ON REFLECTION AND REFRACTION

Paul Robertson
Dynamic Object Language Group
2 Parsonage Hill Road, Haverhill,
MA 01832

ABSTRACT

This paper explores problems in using OOP technology in a heterogeneous object environment. We describe experiments aimed at better understanding the role of a reflective protocol in facilitating such coexistence.

INTRODUCTION

In two decades of Object Oriented Programming (OOP), most practical experience has been with homogeneous environments. The movement toward object oriented applications, and the use of object oriented databases (OODB) and object oriented operating systems (OOOS), ensure that future object oriented programming will be increasingly heterogeneous. It is crucial that your object oriented programming language can cooperate with the host object environment, which might be an OODB or OOOS. By studying several hard but well understood problems, we have attempted to better understand how a language can support diverse object system interactions.

Claims persist that OOP will bring reusable code among its benefits. Can reusable code be extended to heterogeneous environments? Such extension is essential for OOOS and OODB's to succeed. One of the requirements on reusable code is portability, and interfacing with other object systems, since reusability theoretically derives from a separation of contract and implementation.

Experience has already shown that these issues are not typically well addressed by existing object systems. On the other hand reflective protocols seem to provide a level of abstraction that transcends a particular object system, and we believe, holds the key to solving the problem that we refer to here as refraction. Over the years the problems addressed by this work have been solved in conventional languages and object oriented languages [em-] in some cases in pre-MOP CLOS. Others have reported work using CLOS+MOP to solve some of these problems, for example Paepcke [AP90].

We have had numerous prior experiences with real world applications that required the object oriented language to cooperate with a host object environment[SYMB88], and we are familiar with the kinds of problems encountered. Our approach was to take these real world applications, reduce and simplify them to a manageable size, and see what is needed in a reflective protocol in order to make the interactions as easy as possible.

An experimental language Yolambda [RL92], an object oriented scheme dialect has been implemented in support of this work. The primary reason for inventing Yolambda was to support investigation into semantic issues. Yolambda is neither complete nor of production quality.

REFRACTION

We coin the term refraction to describe the difficulties resulting from trying to make your object oriented language of choice interface with the host object environment. For example, if a folder class is defined in the host environment, your application may need to create, use, and specialize folders within your object oriented language.

If a class and operations on it are provided in an OOOS, it should not be necessary to know the details of the internal data structures or how the operations are implemented.

In theory, only the contract of the operations and class should need to be known for both **use** and

specialization of the facility. In practice, if the provided facility was coded in C++, and you wish to use and extend it in Smalltalk[AG83], the implementation details must be exposed.

Only by addressing the interface issues directly can this issue of heterogeneous coexistence be solved. Our goal has been to encapsulate the interface issues in a clean way through a reflective protocol designed around this consideration. Reusability may then be achievable.

CLASS VS OBJECT SYSTEM

A common problem is caused by the accidental confusion between a class system and an object system. These two concepts are often used interchangeably and are even considered to be synonyms. The source of the problem is that when working with homogeneous systems, there is no reason to separate the concepts.

A class system provides a system of types with whatever generality or restriction the designer intended. It says nothing about how the types are realized as objects.

An object system provides a mechanism for creating and manipulating objects. Objects can be implemented as: in-core contiguous objects like conventional instances; or in a holographic manner (allocate only a tag, such as an integer which is used with the slots as keys to access values in a hash table); or as out-of-core objects such as with a conventional database; or as a system of composed first class functions; or as any representation that a designer may choose.

It is crucial to maintain a clear distinction between the class system and object system. When the implementation details of the object system for a class system are purely implicit, the distinction is difficult to maintain. In contrast, Yolambda explicitly represents the object system in it's metaclass structure. In this way, we have found it convenient in Yolambda to treat object systems as interchangeable by substituting at the metaclass level. Cleanliness of the reflective protocol makes this kind of cooperation with foreign object systems seem natural, and once the investment in the foreign object systems metaclass has been made, the foreign object system can be utilized with only minimal consideration for the implementation details -- notably restrictions, such as not being able to put transient pointers in a persistent object.
Yolambda accomplishes this by treating native and external objects in exactly the same way. In Yolambda, native objects are treated as external, and internalized by defining metaclasses for them. External objects can similarly be internalized.

Traditionally, reflective protocols have provided means of introspection. By expanding the scope of reflective protocols to encompass foreign object systems, a reflective protocol provides a mechanism that allows one to see into other object systems. The reflective protocol can be thought of as a lens construction kit, where the lens that you construct lets you see out into the object systems in the environment, whether they be OODB's OOOS's or custom application object systems. This has been the vision of the Yolambda eflective protocol.

METACLASS STRUCTURE

How an instance is represented as a bit pattern in the computer is a crucial role of a reflective protocol. How responsibility of instance creation and initialization are divided between the class and metaclass, dramatically affects the metaclass structure. For example, the Smalltalk metaclass system exists as a mirror of the class hierarchy. For every class, there is an anonymous metaclass, and the metaclass inheritance is a perfect mirror of the class structure. This isomorphism is forced by the way object creation and initialization is handled in Smal talk. The enforced isomorphism robs the Smalltalk programmer of a dimension of progra mability severely limiting the usefulness of Smalltalk metaclasses.

In CLOS[SEK89], object creation and instance initialization are separated avoiding the Smalltalk problem, Instead, CLOS unnaturally funnels classes through a small number of metaclasses, standard-class, structure-class, builtin-class. Complexity results from their overloaded use, in how MOP[BDGKKM] allows us to reflect on our classes.

In Yolambda, the reflective protocol is intended to encapsulate all knowledge of a class, creation of its instances, and allowable operations on it, in the form of a class instance and generic procedures. A class knows what it is to be an instance, how to access components of its instances etc. A metaclass on the other hand, knows what it is to be a class, how a class isconstructed, and how to access its components. How the instance is made/allocated/represented is the job of the metaclass, and as in CLOS, how the instance is initialized, is the job of the class. The class hierarchy is used to determine instance initialization, so that method combination on initializeInstance can cause the class to be appropriately initialized. As in Smalltalk, we provide a metaclass or each class, but because classes are responsible for initializeInstance, the inheritance isomorphism of Smalltalk is not encountered.

The programmer is encouraged to make effective use of metaclasses, because their natural decomposition allows normal method combination to work intuitively.

EXAMPLE PROBLEMS

Space limitations restrict the scope of examples that we can accommodate. Here we describe how metaobject structure is present even in Yolambda's own Metaclasses, and how an interface to the OS/2 System Object Model (SOM)[IBM91] was accommodated.

YOLAMBDA DATATYPES

Yolambda has Metaclasses for the following primitive physical representations:

Marker, Char, SmallInteger, String, Symbol, Pair:
These classes allocate compact representations that are efficiently implemented by the underlying substrate.

Instance:
This class catches all other cases. It allocates variable sized contiguous storage blocks used for user classes including arrays.

Yolambda makes explicit use of metaclass structure. For example, since metaclasses support multiple inheritance just the same as any other class, metaclasses may be mixed to the extent that it makes sense. Examples of such metaclasses that are often mixed in are:

CallableMetaclass:
when mixed into a metaclass, defines a physical object class that can be called as a procedure. Classes that use this metaclass are used to make trampolines, callable instances, generic procedures and any other user defined type that can be called.

BinaryMetaclass:
Defines a metaclass whose class instances will be constructed in a space not scanned by GC.

Others include separate metaclasses for mutable, weak, and resourced object types.

The object system has interesting structure in much the same way as objects in the real world do. Just as we represent that structure in our programs as classes, we seek to reflect the structure of our objects as classes -- metaclasses.

Concrete historical insight is hard to come by. Smalltalk metaclasses are inflexible because their inheritance structure is forced to mirror the class structure. CLOS is too new, and MOP is still absent from many implementations. Because CLOS+MOP bundles all simple primitive object classes into the flat builtin-class metaclass, and all CLOS instances are routinely made from standard-class, there is little historical support for building structure in metaclasses.

In Yolambda, since all datatypes are defined explicitly through metaclasses, the need for method combination and multiple inheritance in the metaclass hierarchy is evident. As a simple example lets look at the definition of IEEESingleFloat:

```
(defineClass IEEESingleFloat (ShortReal InexactNumber)
  ((sign bytesize: 1 position: 31)
   (exponent bytesize: 8 position: 23)
   (mantissa bytesize: 23 position: 0))
  Metaclasses: (Binary Instance Bitfield)
  Documentation: "Readonly float.")
```

Standard method combination takes care of combining the defining characteristics of the three inherited metaclasses and makes the anonymous metaclass for the IEEE floating point type. Subsequently a floating point object can be defined by:

```
(define fpexample
   (make IEEESingleFloat
         sign: 0
         exponent: 127
         mantissa: 0))
```

The above will define fpexample to be the IEEE single floating point number 1.0, the bitfield metaclass will take care of initializing the bitfields appropriately, the binary metaclass will cause the object to be allocated from binary space, and Instance will cause the representation to be a standard Instance header followed by the specified binary bits. By not mixing in Mutable, setters for the bitfield slots will not be generated, but accessors will be created.

In this simple example, none of the metaobject protocol's generic procedures needed to be specialized. Method combination adequately handled this case. We consider this simplicity as evidence that we are on the right track. In more complex cases we would specialize certain generic procedures so as to add support for extensions such as new slot options or class options. In such cases, we would make an explicit named metaclass so that we can refer to its name when specializing the MOP procedures.

ACCOMMODATING SOM CLASSES

In this example, we describe how we accommodated an OOOS external object system. SOM is the object system that is used to implement all Operating System objects in the OS/2 2.0 workplace shell. Although rather crude, SOM is one of the first OOOS extension environments. It represents a glimpse of future direction in OOOS development, and coexisting with it is a prerequisite for operating system extension. What we sought through the reflective protocol was a painless interaction with the Object system that allows bidirectional use of SOM.

By bidirectional use, we refer to the ability to use and specialize existing operating system classes like Folder, have generic procedures invoke operating system methods, and have making an

instance with [boldify,make] allocate a persistent operating system object. When we subclass a SOM class, we expect the new class to be available to other users of the operating system, and for generic procedures defined on this new class to be invoked when they are called by other operating system applications.

SOM is a full object oriented language that resembles Objective C with its own metaclass system and runtime types. SOM is designed to be language independent. An interface is available for C, providing function calls for such things as making instances, invoking methods, and registering and de-registering classes. Once registered, a class may be used by any user or application of the operating system. The operating system provides base WPS classes (WorkPlace Shell) with subclasses for common operating system objects, such as Folders, Programs, Datafiles, Sound, and the Shredder. Instances of WPS classes are persistent [em-] they survive termination of the program that generated them and reboot of the machine.

The mechanics of how code that implements a new class and its methods is registered with the operating system is complex, but the details are uninteresting to this discussion, so we don't elaborate them here. Interested readers are referred to the SOM guide [IBM91].

IMPORTING SOM

Before we can make an instance of an externally defined SOM class, we need a Metaclass for SOM classes, and a Class whose metaclass is SOM, that represents the externally defined class.

For an externally defined class som_frob, we will have a class internal to Yolambda (SomFrob) that represents the externally defined class.

```
(defineClass SomFrob ()
  ((foo SomOffset: 8 Class: Cint))
  Metaclass: SOM
  ExternalName: "som_frob"
  Proxy: #t
  Documentation: "Proxy for som_frob")
```

The above class definition defines a class SomFrob, that exists within the Yolambda class system as a proxy for the external SOM class sob-frob.

No Inherited classes are specified because the inheritance of this class is defined externally by "som_frob". When this class is created, the superclasses of som_frob are obtained by calling the appropriate SOM functions. In that way, the inherited structure of SomFrob can be automatically synthesized. This dynamic inheritance discovery is performed by a specialization of **newClass** on the metaclass SOM.

Unfortunately, SOM does not make slot information, or slot type information available. SOM will tell us the size of the object, but not what the bits that are contained in its body represent, or the names of the slots. It is intended that programs wishing to access these instance data sections use a C header file that contains such information. We could have written a C parser that parses C header files to extract this information, but instead we manually add the slots as shown above.

The procedure interpretSlotOptions is specialized by the metaclass SOM to interpret SomOffset: as a byte offset into the instance data section, and Class: as a class describing the contents and size of the slot. We have created metaclasses for C datatypes that provide translation when accessed and set between native Yolambda types and C types. In this case the slot is a Cint, a 32 bit integer.

The procedure interpretClassOptions is specialized by the metaclass SOM to interpret class

options ExternalName: and Proxy: External name specifies the name that is registered in SOM. This class will be available to other applications by this external name. Proxy: if specified #t indicates that the class already exists and that we are creating a local proxy class for it. This causes the existing class information to be interrogated.

We can create a subclass of SomFrob as follows:

```
(defineClass SomFrotz (SomFrob)
  ((baz Class: Cchar))
  ExternalName: "som_frotz"
  Documentation: "son of som_frob")
```

The option Metaclass: is not specified in this example. The MOP generic procedure compute-Metaclass is invoked on the superclasses to determine the metaclass if none is specified. If no superclasses are specified, StandardClass is used. In this case, the creation of this class, as before is governed by the metaclass SOM. Because the class option Proxy: #t was not specified, this class will be exported. The size of the new object will be expanded by the size of a Cchar, and its external name will be som_frotz.

Normal inheritance will work for som_frotz objects as expected. Externally defined methods on som_frob will be inherited by som_frotz as will locally defined methods on SomFrob.

Having defined these classes, we can immediately create new instances of them.

```
(define aSomFrob
  (make SomFrob foo: 42))

(define aSomFrotz
  (make SomFrotz foo: 666 bar: #\X))
```

Before any methods that are already defined on the class som_frob can be invoked, they must be internalized. Although SOM allows method functions to be found and invoked by name, it does not provide a way to enumerate the valid methods of a SOM class, so again this must be done manually.

```
(define (doThing SomFrob|object)
  (SOM-Invoke "doThing"  object))
```

The define creates a native Yolambda generic procedure. We can now do:

```
(doThing aSomFrob)
```

which will result in the equivalent of the C _doThing(aSomFrob);

It is clear that the refraction in this example that cannot be eliminated results from deficiencies in SOM. While there is value in diversity of object systems, if they are to coexist, an adequate MOP is essential.

In the available space we have given an overview of the approach to communicating with a foreign object system alluding to relevant parts of the Yolambda metaobject protocol where appropriate. Although in many ways SOM is designed for exactly the kind of use that we made of it, it still lacks things that are needed to effectively interact dynamically with such a system.

CONCLUSIONS

We studied several attempts to interface disparate object systems. All of these cases had long histories and solutions implemented in C++ or Lisp. Problems with these prior solutions prompted this work.

For an object oriented programming language to be viable in a rich heterogeneous objected oriented environment, the language must provide a rich reflection protocol. It must support runtime metaclasses, and it must separate class from object so that the class system stands on its own.

Existing languages that lack runtime class information, such as C++, are a poor choice for implementing OOOS's, OODB's, or other object oriented services targeted at a heterogeneous environment. New languages, efficient enough for widespread use, but with as much attention paid to richness in reflective support are needed if the current trend toward object oriented operating system and database services is not to have its benefit eclipsed by excessive inter object system translation complexity.

REFERENCES

RL92, Robertson, P. & Laddaga, R.
'Yolambda 1.0 Language description and design considerations'
DOLG working paper 1992.

PR92, Robertson, P.
'On Reflection and Refraction'
DOLG working paper 1992.

SEK89, Keene, S. E.
'Object-Oriented Programming in Common Lisp'
Addison-Wesley 1989.

IBM91, IBM
'OS/2 2.0 Technical Library System Object Model Guide and Reference'
IBM Document Number 10G6309 1991.

BDGKKM, Bobrow, D. G. DeMichiel, L. G.
Gabriel, R. P. Keene, S. E. Kiczales, G. and Moon, D. A.
'Common Lisp Object System Specification'
X3J13 Document 88-002R, 1988.

AP90, Paepcke, A.
'PCLOS: Stress Testing CLOS Experiencing the Metaobject Protocol'
ECOOP/OOPSLA '90 Proceedings.

AG83, Goldberg, A. & Robson, D.
'Smalltalk-80 The Language and its Implementation'
Addison-Wesley 1983.

SYMB88, Symbolics,Inc.
'Joshua Reference Manual'
Symbolics, Inc. 1988