of OCL pre and post-conditions will depend on the approach taken to testing, therefore it is appropriate to embed OCL within a *domain specific language* for testing whose semantics provides the required test executions and reporting. Furthermore, it is proposed that a *meta-object protocol* used as the basis for OCL within the DSL is a suitable mechanism for bridging the difference between the original model and the implementation.

The contribution of this paper is to show how OCL can be embedded within a DSL for testing. XMF [1] is an open-source object-oriented language for meta-programming and language engineering. XMF is used to define the DSL since it is based on OCL and provides technology for DSL definition; however the approach could be used within any suitable technology. A *meta-*
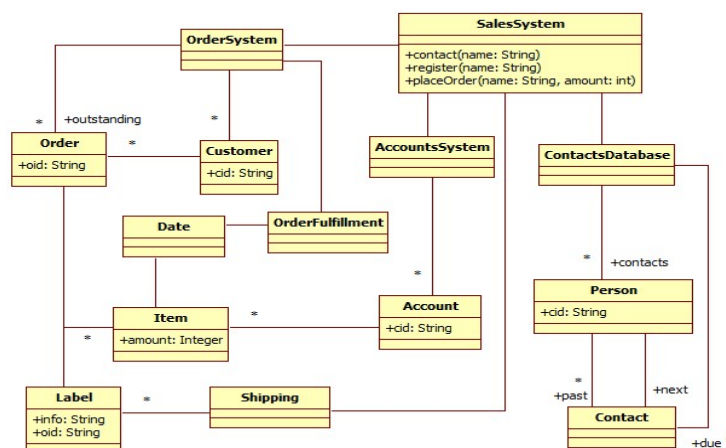
Figure 1: A Sales System

object protocol (MOP) [17] allows the execution mechansism of a language to be controlled by the programmer. A further contribution is to define the XMF MOP that controls how OCL can be linked to Java, thereby bridging the implementation issue.

This paper is structured as follows: section 2 describes a simple model that will be used as a case study; section 3 describes the key features of the XMF platform that are used to implement the DSL and the MOP; section 4 defines a testing DSL and gives examples in terms of the case study; section 5 describes how the DSL is implemented; section 6 describes how the MOP can be used to bridge the implementation gap; finally section 7 analyses the approach and describes related systems.

2 A Model and its Java Application

Fugure 1 shows a model of a sales system (taken from [9]) that consists of components for recording contacts, registering customers, placing orders and delivering orders. The idea is that sales representatives make contact with prospective customers who subsequently register with the system. Once registered a customer can place orders for items and the orders are subsequently shipped.

Consider the operations `contact`, `register` and `placeOrder` defined on the class `SalesSystem`. Each operation can be specified using OCL pre and post-conditions as shown in figure 2: a contact should not be made twice and causes a change in the contacts database; a contact must be made before a customer can register; an order extends a customer's account with a new item.

The operation specifications given in figure 2 are correct with respect to the model in figure 1. The specifications do not constitute a test script since there is no way to link them to an implementation. Furthermore, there are a large number of possible implementation choices. For

```
context SalesSystem::contact(name:String)
  pre: not contactsDatabase.contacts->exists(p | p.cid = name)
  post: contactsDatabase.contacts->exists(p | p.cid = name)

context SalesSystem::register(name:String)
  pre: contactsDatabase.contacts->exists(p | p.cid = name) and
       not accountsSystem.accounts->exists(a | a.cid = name)
  post: accountsSystem.accounts->exists(a | a.cid = name)

context SalesSystem::placeOrder(name:String,amount:int)
  pre: accountsSystem.accounts->exists(a | a.cid = name)
  post: accountsSystem.getAccount(name).items->exists(item |
         item.amount = amount) and
         accountsSystem.getAccount(name).items->size =
         self@pre.accountsSystem.getAccount(name).items->size + 1
```

Figure 2: Sales System Operation Specifications

example, in Java, associations with multiplicities greater than 1 may be implemented as vectors or arrays or some associations may be viewed as derived.

3 XMF Features

This paper proposes that OCL should be embedded into a DSL testing language on an application specific basis and that a MOP should be used to map from a model to the system implementation in order that the OCL pre and post-conditions can be run against the implementation. This paper shows how this can be achieved using the XMF platform. This section reviews the key features of XMF that will be used; the features are explained in terms of a simple Library example.

XMF is an engine that provides a collection of features that support language design. XMF provides an object-oriented language based on an imperative version of OCL. For example, the following is a pair of class definitions for a library containing a collection of books:

```
context Root
  @Class Book
    @Attribute name : String end
  end
context Root
  @Class Library
    @Attribute books : Seq(Book) end
  end
```

Top-level named elements are added to a name-space using the `context` keyword. In the example above, both `Book` and `Library` are added to the global name-space `Root`. Language features in XMF are preceded by @ followed by the name of a *syntax class* that defines the concrete and abstract syntax representations for the feature. XMF provides many built-in language features, such as `Class` and `Attribute` above, and allows users to define their own. In this paper we will define two new language features that support testing Java methods.

Nested named elements can be defined inside the containing name-space or can be added using `context`. An operation is a named element that can be added to a class (which is a name-space):

```
context Library
  @Operation addBook(b:Book):Library
    self.books := books->including(b)
  end
```

The `addBook` operation uses the OCL `including` operation to add the supplied book to the value of the attribute `books`.

Everything in XMF has a type that describes the structure and behaviour of its instances. Classes have types which are `meta-classes`. For example, suppose that the class `Book` is redefined to keep track of all its instances:

```
context Root
  @Class InstanceManager extends Class
    @Attribute allInstances : Set(Element) end
    @Operation new()
      let object = super()
      in self.allInstances := allInstances->including(object);
        object
      end
    end
  end
context Root
  @Class Book
    @Attribute name : String end
  end
```

A *syntax class* introduces a new language feature by defining a grammar that processes concrete syntax. Once defined, the new language feature `F` can occur in any XMF program using the reference `@F ....`. The syntax class is responsible for synthesizing abstract syntax that uses existing language features and OCL. For example, it would be convenient to construct a library by just listing the names of the books initially on the shelves:

```
context Library
  @Grammar
    Library ::= names = Name* 'end' {
      names->iterate(n exp = [| Library() |] |
        [| <exp>.addBook(Book(<n.lift()>)) |])
    }.
  end
```

A grammar consists of a collection of named rules, one of which must have the name of the syntax class: this is the starting non-terminal. A library consists of a sequence of names, bound to the rule variable `names` followed by the terminal `'end'`. The synthesizing action within { and } creates an abstract syntax tree that constructs a library and populates it with books. The names of the books are supplied in the language feature; an `iterate` expression is used to process the names and transform them into calls of `addBook`. XMF provides quasi-quotes (`[|` and `|]`) and drop-quotes ($<$ and $>$) for syntax *templates* where quasi-quotes construct abstract syntax trees using concrete syntax and drop-quotes provide template holes. The operation `lift` is defined for any XMF value and returns an expression that, when executed at run-time, reconstructs the value. Therefore the following library:

```
@Library book1 book2 end
```

is transformed to:

```
Library().addBook(Book("book1")).addBook(Book("book2"))
```

XMF provides *code walkers* that can be used to translate source code. A walker is a class that is supplied with an instance of a given class and then traverses the structure calling operations on the sub-components. A code walker defines operations for all the OCL language features. For example, suppose that large libraries are to be defined and that it is much more efficient to set the `books` attribute of a library rather than make many calls to `addBook`. A walker might be defined:

```
context Root
  @Class ReplaceAddBook extends OCLWalker
    @Operation walkSend(target,name,args)
      if isNestedAddBook(target,name,args)
      then [| <target>.books := <getBookExps(args)> |]
      else super(target,name,args)
      end
    end
  end
```

where `isNestedAddBook` returns true when the supplied arguments are a chained call of `addBook`, and where `getBookExps` transforms a collection of nested `addBook` calls to a sequence expression containing the book expressions. Using the code walker, the library feature is translated to:

```
Library().books := Seq{Book("book1"),Book("book2")}
```

Finally, XMF provides an interface to Java where compiled Java classes can be manipulated as ordinary XMF classes. For example, suppose that the Book class was implemented in Java in a package called `library` then the Java class can be loaded:

```
Book ::= xmf.javaClass("library.Book")
```

after which, subject to a suitable Java constructor, the Java class can be instantiated and used just like a normal XMF class. When XMF performs an operation (object creation, slot access, slot update and method invocation) on an instance of such a class then the XMF VM makes use of a user defined *meta-object protocol* (MOP) written in Java that defines how to handle the operation. A default MOP that performs the obvious operation is supplied and used by default.

It is useful to be able to extend the Java classes that are loaded into XMF. The meta-class `JavaClass` is provided that allows an XMF class to *wrap* a Java class and add new attributes and operations to it. Consider a situation where the classes `Library` and `Book` are both implemented in Java, then:

```
context Root
  @Class Library metaclass JavaClass
    JavaDescriptor("library.Library")
    @Operation hasBook(name:String):Boolean
      books->exists(b | b.name = name)
    end
  end
```

Instantiating the class defined above creates an instance of the Java class named in the `JavaDescriptor`. Methods and fields defined by the Java class are available within XMF. The operation `hasBook` shows how an existing Java class in this case `library.Library`) is extended with definitions involving OCL (in this case `exists`).

```
context SalesSystem
  @MSpec successfulContact[contact](name)
    pre not contactsDatabase.contacts->exists(p | p.cid = name)
    do run
    post contactsDatabase.contacts->exists(p | p.cid = name)
  end
context SalesSystem
  @MSpec successfulRegister[register](name)
    pre contactsDatabase.contacts->exists(p | p.cid = name) and
        not accountsSystem.accounts->exists(a | a.cid = name)
    do run
    post accountsSystem.accounts->exists(a | a.cid = name)
  end
context SalesSystem
  @MSpec successfulPlaceOrder[placeOrder](name,amount)
    pre accountsSystem.accounts->exists(a | a.cid = name)
    do run
    post accountsSystem.getAccount(name).items->exists(item |
            item.amount = amount) and
         accountsSystem.getAccount(name).items->size =
         preSelf.accountsSystem.getAccount(name).items->size + 1
  end
```

Figure 3: Sales System Specifications

4 A Testing Language

Given a sales system defined in Java, our aim is to specify methods in terms of pre and post-conditions and to define test scenarios as sequences of method calls. These can be attached to the appropriate Java classes by defining a new language feature. Each of the Java classes are defined in XMF using the meta-class `JavaClass`. The language feature for method specification is:

```
@MSpec <name> [<method-name>] (<args>)
  pre <pre-condition>
  do <body>
  post <post-condition>
end
```

where `name` is the name of the specification (a given Java method may have more than one specification), `method-name` is the name of the specified Java method, `args` are the names of arguments to the Java method, `pre-condition` and `post-condition` are OCL expressions, `body` is an XMF command. The semantics of a method specification is that if the pre-condition is true then the body is performed and the post-condition is expected to be true. The body may reference the special variable `run` which causes the Java method to be called with the supplied arguments. The post-condition may reference `preSelf` which is the state of the receiver of the Java message *before* the body is performed. The OCL constraints defined in section 2 are shown, written in the DSL, in figure 3. A scenario language feature just lists the steps in the scenario:

```
context SalesSystem
  @Test test1
    successfulContact("fred")
    successfulRegister("fred")
    successfulPlaceOrder("fred",100)
  end
```

5 Language Implementation

The testing language features described in section 4 are defined as syntax classes in XMF. The method specification feature is defined in section 5.1 and the testing scenario feature is defined in section 5.2.

5.1 Method Specifications

A method specification is added as a new operation to an instance of `JavaClass`. The pre and post-conditions are expressed in OCL and are checked respectively before and after the body of performed. The result of calling a method specification depends on whether the pre and post-conditions are satisfied. Firstly, the pre-condition is checked, if that fails then the specification returns. Otherwise, the body of performed and the post-condition is checked. The rest of this section describes the implementation of the language feature in detail.

The structure of the `MSpec` class is defined as follows:

```
context Root
  @Class MSpec extends XOCL::Sugar
    @Attribute name : String end
    @Attribute opName : String end
    @Attribute args : Seq(String) end
    @Attribute pre  : OCL end
    @Attribute body : Performable end
    @Attribute post : OCL end
    @Constructor(name,opName,args,pre,body,post) ! end
    @Operation body()
      Subst([| self.send(<opName.lift()>,args) |]).walk(body)
    end
  end
```

Note that the `pre` and `post` attributes are of type `OCL` while the body is any `Performable` action. `MSpec` extends the class `XOCL::Sugar` which means that the class must provide an operation `desugar` that is used by the XMF parser to synthesize abstract syntax. Therefore, instead of returning abstraction syntax from the grammar rules, the actions simply create an instance of `MSpec` and leave the synthesize work to `desugar`.

The operation `body` is defined to replace all occurrences of the variable `run` with a call to the Java method. The code walker `Subst` is initialized with an expression and then walks the body of the method specification. Its definition is as follows:

```
context Root
  @Class Subst extends Walkers::Code::OCLWalker
    @Attribute new : OCL end
    @Constructor(new) end
    @Operation walkVar(line,name,arg)
      if name = "run"
      then new
      else super(line,name,arg)
      end
    end
  end
```

The grammar for the `MSpec` language feature is defined below:

```
context MSpec
  @Grammar extends OCL::OCL.grammar
    MSpec ::= n = Name '[' o = Name ']' as = MArgs
```

```
              p = Pre d = Do q = Post 'end'
              { MSpec(n,o,as,p,d,q) }.
  MArgs ::= '(' as = CNames ')' {as}.
  CNames ::= n = Name ns = (',' Name)* { Seq{n|ns} } | { Seq{} }.
  Pre ::= 'pre' Exp.
  Do ::= 'do' Command.
  Post ::= 'post' Exp.
end
```

Notice that the `MSpec` grammar extends the XMF-supplied grammar for `OCL`. XMF allows grammars to be extended and therefore all the rules from the parent are included in the child. The `OCL` grammar provides the rule named `Exp` that parses and synthesizes OCL expressions.

The `desugar` operation is responsible for returning abstract syntax that implements a method specification. The implementation is just an operation with the method specification name. Notice that `desugar` is an operation that returns an *operation definition expression*. The arguments of the operation expression `.args` is equivalent to the Java `args...` varargs feature:

```
context MSpec
  @Operation desugar()
    [| @Operation <name> (.args)
        <0.to(args->size-1)->iterate(i x = self.desugarBody() |
         [| let <args->at(i)> = args->at(<i.lift()>)
            in <x>
            end |])>
      end |]
  end
```

The body of the operation is code that binds the names of the arguments to the appropriate element of the `args` run-time argument. The names are indexed using `at` at compile-time and the values are indexed at run-time.

The body of the specification operation is produced by `desugarBody`:

```
context MSpec
  @Operation desugarBody()
    [| let preSelf = self.deepCopy()
      in if <pre>
          then
            let result = <self.body()>
            in if <post>
               then CallSucceeds(result,<opName.lift()>,args)
               else PostFails(<opName.lift()>,args)
               end
            end
          else PreFails(<opName.lift()>,args)
          end
      end
    |]
  end
```

The body binds a run-time variable `preSelf` to a deep copy of the receiver. The implementation of `deepCopy` (not shown) is implemented on a case-by-case basis by extending the underlying Java class from within XMF. The variable `preSelf` is used in the post-condition where the values of the fields in the current state of the receiver can be compared to those before the body was performed. There can be three outcomes each of which is an instance of a different class: `PreFails` described the situation where the pre-condition fails; `PostFails` describes the situation where the post-condition fails; and, in `CallSucceeds` both conditions are satisfied and the result is returned.

5.2 Test Scripts

Test scripts are sequences of calls. A call is just a name and argumemts:

```
context Root
  @Class Call
    @Attribute name : String end
    @Attribute args : Seq(Performable) end
    @Constructor(name,args) ! end
    @Operation desugar()
      [| self.send(<name.lift()>,<args->iterate(arg x = [| Seq{} |] |
          [| <x> + Seq{<arg>} |])>)|]
    end
  end
```

When a call is translated to abstract syntax using `desugar`, the resulting expression sends a message to `self` containing a sequence of arguments. The argument sequence is constructed using `iterate` which chains together singleton sequences for each argument (expression).

The `Test` language feature is defined below:

```
context Root
  @Class Test extends XOCL::Sugar
    @Attribute name : String end
    @Attribute calls : Seq(Call) end
    @Constructor(name,calls) ! end
    @Grammar extends OCL::OCL.grammar
      Test ::= n = Name cs = Call* 'end' { Test(n,cs) }.
      Call ::= n = Name '(' as = TestArgs ')' { Call(n,as) }.
      TestArgs ::= e = Exp es = (',' Exp)* { Seq{e | es} }.
    end
    @Operation desugar()
      [| @Operation <name> ()
           <calls->reverse->iterate(call x = [| Seq{} |] |
             [| @Case <call.desugar()> of
                  CallSucceeds(result,name,args) do
                    Seq{CallSucceeds(result,name,args) | <x>}
                  end
                  PreFails(name,args) do
                    Seq{PreFails(name,args)}
                  end
                  PostFails(name,args) do
                    Seq{PostFails(name,args)}
                  end
                end |])>
         end |]
    end
  end
```

The definition of `desugar` above uses a `Case` expression to dispatch on the result of calling each method specification. It builds a sequence expression that will terminate if any of the pre or post-conditions fail to be satisfied. The result of a test scenario is a sequence of method specification outcomes which are either all instances of `CallSucceeds` or terminate with a `PostFails` or `PreFails`.

6 A Meta-Object Protocol

We have shown that a Java implementation of the sales system can be tested using OCL pre and post-conditions by implementing a testing DSL in XMF using an interface that allows Java

classes to be manipulated from XMF. The interface supports the creation of new instances, field access and update, and method invocation. The example has assumed that field references and method invocation in OCL maps directly onto the equivalent in Java. However, this is not always practical, since implementation choices translate properties and associations in a model and implement them in a variety of ways.

The Eclipse Modelling Framework (EMF) [2] is used to represent and manipulate models in Java. An EMF implementation of a model uses factories to create instances of objects and references fields using accessor and updater methods. This is a particular implementation choice compared to, say, using class constructors and direct field reference.

A meta-object protocol (MOP) can be used to make key execution features extensible. The key features for an object-oriented system are: object creation; field access; field update; method invocation. The MOP is implemented using a meta-class and a Java class. The defaults are `JavaClass` and `ForeignObjectMOP`. This section describes the key features of the standard MOP and shows how it can be extended.

6.1 A Standard MOP

A class is instantiated by applying it to initialization arguments. The behaviour for applying a class is defined by the meta-class:

```
context Java
  @Class JavaClass extends Class
    @Attribute descriptor : JavaDescriptor (?,!) end
  end
```

The meta-class `JavaClass` defined above extends the basic XMF `Class` with a descriptor that names a Java class in the file system. When a descriptor is supplied in the definition of a class with meta-class `JavaClass` the operation `addDescriptor` is used to process the descriptor:

```
context JavaClass
  @Operation addDescriptor(d:JavaDescriptor)
    xmf.foreignTypeMapping().put(d.type(),self);
    xmf.foreignMOPMapping().put(d.type(),d.mopName());
    self.setDescriptor(d)
  end
```

The operation `addDescriptor` uses the operations `foreignTypeMapping` and `foreignMOPMapping` in the system object `xmf` to inform the XMF virtual machine that instances of the XMF class are to be associated with instances of the Java class referenced in the descriptor, and to inform the machine of the MOP for the class.

When a class is applied to arguments in XMF it is instantiated. All meta-classes must implement an operation `invoke` that describes how to instantiate the receiver:

```
context JavaClass
  @Operation invoke(target,args)
    let class = xmf.javaClass(descriptor.type())
    in if class = null
       then self.error("Cannot find Java class " + descriptor.type())
       else class.invoke(target,args)
       end
    end
  end
```

```
public class ForeignObjectMOP {
  public void dot(Machine machine, int object, int name) {
    ForeignObject f = machine.getForeignObject(object);
    String string = machine.valueToString(machine.symbolName(name));
    int value = XJ.getSlot(machine, f.getObject(), string);
    if (value == -1)
      machine.sendSlotMissing(object, name);
    else machine.pushStack(value);
  }
  public void send(Machine machine, int target, int message, int args) {
    if (!handleByXOCL(machine, target, message, args))
      if (!handleByJava(machine, target, message, args))
        noOperationFound(machine, target, message, args);

  }
  public boolean hasSlot(Machine machine, int foreignObj, int name) {
    ForeignObject f = machine.getForeignObject(foreignObj);
    String string = machine.valueToString(machine.symbolName(name));
    return XJ.hasSlot(f.getObject(), string);
  }
  public void set(Machine machine, int obj, int name, int value) {
    ForeignObject f = machine.getForeignObject(obj);
    String string = machine.valueToString(machine.symbolName(name));
    if(XJ.setSlot(machine, f.getObject(), string, value) == -1)
      machine.sendSlotMissing(obj, name,value);
    else machine.pushStack(obj);
  }
}
```

Figure 4: A Basic Java MOP

When a `JavaClass` is invoked it uses the `javaClass` operator to load the Java class named in the descriptor and then sends it an `invoke` message. The VM knows that invoking a Java class causes it to be instantiated via a suitable constructor. The `foreignMOPMapping` operation associates a Java class with a MOP. The MOP is an instance of the XMF supplied class `ForeignObjectMOP` (the default) or one of its sub-classes. The basic features of the default MOP are shown in figure 4: the method `dot` implements field reference; `send` implements message passing; `hasSlot` tests whether an object has a slot; and, `set` sets the value. When the VM attempts to perform one of the standard operations on a foreign object it looks up the MOP for the object and invokes the appropriate method.

The XMF VM is a value of type `Machine` and represents Java objects as values of type `ForeignObject`. The XMF library `XJ` uses `java.lang.reflect` to implement type conversion back and forth between XMF values and Java values and also implements basic access to Java values. Notice in the definition of `send` we have omitted the definition of `handleByXOCL` and `handleByJava` which use basic machine and `XJ` defined primitives to perform message passing (returning true and pushing the return value if successful).

6.2 ECore MOP

The MOP defined in the previous section could be used to support the testing DSL providing that the implementation of the sales system is in one-to-one correspondence with the model shown in figure 1. Consider an EMF implementation of the model. In that case instantiation is performed

```
context Java
  @Class EMFClass extends JavaClass
    @Operation invoke(target,args)
      let factory = self.getFactory() then
          package = self.getPackage() then
          class = package.send("get" + name,Seq{}) then
          object = factory.create(class);
          constructor =
            @Find(c,self.allConstructors())
              when c.names->size = args->size
              else null
            end
      in if constructor <> null
          then
            @For name,value in constructor.names,args do
              object.set(name,value)
            end
          end;
          object
      end
    end
  end
```

Figure 5: The EMFClass meta-class

with respect to a factory and field access must use feature descriptors. This section shows how the basic MOP is extended to support Ecore.

Figure 5 shows how `JavaClass` is extended to support Ecore instantiation. The `invoke` operation is modified to use a specialization of `JavaDescriptor` that references the appropriate factory and containing package. The package is interrogated for the class to be instantiated and the factory is supplied with the class to produce a new object. Since Ecore factories do not support constructors (but XMF classes do), the fields are set based on the names defined in the appropriate constructor.

Figure 6 shows the specialization of `ForeignObjectMOP` to support Ecore field access. There is no difference between `EObjectMOP` message passing and `ForeignObjectMOP` message passing. In all cases, field access and update is performed with respect to feature descriptors.

7 Analysis and Review

A number of OCL interpreters exist, for example [11] and the Dresden toolkit (`http://dresden-ocl.sourceforge.net/index.php`). Where these systems address the variations in SUT implementation strategies, they do not use a MOP. Several tools and approaches exist for model based testing including AGEDIS [13], [3] and [19] however, none address the issue of associating OCL with an implementation. OCL is also used to validate *models* [4] and to describe the behaviour of platform independent models [16] where the implementation issue does not arise.

OCL is used as the source of tests and queries in a number of systems. For example [20] and[14] generate code from conditions expressed as OCL. In many cases model transformations

```
public class EObjectMOP extends ForeignObjectMOP {
  public void dot(Machine machine, int object, int name) {
    ForeignObject f = machine.getForeignObject(object);
    EObject eobject = (EObject)f.getObject();
    EClass eclass = eobject.eClass();
    String string = machine.valueToString(machine.symbolName(name));
    EStructuralFeature feature = eclass.getEStructuralFeature(string);
    if (feature == null)
      machine.sendSlotMissing(object, name);
    else
      machine.pushStack(XJ.mapJavaValue(machine, eobject.eGet(feature)));
    }
  public boolean hasSlot(Machine machine, int foreignObj, int name) {
    ForeignObject f = machine.getForeignObject(foreignObj);
    EObject eobject = (EObject)f.getObject();
    EClass eclass = eobject.eClass();
    String string = machine.valueToString(machine.symbolName(name));
    EStructuralFeature feature = eclass.getEStructuralFeature(string);
    return feature != null;
  }
  public void set(Machine machine, int obj, int name, int value) {
    ForeignObject f = machine.getForeignObject(obj);
    EObject eobject = (EObject)f.getObject();
    EClass eclass = eobject.eClass();
    String string = machine.valueToString(machine.symbolName(name));
    EStructuralFeature feature = eclass.getEStructuralFeature(string);
    if (feature == null)
      machine.sendSlotMissing(obj, name,value);
    else {
      Class type = feature.getEType().getInstanceClass();
      Object newValue = XJ.mapXMFValue(machine, type, value);
      eobject.eSet(feature, newValue);
      machine.pushStack(obj);
    }
  }
}
```

Figure 6: ECore MOP

are used to produce code; the approach described here allows the user to interact with the system under test (SUT) via the XMF interpreter without a separate compilation step.

Whether code is generated or the OCL is interpreted directly, the problem of taking implementation issues into account that differ from the model remains; the novel approach described here involves the use of a MOP to drive an interpretation engine for OCL, the same approach could be used to drive a transformation engine.

An earlier version of this work was presented as an invited talk at an ASTRANet workshop [5], as a tutorial [6] and in [7], where OCL expressions are expressed using both textual and graphical formats. This paper extends that work by addressing the issue of the implementation mapping.

MOPs were used in the definition of Smalltalk and the original description of how to implement a MOP is given in [17]. Code generation techniques using a MOP are described in [15] where the OpenC++ compiler is extended to allow tests to be inserted into code.

The XMF approach of using syntax classes to define DSLs is defined in [8]. The integration of OCL with model-based (i.e. MOF defined) DSLs is discussed in [18]. Other DSLs have

been defined that support testing, for example [12] which is not based on OCL and the language reported in [10] which is based on OCL but does not use a MOP to link to the implementation. JUnit can be viewed as a DSL for testing Java programs and can be used in conjunction with OCL engines. The work described in this paper is more flexibile through the use of MOPs and the meta-interface of Java.

References

[1] Xmf version 2.2, 2008. `http://itcentre.tvu.ac.uk/˜clark/downloads.html`.

[2] Eclipse modelling framework, 2009. `http://www.eclipse.org/modeling/emf/`.

[3] D. Arnold, J.-P. Corriveau, and V. Radonjic. Open framework for conformance testing via scenarios. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 775–776, New York, NY, USA, 2007. ACM.

[4] E. G. Aydal, R. Paige, and J. Woodcock. Observations for assertion-based scenarios in the context of model validation and extension to test case generation. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:11–20, 2008.

[5] T. Clark. Iswim for testing - a model driven approach. Invited Talk, Feb 2007. Presentation available at `http://itcentre.tvu.ac.uk/˜clark/Presentations/ISWIM For Testing.pdf`.

[6] T. Clark. A domain specific language for testing. Tutorial, Feb 2008. Tutorial available at `http://itcentre.tvu.ac.uk/˜clark/XMF/`.

[7] T. Clark. Model based functional testing using pattern directed filmstrips. In *Proceedings of the 4th International Workshop on the Automation of Software Test*. IEEE Computer Society, 2009.

[8] T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible java (xj). In *SCAM*, pages 229–238, 2008.

[9] D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Shupp. Concepts for model-based requirements testing of service oriented systems. In *Software Engineering*, 2009.

[11] M. Gogolla, F. Büttner, and M. Richters. Use: A uml-based specification environment for validating uml and ocl. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[12] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the design of the new testing language ttcn-3. *Testing of Communicating Systems. Ural, H, Probert, R L, von Bochmann, G (eds.). Dordrect, Kluwer*, 13:161–176, 2000.

[13] A. Hartman and K. Nagin. The agedis tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.

[14] F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from ocl invariants. *ECEASST*, 9, 2008.

[15] C. Hobatr and B. A. Malloy. Using ocl-queries for debugging c++. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 839–840, Washington, DC, USA, 2001. IEEE Computer Society.

[16] P. Kelsen, E. Pulvermueller, and C. Glodt. A declarative executable language based on ocl for specifying the behavior of platform-independent models, 2007.

[17] G. Kiczales. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.

[18] D. S. Kolovos, R. F. Paige, and F. Polack. Aligning ocl with domain-specific languages to support instance-level model queries. *ECEASST*, 5, 2006.

[19] A. Pretschner and J. Philipps. Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[20] Šarūnas Packevičius, A. Ušaniov, and E. Bareiša. Software testing using imprecise ocl constraints as oracles. In *CompSysTech '07: Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–6, New York, NY, USA, 2007. ACM.