Meta-Object Protocols for C++: The Iguana Approach

Brendan Gowing* **Vinny Cahill**

Distributed Systems Group,
Department of Computer Science,
Trinity College,
Dublin 2,
Ireland.
http://www.dsg.cs.tcd.ie/

Abstract

Meta-Object Protocols (MOPs) are an important aspect of object-oriented reflective programming. A number of C++ extensions have been implemented that include certain reflective features, however none of these provides a fully featured MOP. In this paper, we describe an extended version of C++ called Iguana that allows various features of the C++ language to be reified and their implementations (dynamically) modified. We show how Iguana can be used to write compiled reflective software.

1 Introduction

The Meta-Object Protocol (MOP) of the CLOS programming language [KRB92] is an exemplary model of how to provide fully-functional reflective support in a language. C++ [Str91], which has become a very popular general–purpose programming language, has by default no such reflective or meta-level features[1], though a number of previous extensions support reified method dispatch [KHL+93], object creation and member access [CM93]. In this paper, we present the Iguana programming language which extends C++ with MOP features.

The development of Iguana has been motivated by our research into adaptable operating system software. We have chosen to use reflection as a mechanism for implementing *dynamically* adaptable system components [GC95]. We were, however, faced with the problem of choosing a programming language that supported both reflection and the complexity of operating system implementation (such as machine level representation). We therefore chose to add reflective features to C++ as it could already support system software development.

Essentially, a MOP [KRB92] *specifies* the implementation of a reflective object-model. We consider that a program's meta-level is an implementation of the object model supported by that language. Thus, a MOP specifies which meta-level objects are needed to implement an object model, be it as a consequence of reification or a meta-level declaration. A MOP itself can be said to be instantiated as a "MOP instance" when each of its meta-level objects are instantiated. Significantly, the interfaces exported by each of the meta-objects become the interface of that MOP.

Iguana includes the following features:

- Multiple, fine-grained MOPs: Objects within a single program can use different object models.

*bgowing@dsg.cs.tcd.ie

[1] If you discount the facility to overload a number of the standard operators, such as `->`, `new`, etc.

- Meta-level classes and objects.
- Reification categories: A list of object model features that Iguana can reify. See Appendix A.
- MOP declaration: Syntax for defining MOPs consisting of meta-level objects and reification category declarations.
- MOP selection: The mechanism for associating base-level objects with one or more MOPs.
- Mechanisms for invoking meta-level objects.
- A meta-level class library containing MOP implementations.

Supporting multiple MOPs and multiple MOP instances allows distinct object models to be used. For example, a distributed object could use a distributed object model while its peers continue to use the standard C++ object model. The object models specified by a MOP are implemented by meta-level classes and their instances. By default Iguana does not reify anything. Reification categories provide the opportunity to specifically reify a component of the object model. In this fashion, a compiled program does not have to suffer the performance overheads of a "reify everything" policy.

MOP declaration is the mechanism for specifying which reification categories will be used and which meta-level objects will implement an object model. MOP selection is the mechanism that base-level objects use to bind themselves to an implementation of an object model. Base-level objects can subsequently invoke meta-level objects explicitly through the base-level objects `meta` member, or implicitly through the reifications made to support the object model.

Finally, Iguana has a meta-level class library which contains the meta-level classes needed to implement a number of standard (in Iguana terms) object models.

In this paper, we describe the meta-level features of the Iguana language. In the next section we discuss some related work. Section 3 discusses the reasons why we have introduced the concepts of fine-grained MOPs and explicit reification categories into Iguana. Section 4 presents the syntax of the Iguana extensions to C++. In Section 5, we present an example MOP supporting active objects. Section 6 discusses performance. Section 7 describes the status of the current implementation of Iguana, while section 8 concludes the paper.

2 Related Work

The first language to feature a MOP was CLOS, the Common Lisp Object System [KRB92]. CLOS implements OO programming in a LISP environment. Before CLOS, a number of distinct OO programming extensions had been added to LISP, each with their own features and eccentricities. The nascent CLOS, by using a MOP, was not only able to supply a "greatest denominator" OO system which supported the majority of OO features provided by the existing systems, but was also an open and adaptable implementation which could be modified to provide features that were not part of standard CLOS behaviour. Iguana takes the MOP precedent of CLOS, but builds on it by allowing multiple MOPs to coexist and features selective reification through reification categories.

For other languages, the way their meta-level interactions occur are less formally specified. Often, the meta-level facilities are somewhat limited in comparison to the abilities of the CLOS MOP. This is especially so for compiled languages. For example, OpenC++ Version 1 [CM93] only reifies method dispatch, object creation and object member access. However, despite the lack of dynamic binding between meta- and base-levels, OpenC++ has been successfully used to implement atomic data types [SW95]. Similarly, MeldC [KHL+93] only reifies method dispatch. Iguana supports a larger

set of reifiable language constructs and also allows dynamic meta-level/base-level binding.

The AL-1/D [OIT93, OI94] reflective programming language implements a Multi-Model Reflection Framework (MMRF). MMRF allows the meta-level to be split into modules of meta-level objects called 'models'. Models can be compared to Iguana's fine-grained MOPs but do not support inheritance, as fine a granularity, or feature combination. AL-1/D does not seem to support dynamic modification of it's meta-level, a feature which is available in Iguana through the reification categories.

An alternative form of MOP is the "compile-time" MOP as exhibited by both MPC++ [Ish94] and OpenC++ Version 2 [Chi95]. MPC++ is a "metalevel processing" version of C++ in which the meta-level architecture specifies an abstract C++ compiler. At compile time, the programmer can use the meta-level features of the MPC++ compiler to extend the syntax of the language with new features. Essentially, MPC++ is an open implementation of a C++ compiler, where the compiler itself is reified as a MOP, which can be extended at compile time. However, the code generated by the compiler is not reflective, although reflective extensions could possibly be added using the notational extension feature. The focus of Iguana is very different. OpenC++ Version 2 is a similar system where a compile-time MOP describes the behaviour of the compiler and guides the code generating process.

Iguana aims at generating reflective software instead of being a reflective compiler itself. The Iguana syntax can not be extended, but the MOP part of the syntax is sufficiently general so as to allow new meta-level concepts to be expressed easily. More importantly, Iguana has been designed to explicitly facilitate *dynamically adaptable* objects, i.e., objects whose behaviour can be adapted at runtime.

3 Compiling Reflective Languages

MOPs have typically been developed in the domain of the interpreted language. There is good reason for this. For a program to be interpreted, an interpreter must construct a lot of behavioural information about that program, such as appropriate dispatching functions to use, inheritance hierarchy lookup, etc.[2] This information is not used directly by the program, but is instead meta-level information needed by the interpreter to execute the program.

In reflective programming, however, reflection occurs when the subject of a computation is the actual interpretation of the program; i.e., the reflective computation is computing an aspect of *how* to interpret the program. The reflective program can then *adapt* its own interpretation via modifications of the meta-level information. To support this feature, mechanisms are needed to access and effect the meta-level behavioural information. For this to occur *dynamically*, adaption of the meta-level must be able to occur at run-time as the program executes.

As an interpreter has already constructed a significant amount of meta-level information, extending the interpreter to be reflective only involves adding support for exposing the meta-level information to the base-level program, allowing the program to influence the decision making process of the interpreter and effect its own behaviour through modification of the meta-level information and mechanisms.

In the case of compilers, the meta-level information that is constructed at compile time is rarely maintained beyond the compilation process. True, a certain amount of meta-level information can be found in the debugging data

[2]We use the term *meta-level information* to describe both the tables of data associated with interpretation/compilation and the implicit knowledge maintained by the interpreter/compiler to order its decision making process regarding the behaviour of the base-level program.

that compilers can provide, but this typically only consists of tables of symbols and their appropriate offsets into either code or data section. There is no easy way to use such sparse information to effect the behaviour of the program. Also the behaviour of the object model has been literally hard coded into the program. In this regard, the object model is implicit and can not be altered.

Adding reflection to a compiled language thus entails maintaining the meta-level information beyond the compilation process and also embellishing the generated code with the appropriate links to the meta-level information that controls its behaviour. The obvious solution would be to ensure that the compiler would indiscriminately generate meta-level information for every element of the program's object model. But this presents a significant problem: the program now has the increased execution overhead involved in evaluating the links and meta-level information, and it is also wasteful of both compilation time and storage space, especially in the case where the meta-level information of a program component will never be used.

To solve this problem, Iguana supports reification categories and multiple, fine-grained MOPs. Reification categories attempt to minimise execution overhead by offering the programmer the opportunity to selectively choose which object model elements need to be reified. Thus reification only occurs where it is expressly needed. Multiple fine-grained MOPs, as well as supporting the notion of multiple object models, address the problem of meta-level information bloating the executable image by supporting the implementation of a modular meta-level architecture. Meta-level information is only generated for an object that selects a MOP.

Reification category selection can only occur within a MOP declaration. Thus it is the act of MOP selection by an object which determines what meta-level information will be available to that object and in what ways it will be reified.

3.1 Reification Categories

In order to avoid falling into the "reify everything" trap and its subsequent performance overhead for compiled programs, Iguana uses *reification categories* to indicate to the compiler where reification should occur. Every reflective language will have a set of object model elements which can be reified. These elements correspond to Iguana's reification categories but, unlike the typical reflective language, reification categories are not implicit. In Iguana, there is an explicit process of selection which can only occur within the context of a MOP declaration.

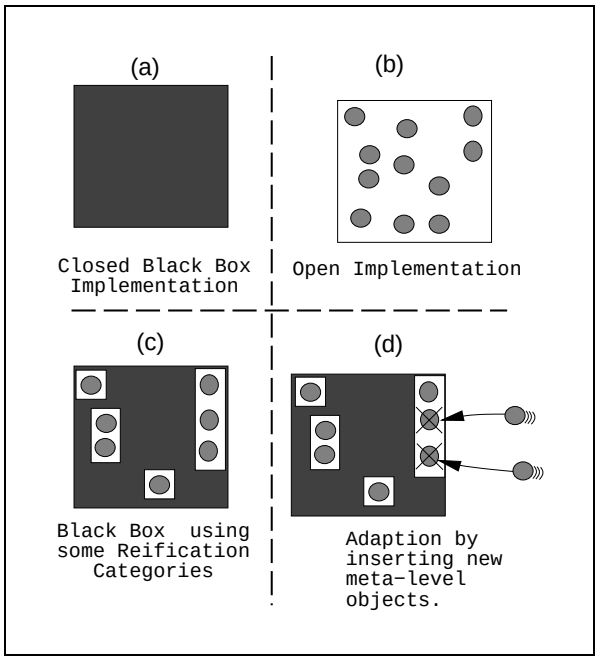

Figure 1: **This example shows (a) a black box approach to object model implementation, (b) an "open" implementation, (c) reification categories exposing parts of the object model, and (d) adaption via dynamically rebinding meta-level objects.**

Whereas the CLOS MOP automatically maintains method information in a language

accessible manner, i.e., methods are always reified, Iguana allows each MOP to specify which reification categories the MOP needs to use. Figure 1 shows the distinction between (a) a black box object model implementation, where no adaption can be performed, (b) the open implementation which has all elements of the object model reified, and (c) the reification category version where meta-level objects are exposed for certain features of the object model. Part (d) of figure 1 shows how adaption can be achieved through dynamic rebinding of meta-level objects.

For each reification category that is used, two actions will be performed by Iguana. First, the actual reification will be performed which involves modifying any affected base-level classes to use a different object model mechanism than the default. Secondly, a meta-level object which implements the new mechanism will be associated (i.e., dynamically linked) with the base-level objects.

As an example, consider the case of the invocation reification category. When the statement `reify Invocation;` appears in a MOP declaration, it will cause the Iguana preprocessor to create an invocation meta-level object and reify invocation within the base-level objects which have selected the MOP. In this particular case, the meta-level object that has been created will be given the default name of `invoke` and be an instance of class `MInvocation`.

3.2 Multiple, Fine-Grained MOPs

Iguana provides the ability to have multiple, fine-grained MOPs. This interesting feature means that an application can have objects that use different object models. Essentially, this means that an object's meta-level implementation can differ significantly from the implementation of another object's meta-level, even though the objects are part of the same application. In an example where active objects and "normal" C++ passive objects coexist in the same application, we describe the "active object" object model as co-existing with the other object models.

An object that subsequently needs to modify its object model (or more correctly, modify its meta-level implementation) can do so knowing that any changes will not affect other object models. We term this feature as being "meta-level locality of change."

In Iguana, MOPs can actually be quite fine-grained. From our research into operating system design, we consider that a fine-grained modular approach to meta-level implementation can more easily facilitate both its implementation and runtime adaption. Thus a programmer can design a MOP which only implements a particular form of invocation and a separate MOP which only implements a particular form of method dispatching. The combination of both of these MOPs can then provide an implementation of invocation and dispatching for base-level objects, but yet the implementation is separated into the two modular components.

4 Iguana Syntax

In this section we describe the extensions made to the syntax of C++ to support reflection. Iguana is implemented as a preprocessor which reads in the Iguana source, digests the meta-level extensions, makes the appropriate meta-level modifications, and then outputs modified C++ code. If little use has been made of the reflective features then there will be little difference between the input and output code. After the preprocessing has completed, Iguana will invoke a C++ compiler to compile the output code into an executable.

4.1 Meta-Level Classes and Objects

Syntactically, there is no difference between meta-level and base-level classes and objects. Meta-level classes and objects are declared using the existing C++ syntax. The features that do distinguish between base- and meta-level are

where an object is declared and what its functionality is. Meta-level classes and objects are usually declared within a MOP declaration and their functionality usually concerns meta-level features, such as method dispatch, which would make their inclusion at the base-level rather meaningless and somewhat impractical.

An example of where a class could be usefully employed by both the meta- and base-levels occurs with general purpose classes such as lists, arrays, etc., where the functionality is somewhat generic.

By convention, the classes in the Iguana Meta-Level Class Library (see Section 4.6) have names beginning with a capital 'M'. This is meant to be an indication to a programmer that instances of these classes would not make very good base-level objects. However, this is by no means a part of the language specification and programmers can choose to name their meta-level classes as they like.

4.2 Reification Categories

As explained above, Iguana is a compiled language which uses reification categories to selectively choose what language elements must be reified. A reification category can only be selected within a MOP declaration (see Section 4.3) so that reification will only occur in accordance with the chosen categories of the MOP.

Syntax for choosing a reification category starts with the `reify` keyword which is followed by the name of a single category that must be reified, for example `Dispatch`. Each reification category has a default meta-level class and instance name. In the dispatch example, the default names are `MDispatch` and `dispatch` respectively. Consequentially, for an object that has selected a MOP which includes a `reify Dispatch;` statement, that object will have a meta-level object called `dispatch` which is an instance of the (meta-level) class `MDispatch`. Also, the code generated for the object by Iguana will use the `dispatch` object to perform method dispatch.

The default meta-level class and instance name can be overridden using two optional parameters to the `reify` command. The first of these is the alternative class name. For example, if a programmer had implemented their own dispatching mechanism in a class called `MyDispatcher`, a subclass of `MDispatch`, then that can be installed as the dispatcher by using `reify Dispatch: MyDispatcher;`.

The second optional parameter similarly allows an alternative choice for the name of the meta-level instance. Instead of using the default `dispatch`, a programmer can select that their own identifier be used; e.g., `reify Dispatch: MyDispatcher dis;`, which just means that method dispatch for objects that have selected the MOP containing this statement will be directed at an object named `dis` instead of the default `dispatch`. Note, however, that the instance name parameter can only be present if it is preceded by the class parameter.

The following are some possible examples of reifying classes:

1. `reify Class;` Classes will be reified and exist at runtime as instance objects of the default class `MClass` and named `mclass`.

2. `reify Class: MyMetaClass;` The classes reified by this statement will be instances of the class `MyMetaClass` but will still be given the default instance name of `mclass`.

3. `reify Class: MyMetaClass myclass;` In this case, the reified classes will be instances of class `MyMetaClass` and name `myclass`.

See Appendix A for a list of Iguana's reification categories.

4.3 MOP Declaration

Syntactically, defining a MOP in Iguana is not unlike defining a C++ class. A MOP declaration starts with the keyword `protocol` and

consists of a name, an optional list of base-MOPs (as opposed to a class's list of base-classes), and a brace-enclosed list of member components separated into sections.

The member components of a MOP definition are either object declarations and reification category declarations or references to other MOP declarations. The former must appear in either the local, shared or global sections, while the latter denote a MOP dependency relationship and must appear in the dependent section. All sections are optional but must appear in the following order:

1. **The dependent section**: In this section, a programmer can list the other MOPs upon which the one being declared is dependent. This will tell Iguana that it must ensure to include the dependent MOPs in any associated objects meta-level. Note, that this is not inheritance, but a horizontal relationship between MOPs; i.e., one MOP cannot function correctly without the presence of another contemporary MOP.

2. **The local section**: Object declarations and reification category declarations in this section will cause object model meta-level objects to be instantiated privately to an associated base-level object.

3. **The shared section**: Object declarations and reification category declarations in this section will be shared by all base-level objects associated with the MOP. This is a similar concept to C++'s static class members.

4. **The global section**: The global section is used for declaring globally shared meta-level objects and reification category generated objects.

Thus a complete MOP definition could look like:

```
protocol SuperMOP;
protocol FundamentalMOP;

protocol MyMOP : SuperMOP
{
  dependent:
    FundamentalMOP;
  local:
    Object myPrivateMetaObject;
    reify Class : MClass
                myPrivateClass;
  shared:
    Object objSharedByMetaLevel;
  global:
    Object objGloballyShared;
};
```

4.4 Protocol Selection

Protocol selection is the act of *associating* one or more MOPs with a base-level object. In Iguana terminology, a base-level object which selects a MOP is said to be associated with the MOP. There are four forms of selection in Iguana: class protocol selection, default protocol selection, instance protocol selection and expression protocol selection. All forms of selection use the "selection operator" (`==>`) to introduce a list of comma separated MOP identifiers.

Class protocol selection is the most common mechanism for selecting MOPs. When originally designing Iguana, we hypothesised that the best person for deciding which MOPs should be used with an object would be the programmer/designer creating the class for the object, as they would know the internal details of the class. During experimentation with a prototype version of Iguana, we found that this was too strict a generalization and that, in fact, it would be useful to (a) set a default set of MOPs which would be associated with all classes in a file (default protocol selection), (b) associate a MOP with an object at the object's instantiation (instance protocol selection), and (c) be able to associate a MOP with an expression (expression protocol selection), a useful facility for some specialised cases.

4.4.1 Class Protocol Selection

Class protocol selection is the association of one or more MOPs with class instances. In a class declaration, the programmer can include a list of MOPs which will form the meta-level for instances of the class. The syntax for this is placed between base-class inheritance (if there are any) and the brace-enclosed list of members. The selection operator (`==>`) is followed by a list of the MOPs to be selected; e.g.:

```
class X : SuperX ==> MetaX, MetaX2
{ ...
};
```

Note that the meta-level association is made between the class instance objects and not the class itself. If classes are reified, any of the MOPs that a class has selected will not form part of the class's meta-level. Instead, the meta-level class of which the base-level will be an instance must select the MOPs for its instances.

4.4.2 Default Protocol Selection

For a common MOP, such as the standard C++ MOP `MetaCpp`, it is useful to have a mechanism for declaring that all classes should select the given MOP. To do this, the default protocol selection construct allows a MOP to be selected by all classes from the declaration to the end of the file. The mechanism includes the facility to turn off the default selection on a per MOP basis.

As an example, the following is used to ensure that all classes select the `MetaCpp` MOP (line 1) while line 6 shows the `MetaCpp` MOP being removed from the default protocol selection list before a class `Y` is defined and subsequently re-selected. In this case, `Y` would use the standard black box implementation of the C++ object model:

```
[ 1] protocol default ==>
[ 2]     MetaCpp;
[ 3]
[ 4] class X { ....};
[ 5]
[ 6] protocol default ==>
[ 7]    --MetaCpp;
[ 8] class Y {...};
[ 9] protocol default ==>
[10]    ++MetaCpp;
```

Note that default MOP selection only provides defaults for class protocol selection. Instance and expression protocol selection are left unaffected. Also note the `++` and `--` operators. These respectively add or remove a MOP from the list of MOPs that will be associated with a base-level object. When these operators are left out of a MOP selection statement, the default behaviour is to add a MOP to the MOP list. It is common practice not to use `++` operator, especially with class protocol selections where the additive behaviour is the most common.

4.4.3 Instance Protocol Selection

Whereas class protocol selection associates a MOP with all instances of a class, instance protocol selection associates a MOP only with a single instance object. The syntax involves following the object declaration with the selection operator (`==>`) and a list of MOPs. For example, given a MOP called `Distributed` which implements support for distributed objects, a distributed `Integer` can be declared as:

```
protocol Distributed;
Integer i ==> Distributed;
```

The actual implementation of this feature involves replacing the declared class of the object with a sub-class which contains the necessary meta-level adjustments. In the above example, the preprocessor would alter the declaration to make it become `Integer__Meta1 i;` and precede it with a definition for the `Integer__Meta1` class. Other instances of class `Integer` will not be affected. New classes with the `__Meta` suffix are only generated for instance protocol selection and expression protocol selection where necessary. If there are

no other `Integer` instances which select the `Distributed` MOP, then `i` in this case will be the only instance of `Integer__Meta1`.

4.4.4 Expression Protocol Selection

We have found that in some cases it is useful to have a part of a MOP used under very specialised circumstances. As an example, consider the following. We were using Iguana to implement a reflective user-space thread package. As a part of the thread packages implementation, we had considered that a very "clean" context switch could be written based on object invocation. By clean we meant that context switching should be both easy to write and also easy to use.

By structuring the thread package to context switch at an object invocation, programmers would be able to perform seamless context switches by simply invoking the method of an object. Thus, to switch context to a thread in another object, a method in that object was simply invoked.

The second benefit related to implementation. It was easier to implement context switching knowing that a context switch always occurs at a method invocation boundary.[3]

Having decided that this was a good way of implementing context switching for both the scheduler and threads releasing control of the CPU, our next problem was to actually implement it in Iguana. By reifying invocation using class or instance protocol selection, all the invocations in an object would also trigger context switches, something we wanted to avoid.

Our solution is to use expression protocol selection, where a MOP to be used with an expression is selected from within that expression.[4] The syntax involves enclosing the expression in parenthesis and inserting a selection operator and a list of MOPs at the end of the expression, but before the closing parenthesis. For example:

```
[ 1] protocol ContextSwitchInvoke
[ 2] {
[ 3] local:
[ 4]   reify Invocation:
[ 5]         MContextSwitch;
[ 6] };
[ 7]
[ 8] class MContextSwitch:
[ 9]         MInvocation
[10] { .... };
[11]
[12] void X::y (void)
[13] { ....
[14]   (obj->method() ==>
[15]       ContextSwitchInvoke);
[16] }
```

Notice how we do not have to include any concepts such as context switching which would be alien to the C++ (and hence Iguana) definition. By simply reifying an existing language construct, in this case method invocation, we can use that to insert the appropriate piece of code which meets our requirement.

So, how does this work? Essentially, the Iguana preprocessor will reify invocation for the expression. This means that the method invocation on `obj` will be transformed into a meta-level invocation to a `send` method. In this case, the `send` method is a member function of the class `MContextSwitch`. Thus the line

```
(obj->method() ==>
    ContextSwitchInvoke);
```

is transformed into a statement such as

```
(meta->invoke->send
    (obj, Object::method, NULL));
```

This mechanism was originally called *statement* protocol selection, but the tighter granularity of expressions offers greater flexibility.

[3] As an aside, context switching on exit from a method can be achieved by reifying `MethodAccess`.

[4] This does not prevent other MOPs which do not reify invocation from coexisting.

4.5 Meta-Level Invocations

Invoking a meta-level object is achieved through the `meta` class member, i.e., in a method, a programmer can invoke the `send` method of an `MInvocation` meta-level object via:

```
meta->invoke->send (...);
```

The address of a meta-level object is also available, so the following is valid code to copy the invocation meta-level object to a local pointer:

```
MInvocation* myInvokerML0 =
    meta->invoke;
```

As `meta` is a pointer, a base-level object is dynamically bound to its meta-level objects. Re-binding to a different meta-level is simply a matter of replacing one or more of the existing meta-level objects. For example, for an object to replace its invocation meta-level object, it simply has to replace it with a new invocation object:

```
MInvocation* tmp = meta->invoke;
meta->invoke =
    new MInvocationType2;
delete tmp;
```

The old invocation meta-level object can only be deleted if it is local, i.e., not shared with other objects. It is also possible for an object to replace its entire meta-level with one from another class, such as:

```
Meta* tmp=meta;
meta = (Meta*) new
      SomeOtherClass::Meta;
delete meta;
```

In this case, a new meta-level will be constructed for the object. This can, however, be a somewhat hazardous process if the reification categories of the meta-level's are different. There is as yet no process of migration in Iguana to provide automatic checking for meta-level compatibility like that provided by the Apertos operating system [Yok93].

4.6 Meta-Level Class Library

The Iguana Meta-Level Class Library is a library of meta-level classes that implement meta-level language related concepts. For example, it contains classes such as `MClass`, `MDispatch`, `MInvocation`, etc. These provide implementations of the default behaviour for MOPs such as `MetaCpp`, the standard C++ MOP.

The Meta-Level Class Library consists of:

- A set of header files for standard MOP declarations (such as `MetaCpp` and `MinCpp` standard C++ MOPs), identifiable through their "`.mop`" file extensions.
- A set of header files for the standard classes which implement the standard MOPs.
- An actual library containing the implementation of the standard meta-level classes, against which Iguana code can be linked.

5 Example: C++ Active Objects

As an example of how Iguana can be used, consider the implementation of active objects in C++. For our example, we consider an active object to be an object which has one or more threads associated with it. In this particular case, we will be creating a thread for each method within an object. This might seem to be extravagant, but remember that we will only be making active objects out of the instances of those classes which actually select the `ActiveDispatcher`, presented below, as one of their MOPs. The beauty of this example is that programmers can write multi-threaded applications using the ActiveDispatcher MOP without having to explicitly make thread and locking calls in the base-level classes.

We begin by writing the actual MOP declaration:

```
[ 1] protocol ActiveDispatcher
[ 2] {
[ 3] local:
[ 4]   reify Dispatch:
[ 5]     MActiveDispatcher;
[ 6]   reify MethodAddress;
[ 7]   reify StateAccess:
[ 8]     MLockableAccessor;
[ 9] };
```

Here, we have declared that message dispatch (line 4), method addresses (line 6) and state access (line 7) should be reified; i.e., at runtime message dispatch should be trapped, the addresses of methods should be maintained, and accessing a component of an object should be trapped. We have then chosen the class `MActiveDispatcher` to implement the dispatching routine (line 5), and class `MLockableAccessor` to implement state access (line 8). Method addresses do not need a class to be specified because we want to use the default `MMethod` class (line 6).

Notice how we have not reified any abstract notions such as threads or locking. In a reflective language, one can typically only reify existing language constructs. As neither Iguana, nor its parent C++, have a language-level thread construct, threads can not be reified. Instead, we must reify the language elements that will allow us to implement what we desire but using a thread class.

The code for class `MActiveDispatcher` includes the `dispatch` method which implements the actual dispatching of method invocations:

```
class MActiveDispatcher:
    MDispatcher
{
  int size;
  Bool* activities;
  Thread** threads;
public:
  MActiveDispatcher ();
  void dispatch (MObject*,
      MMethod*,MActFrame*);
};
```

```
MActiveDispatcher::
    MActiveDispatcher (void)
{
  size = methodAddress->size();
  activities = new Bool [size];
  threads = new Thread* [size];
  for (int i=0; i<size; i++)
  {
    activities[i] = FALSE;
    threads[i] =
      new Thread (&BASE::method);
  }
}
```

The method for implementing dispatch can then be written as follows:

```
void MActiveDispatcher::
  dispatch (MObject* obj,MMethod* m,
    MActFrame* p)
{
  int i=methodAddress->number(m);
  threads[i]->queue
    (new MInvocation(m,p));
}
```

As there are multiple threads executing in a single object, some form of locking must be implemented to ensure that data integrity is maintained. By reifying state access to use the `MLockableAccessor`, we can ensure that state updates maintain integrity.

Finally, for an object to be an active object, the `ActiveDispatcher` can be selected by that object either through its class or at declaration time. For example, the former situation occurs with any instance of class `Server` while the object `client`, an instance of the none-active `Client` class (declaration not shown), has selected the `ActiveDispatcher` in its declaration.

```
class Server ==> ActiveDispatcher
{
  int i;
public:
  Server ();
```

```
  Result& process (...);
};

Server serve;
Client client ==> ActiveDispatcher;
```

6 Performance

There are a number of factors that contribute to the cost of applications compiled with Iguana:

1. The number of indirections required to initiate dynamic binding;
2. The number of meta-level invocations;
3. The cost of computation at the meta-level;
4. The cost of object creation.

The following tests were all conducted on a 33MHz 486-DX PC running the Linux operating system.

6.1 Indirections

On our test platform, the number of indirections used by Iguana causes an extra four move instructions[5] to be used per invocation to obtain a reference to the destination meta-level object. This is a direct consequence of using the dynamically bound `meta` as a reference pointer to an objects meta-level and dynamically bound component references to the actual meta-level objects. For example, the meta-level invocation needed to invoke a meta-level reception object of a base-level object `obj` is:

```
obj->meta->reception->receive (...);
```

An alternative approach would have been to flatten the references by using a non-reference member object for the meta-level and to have separately identifiable component meta-level objects. In such a case, the above statement would become:

```
obj.meta_reception.receive (...);
```

We chose not to implement such a format because the performance gain was minimal and could not be compared to the advantages of (a) having a dynamically bound and adaptable meta-level and (b) having a single encapsulated meta-level for each object which can easily be altered as a whole. See Section 4.5 for an example of point (b) in use. The cost of Iguana indirections per-meta-level invocation is given in the table in Figure 2

6.2 Invocations and Computations

The number of meta-level invocations that must be made to implement a base-level feature relates to the number of reification categories that are used; an extra meta-level invocation must be made for each reification category. For example, the following timings in Figure 2 were gained by comparing a typical C++ object invocation with a reflective equivalent. The first figure shows the cost of a C++ invocation (and function execution). Then, using Iguana, the same invocation was timed using one reification, i.e., `Reception`. This increased the cost of the invocation by half as it added one intermediate meta-level invocation. Using three reifications (namely, `Invocation`, `Reception`, and `Dispatch`) adds the cost of an extra three meta-level invocations on to the cost of the base-level invocation and the subsequent time is approximately four times that of the C++ invocation.[6]

It must be noted that the expense involved with reification categories is only paid by reflective objects because they have selected a

[5] All four of these move instructions were of the more expensive register-memory format as opposed to register-register — an indication of the limited general purpose registers available on the test platform's 486-based architecture.

[6] In some simple cases where only one reification was used, Iguana has actually *reduced* the cost of method invocation by 14%–17% over the equivalent C++. These can be attributed to the flattening of some base-level indirections and virtual function dispatch by the Iguana preprocessor.

Iguana Invocation Timings	
Iguana Indirection	0.78 μsecs.
C++ Invocation	2.064 μsecs.
1 Reification	3.009 μsecs.
3 Reifications	8.535 μsecs.
4 Reifications	21.646 μsecs.

Figure 2: **Reified and non-reified timings for method invocation/dispatch.**

protocol. Invocations between non-reflective C++ objects are not encumbered in *any* way. This contrasts favourably to AL-1/D, which does not support reification categories or fine-grained MOPs, where modifications to support distributed computing caused a 20-fold increase in **local** message communication [OI94].

What these figures do not show is the cost of meta-level computation, i.e., the particular functionality supported by a given meta-level and the overhead that is involved. For example, the performance of a meta-level implementing distributed computing facilities has by definition to be worse than a contemporary non-reflective, non-distributed C++ program, as the former must include the cost of network communication. An example of such computation is shown in our final entry in Figure 2. In this case we used the same three reification categories from the previous test and added `DispatchAccess` as the fourth. The implementation for the dispatch access meta-level object used a naive lookup mechanism which, although simple to implement, demonstrated sub-optimal performance. This increased the cost of the invocation to over ten times that of the original C++. This emphasises the point that meta-level programming can be expensive when care is not taken in its implementation.

6.3 Object Creation

Another performance issue that requires consideration when programming in Iguana is that the creation of reflective objects is more expensive than the creation of their non-reflective counter-parts. This is due to the fact that a reflective object needs to have its meta-level constructed at creation time. In the case where an object uses only shared meta-level objects, the cost is merely in terms of the number of meta-level object references which have to be bound to the shared meta-level objects. However, for non-shared local meta-level objects each must be allocated and constructed as part of the base-level object creation process.

7 Status

We are now in our second implementation of Iguana. Our initial prototype implementation was limited in two ways: first, for ease of implementation, its parser was not designed to parse all of the C++ base language. Secondly, as a prototype we did not implement all of the categories of reification (which also meant that the prototype implementation's Meta-Level Class Library was quite small). Not only has Iguana's syntax changed from the initial implementation, but we have also reorganized and increased the number of reification categories.

As of writing, we are in the process of completing the implementation of the complete Iguana language on Unix platforms using the Cppp C++ front-end parser from Brown University. The new version will include support for all of the C++ base language, code generation for the larger set of reification categories, and a more substantial Meta-Level Class Library.

8 Summary and Conclusion

In this paper, we presented the Iguana programming language. Iguana is an extended version of C++ which includes support for reflective programming.

To prevent incurring unnecessary overhead, Iguana does not implement a "reify everything" policy. Instead, programmers can selectively choose which object model elements

(reification categories) need to be reified to achieve their task at hand. This process of selection occurs within a MOP declaration. For a given MOP declaration, any number of the reification categories can be selected.

Iguana supports multiple MOPs, meaning that there can be many actual MOP declarations, each oriented towards a specific task. Also a MOP can be said to be "instantiated", i.e., each of the meta-level objects in the MOP are instantiated. Iguana also supports the use of multiple MOP instances. In this fashion, different base-level objects can have differing meta-level objects instantiated from distinct MOPs and which implement different meta-level features and reifications.

References

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *10th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299, 1995.

[CM93] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In O.M. Nierstrasz, editor, *7th European Conference for Object-Oriented Programming, ECOOP '93*, Springer-Verlag LNCS 707, pages 482–501, July 1993.

[GC95] B. Gowing and V. Cahill. Making Meta-Object Protocols Practical for Operating Systems. In *4th International Workshop on Object Orientation in Operating Systems*, pages 52–55, 1995.

[Ish94] Y. Ishikawa. Metalevel Architecure for Extended C++. Technical Report TR-94024, Tsukuba Research Center, 1994.

[KHL+93] G.E. Kaiser, W. Hseush, J.C. Lee, S.F. Wu, E. Woo, E. Hilsdale, and S. Meyer. MeldC: A Reflective Object-Oriented Coordination Language. Technical Report CUCS-001-93, Columbia University, NY, January 1993.

[KRB92] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1992.

[NY93] S. Nishio and A. Yonezawa, editors. Lecture Notes in Computer Science 742, Kanazawa, Japan, November 1993. Springer-Verlag.

[OI94] H. Okamura and Y. Ishikawa. Object Location Control Using Metalevel Programming. In M. Tokoro and R. Pareschi, editors, *8th European Conference in Object-Oriented Programming*, LNCS 821, pages 299–319, Bologna, Italy, July 1994. Springer-Verlag.

[OIT93] H. Okamura, Y. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. In Nishio and Yonezawa [NY93], pages 110–127.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2 edition, 1991.

[SW95] R.J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. In *9th European Conference on Object-Oriented Programming (ECOOP)*, pages 168–189, Aarhus, Denmark, August 1995.

[Yok93] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In Nishio and Yonezawa [NY93], pages 145–0.

A Iguana Reification Categories

Iguana Reification Categories			
Name	*Default Class*	*Default Instance*	*Description*
`ActivationFrame`	`MActFrame`	`actFrame`	Reification of method activation frames, activation frame meta-level objects contain method parameters.
`Class`	`MClass`	`mclass`	Reify a class to exist as a meta-level class instance at run-time.
`Creation`	`MCreation`	`mcreate`	Reification of object creation, similar to overloading the `new` operator.
`Deletion`	`MDeletion`	`mdelete`	The opposite to creation and bearing a kinship with `delete` operator.
`Dispatch`	`MDispatch`	`dispatch`	Reification of the receipt of messages and their dispatch to the appropriate methods.
`DispatchAccess`	`MDispatchAccess`	`dispatchAccess`	Reification of the method lookup mechanism.
`Identity`	`MIdentity`	`identity`	An object's identity can be reified as an object.
`Inheritance`	`MInheritance`	`inheritance`	Reify the inheritance mechanism of C++; The `MInheritance` class does not provide features such as evolution or dynamic inheritance. These are left to specific subclasses.
`InheritanceTree`	`MInheritanceTree`	`inheritanceTree`	A reification of an object's inheritance tree. Can provide access to all the super/base-classes of an object.
`Invocation`	`MInvocation`	`invoke`	Method invocations are reified to use a meta-level implementation as opposed to the default C++ object models implementation.
`Method`	`MMethod`	`method`	Methods are reified to appear as objects at run time.
`MethodAccess`	`MMethodAccess`	`methodAccess`	Reification of the entering and exit (access) of a method.
`MethodAddress`	`MMethodAddress`	`methodAddress`	A method address table can be reified as an object. This provides access to an explicit (and more complete) form of the standard C++ virtual function table.
`MethodName`	`MMethodName`	`methodName`	Reification of a table of method symbolic names.
`Object`	`MObject`	`object`	A meta-level reference for base-level objects.

Iguana Reification Categories (continued)			
Name	*Default Class*	*Default Instance*	*Description*
`Reception`	`MReception`	`reception`	The act of receiving a message by a base-level object before it is dispatched can be reified.
`State`	`MState`	`state`	An object's state information can be made available as a distinct object.
`StateAccess`	`MStateAccess`	`stateAccess`	Reification of access to an object's state information.
`StateAddress`	`MStateAddress`	`stateAddress`	A table of addresses to the members of an object's state.
`StateName`	`MStateName`	`stateName`	A table of symbolic names used by objects to reference their state.
`Source`	`MSource`	`source`	Runtime access to source code. Typically useful for debugging.
`Type`	`MType`	`type`	The type of an object is reified as an object. The new draft C++ standard is now proposing its own runtime type system.
`TypeSoft`	`MTypeSoft`	`typeSoft`	Reify a soft typing mechanism where compile time type checking is not performed. This is similar to the Smalltalk/Objective-C type system where messages can be sent to objects even though it is not known if their is a corresponding method to implement the message.