Reflection for Statically Typed Languages

José de Oliveira Guimarães *

Departamento de Computação
UFSCar, São Carlos - SP, Brazil
jose@dc.ufscar.br

Abstract. An object-oriented language that permits changing the behavior of a class or of a single object is said to support computational reflection. Existing reflective facilities in object-oriented languages are either complex, type unsafe, or have a large performance penalty. We propose a simple, easy-to-understand, and statically typed model that captures much of the functionalities of other reflective facilities. It brings the power of reflection to the world of type safe and efficient languages.

1 Introduction

Computational reflection is the activity performed by a computational system when doing computation about its own computation [20]. Reflection has become an important issue in object-oriented programming and is usually realized through metaclasses and metaobjects which has the drawbacks of being difficult to learn [2] [7] [23] and/or causes reasonable performance penalty. This paper presents two new reflective constructs that have much of the functionalities of metaclasses and metaobjects. They can be introduced in many statically typed languages without damaging the type system. The performance is minimally affected and the constructs are easy to implement, learn and use.

This paper focuses on the ability of a program to change its computation. It does not discuss features that merely allow a program to examine its structure, such as features that allow a program to list all methods of a class, to discover the class of an object, to examine the types of the parameters of a method, and so on. These features are easy to implement efficiently and safely. Instead, this paper focuses on features that have not been implemented efficiently and safely before. This includes allowing an object to change its class and changing the implementation of an entire class of objects.

This paper is organized as follows. Section 2 describes existing reflective models of object-oriented languages and some problems with them. Section 3 proposes two new constructs for reflectivity. Some examples of problems that can be solved by these constructs are shown in Section 4. Section 5 describes the implementation of these constructs. Section 6 compares the proposed constructs with existing ones and presents related work.

* This work was partially supported by CNPq, the Brazilian financial agency for scientific projects under process number 200466-94.1

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 440–461, 1998.

2 Existing Reflective Models

According to Ferber [9], reflective facilities of object-oriented languages can be divided in three categories:

1. the metaclass model;
2. the metaobject model;
3. the metacommunication model.

The Metaclass Model was introduced by Smalltalk [13]. In this language, each class is an instance of a metaclass that describes its structure (instance variables and methods it supports). A metaclass is automatically created by each program class and there is a one-to-one correspondence between classes and metaclasses. The programmer has no direct control in the creation of metaclasses.

The ObjVlisp model [7] removes the limitations of Smalltalk metaclasses. It allows the programmer to explicitly create metaclasses. A set of classes can share a single metaclass and the metalinks can be created indefinitely. A metalink is the linking between a class and its metaclass.

In Smalltalk and ObjVlisp, a message sent to an object is redirect to the object class. That is, the search for the appropriate method is made in the object class. Therefore, there is no way to change the methods of a single object or making a specific message interpreter to it.

The Metaobject Model was introduced by Maes [20] and allows specialized method look-up for a single object. Each object has an associated metaobject that holds the information about its implementation and interpretation. Since there is a one-to-one correspondence between objects and metaobjects, a metaobject can be modified to redirect the messages sent to the object, keep statistical information about this specific object, trace the messages sent to it and so on.

When a message is sent to an object with a metaobject, the normal method look-up is not performed. Instead, a message called `MethodCall` is sent to the metaobject taking as real parameter an object containing the original message data (selector and parameters). Then the metaobject can manipulate the selector and real parameters of the original message as data — the message is reified.

The Metacommunication Model permits reifying message sends. When a message is sent, the run-time system creates an object of class `MESSAGE` and initializes it with the message data (selector, receiver, parameters). Then, the message `SEND` is sent to this object that executes the appropriate method look-up algorithm. Subclasses of class `MESSAGE` can be defined and used with a special syntax thus modifying the original meaning of message send.

Problems with Metaclass/Metaobject Models

Metaclasses are the concept of Smalltalk most difficult to learn [23]. Borning and O'Shea [2] have even proposed that they should be eliminated from this language. It is difficult to understand the differences between the relationships created by instantiation, inheritance and metaclasses. It is difficult to realize the

responsibilities of each of these relationships in complex inheritance/metaclass models like Smalltalk and ObjVlisp. For example, what will happen to an instance variable of class `A` if I modify the metaclass of an `A` superclass ? How can a class called `Class` inherit from class `Object` if `Object` is an instance of `Class` ? Should not a superclass be created before its subclasses and a class before its instances ? Metaclasses lead to the "infinite regression problem": a class should have a metaclass that should have a meta-metaclass and so on. In practice, this problem is solved by introducing nonstandard inheritance/metalink relationships (as the one between classes `Object` and `Class`) which damage the learnability of the language.

The three models described above are naturally type unsafe. They let a programmer remove methods from classes/objects, change inheritance links, modify method look-up, and so on. They can cause run-time type errors in most reflective languages with a few exceptions [4] [14] [22] [28] [29] [31]. It seems that there is no easy way to make reflective constructs type safe without removing some of their power. Just as there is no statically typed language that allows all the flexibility of untyped languages.

3 A Statically Typed Reflective Model

To present our proposal, we use a statically-typed language with single inheritance called Green [17]. Although Green separates subtyping from subclassing, in this paper we will consider Green has a type system similar to C++. That is, all subtypes are subclasses — an object of a class can be used whenever a superclass object is expected. Unlike C++, all instance variables are only accessed by message sends to public methods. The type of a variable can be a basic type (`integer`, `boolean`, ...) or a class. If the type is a class, the variable is a pointer to an object of that class, as in most object oriented language. All objects are dynamically allocated.

3.1 The Outline of a Model

Our objective is to create reflective features that can:

1. replace methods of a class and add instance variables to it at run time;
2. redirect messages sent to an object to another object.

1 and 2 play roles of metaclasses and metaobjects, respectively. These features should be statically-typed, simple, easy-to-understand and efficient. To achieve these goals, we eliminate the metalevel while still keeping most of its features. The model should use the language type system and method look-up algorithm. The absence of a metalevel implies that the familiar language mechanisms would be used to support reflectivity, although through the introduction of new constructs. The lack of a metalevel would make the model:

1. statically-typed and as simple as the language's type system;
2. easy-to-understand since the concepts would not be thoroughly new and;
3. efficient since the normal and fast method look-up algorithm would be used.

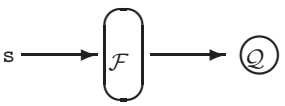

Fig. 1. A shell $\mathcal{F}$ attached to an object $\mathcal{Q}$

The following sections describe two reflective constructs that meet these conditions.

3.2 Dynamic Shell

The first construct is based on the concept of *shell*. A shell is a layer of method/instance variables that is placed between a variable and the object it refers to. Figure 1 shows a shell $\mathcal{F}$ that covers an object $\mathcal{Q}$. A message sent to $\mathcal{Q}$ through variable s will be searched for first in the shell $\mathcal{F}$ and then, if it is not found there, in $\mathcal{Q}$. If a message is sent to `self` inside $\mathcal{Q}$ methods, the search for a method begins at shell $\mathcal{F}$. So, shells allow a class-based language to have some characteristics of languages with delegation.

A shell can be put between an object and the variables that refer to it using *dynamic shell*. Any number of shell objects can be plugged into an object.

The shell object $\mathcal{F}$ of Fig. 1 must have a subset of $\mathcal{Q}$ methods. Since any messages sent through `s` are searched for in $\mathcal{F}$ and then in $\mathcal{Q}$, the composite object $\mathcal{F}$-$\mathcal{Q}$ has the same set of methods as $\mathcal{Q}$. Then, to attach a shell to $\mathcal{Q}$ does not introduce type errors because we are replacing an object by another one with the same set of methods.

Figure 2 defines a dynamic shell class called `BoundedPoint` whose shell objects can be attached to objects of `Point` and its subclasses. The set of methods of `BoundedPoint` must be a subset of the set of methods of `Point`. That means the shell objects created from `BoundedPoint` (as $\mathcal{F}$) have a subset of the set of methods of the objects they are attached to (as $\mathcal{Q}$). The syntactic and semantic rules for dynamic shell assures that each shell object is associated with exactly one object, although a sequence of shell objects may be plugged into an object.

Class-`BoundedPoint` methods can call class-`Point` methods by sending messages to "`object`", which is similar to "`super`" in Smalltalk. As an example, assume that the class of $\mathcal{F}$ of Fig. 1 is `BoundedPoint` and the class of $\mathcal{Q}$ is `Point`. A message send "`object.setX(newX)`" inside $\mathcal{F}$ (class `BoundedPoint`) methods causes the search for method `setX` begin in object $\mathcal{Q}$.

```
shell class BoundedPoint(Point)
  private:
    ...
  public:

      { This is a comment.
        Keyword "proc" starts the declaration of methods. }

    proc setX(newX : integer) : boolean
      begin
      if newX >= 1 and newX <= 80
      then
        return object.setX(newX);
      else
        return false;
      endif
      end
endclass
```

Fig. 2. An example of dynamic shell class

`BoundedPoint` shells can only be attached to objects of classes specified in advance through a compiler option. These classes must be subclasses of `Point` and will be called "the allowed set" of `BoundedPoint`. By default, `Point` will always be in this set. It is not necessary to have the source code of the classes in the allowed set of `BoundedPoint`. It is only necessary to have the source code of `BoundedPoint`. We do not need to have the source code of a class in order to attach a dynamic shell to an object of that class.

Shells can be implemented without the specification of the "allowed set" but that would require the run-time creation of classes, which we want to avoid for performance reasons.

A dynamic shell object of class `BoundedPoint` is attached to an object `p` through the syntax

```
if Reflect.attachShell(p, BoundedPoint.new()) <> 0
then ...
```

`Reflect` is an object whose method `attachShell` attaches a shell to another object. This method returns `0` if there is no error, `1` if there is not enough memory to create the shell object, and `2` if the class of `p` does not belong to the "allowed set" of `BoundedPoint`. Since the class of the object `p` will be known only at run time, the compiler cannot guarantee the correctness of this statement.

A shell is removed from an object by calling method `removeShell`:

```
if Reflect.removeShell(p) <> 0
then ...
```

This method removes the last shell object that was attached to the object pointed to by `p`. It returns `0` if there is no error or `1` if `p` does not have a shell.

The `BoundedPoint` class can redefine the `new` method for creating objects. Then, we can redefine method `new` to take as parameter an object `ObjLimits` that will keep the allowed limits for the point. The shell would have a pointer to this object that could be shared by a set of points.

A class can be declared as reflective to shells using the syntax

```
reflective(shell) class Point
  ...
endclass
```

If a shell is attached to an object of this class, the access to shell instance variables will be faster than if class `Point` were declared as a normal class (without the "`reflective(shell)`" prefix). Details about the implementation of reflective(shell) and normal classes will be given in Sect. 5.

A New Feature for Dynamic Shells

When using metaobjects, each message sent to the object is packed in a message object containing the receiver, message name and real parameters. Then method `MethodCall` of the metaobject is called passing as parameter the message object. So, a single method (`MethodCall`) of the metaobject traces all messages sent to the object. In this way it is easy to send the message through a network or to gather statistic information using metaobjects. A single method of the metaobject must be modified to store data about all messages sent to the object.

To make shells support this functionality, one can declare a method

```
proc interceptAll(
      mi : Method;
      vetArg : array(Object)[]
      ) : Object
```

in the shell class. "`interceptAll`" is a language-Green reserved word and should only be used as method name in shell classes. Method `interceptAll` should always have the signature (interface) shown above. `Method` is a class of the introspective class library of the language. To each program method there is a `Method` object that describes it. The elements of the array `vetArg` are objects of class `Object` and its subclasses. Class `Object` is superclass of every other class or array in the program. `vetArg` stores the real parameters to method `mi`. If a parameter is a value of a basic type (e.g. `integer`) it is packed in an object of a wrapper class (e.g. `INTEGER`). An object of class `INTEGER` just stores an integer number.

When a message `m` is sent to an object $\mathcal{Q}$ with a shell, method `m` of the shell will be executed. If the shell does not have a method `m` but it does have a method `interceptAll`, the message parameters are packed in an array passed as a parameters in a call to shell method `interceptAll`. The first real parameter to this method call will be the object of class `Method` that describes the method `m` of object $\mathcal{Q}$ (the object the shell is attached to). Method `interceptAll` can call method `m` of $\mathcal{Q}$ using method `invoke` of class `Method`:

```
mi.invoke(self, vetArg)
```

Method `interceptAll` allows shells to have all the functionality of metaobjects although with the associate overhead. Class `Method` and method `invoke` were based in similar features of Java Core Reflection [32].

3.3 Dynamic Extension

Dynamic shells change individual objects. *Dynamic extension* does the same for classes. Dynamic extension allows a program to replace the methods of a class and its subclasses by the methods of another class at run time. The syntax for a dynamic extension class is similar to the one for dynamic shell. Like this construct, the keyword "`object`" in a dynamic extension class is used to call methods of the original class.

```
reflective(extension) class Window
  ...
  public:
    proc draw() ...
    ...
endclass

extension class BorderedWindow(Window)
  ...
  public:
    proc draw()
      begin
      self.drawBorder();
        { call Window method }
      object.draw();
      end
endclass
```

Fig. 3. An example of dynamic extension class declaration

An example of dynamic extension is class `BorderedWindow` of Fig. 3. This class can be attached to any class that:

1. is specified through a compiler option and;
2. is subclass of `Window` or `Window` itself.

The classes that obey these requirements compose the "allowed set" of `BorderedWindow`. By default, `Window` belongs to this set.

To attach an extension to class `A`, it must have been declared as

```
reflective(extension) class A
  ...
endclass
```

A class can be also declared as

```
reflective(shell, extension) class A
  ...
endclass
```

with the obvious meaning. If a class `A` is declared as

```
reflective(X) class A
```

then all subclasses must be declared in the same way. It does not mind whether `X` is `shell`, `extension` or both things.

The expression

```
Reflect.attachExtension(Window, BorderedWindow)
```

attaches `BorderedWindow` to `Window` at run time. It returns `0` if there is no error and `1` if there is not enough memory to create some auxiliary structures. If `Window` does not belong to the "allowed set" of `BorderedWindow`, there is a compile error. Any message "`draw`" sent to a `Window` object will now invoke `BorderedWindow` method `draw`. This method draw a border and then call method `draw` of `Window`.

An extension is removed from a class by calling method `removeExtension`:

```
if Reflect.removeExtension(Window) <> 0
then
  ...
endif
```

This method removes the last extension of class `Window`. It returns `0` if there is no error or `1` if no extension is attached to `Window`.

4 Some More Examples

An example of use of dynamic shell is to trace messages sent to an object. Figure 4 shows a dynamic shell class `TracePerson` whose shell objects print a message in the screen each time they receive the message "`set`". To trace object `p`, we should execute the code

```
if Reflect.attachShell(p, TracePerson.new()) <> 0
then
  ...
endif
```

Dynamic extension can be used to confirm pre- and post-conditions for methods of a class. As an example, Fig. 5 shows a class `String` that has a method `get` to return the $\texttt{i}^{th}$ character of the string. The dynamic extension class `SafeString` redefines method `get` to confirm the limits of the parameter `i`,

```
reflective(shell) class Person
  ...
  public:
    proc set( name : String; age : integer ) ...
    proc getName() : String ...
    ...
endclass

shell class TracePerson(Person)
  public:
    proc set( name : String; age : integer )
      begin
      printf("Message set sent to %s\n", object.getName());
      object.set( name, age );
      end
endclass
```

Fig. 4. Tracing using dynamic shell

which is a pre-condition for method `get` of class `String`. `SafeString` should be plugged into `String` during the test phase of the program and removed after that.

5 Implementation

The layout of an object in memory is shown in Fig. 6 and includes slots for its instance variables (IV) and a pointer `mt` to an array of pointers to the methods of the object class. If the object class is `A`, this array will be called the `class-A` method table. In fact, the first element of this array (`mt[0]`) does not point to a method but to an object that contains information about class `A`.

Objects of reflective(shell) classes have a pointer `sv` besides pointer `mt`, as shown in Fig. 7. `sv` points to a memory area that contains the shell instance variables.

We will explain the implementation of the shell constructs in detail now. To do that, it is necessary to understand how the algorithm for method look-up works.

All objects of the same class point to the same method table. Each method in a class is numbered, and a subclass reuses the numbers of its superclass. A method's number is used as an index in the method table. So, if object `p` belongs to class `Point` and the number of method `draw` is 3, the address of `draw` is "`p.mt[3]`".[1]

Message sends to `super` are usually resolved at compile time; i.e. they are statically bound. However if a class is declared as reflective(extension), message

[1] We consider that `p.mt[1]` means the second element of an array whose initial address is given by the pointer `p.mt`

```
reflective(extension) class String
  ...
  public:
       { number of characters }
    proc length() : integer    ...

       { get the i-th element }
    proc get( i : integer ) : char
      begin
      return s[i];
      end
    ...
endclass

extension class SafeString(String)
  public:
    proc get( i : integer ) : char
      begin
      if i >= 0 and i < object.length()
      then
        return object.get(i);
      else
        printf("Error ...");
        exit(1);
      endif
      end
endclass
```

Fig. 5. Conference of pre-conditions using dynamic extension

sends to `super` in its subclasses are made through the class method table. Dynamic extension makes methods of the dynamic extension class replace methods of a class. Therefore, static binding would cause errors. For example, suppose "`super.m()`" results in static binding to method `m` of class `A`. If a dynamic extension is attached to `A` replacing method `m`, "`super.m()`" continues to call the original class-`A` method `m`, when it is supposed to call method `m` of the dynamic extension. As will become clear later, to call methods through the method table solves this problem.

The implementation described next ignores the possibility that

1. a shell is attached to an object that already has a shell;
2. an extension is attached to a class that has an extension;

To explain these cases is outside the scope of this paper.

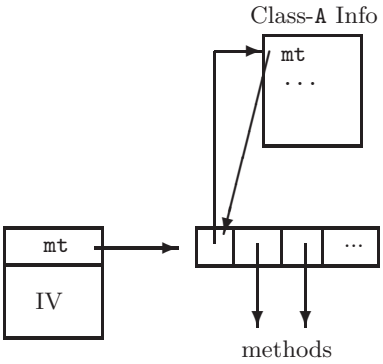

Fig. 6. The layout of a class-A object

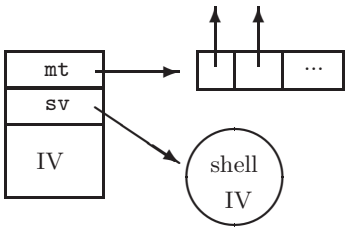

Fig. 7. Object of reflective(shell) class

Dynamic Shell Implementation

An "allowed set" of reflective(shell) classes is associated with a dynamic shell class B. A shell of class B can be attached to any object whose class belong to this set. For each class A belonging to the allowed set of a dynamic shell class B, the compiler

1. makes a copy of class B, renames it B′, and makes B′ inherit from A. Each message send to "object" inside class-B′ methods is changed to a call to the corresponding class-A method. This is made through class-A method table if A is a reflective(extension) class and through a static call to the class-A method otherwise. The compiler enforces that class B does not define any method not defined in class A. Class B′ could be created at runtime but we chose to create it at compile time for performance reasons.
2. inserts in the beginning of each class-B′ method:
    1. an assignment "iv = self.sv" if A is reflective(shell). That makes iv point to the shell instance variables that are pointed to by sv in reflective(shell) classes. See Fig. 7.

2. code to retrieve the address of shell instance variables from a hash table using the object address as key if `A` is not reflective(shell). The address retrieved is also assigned to `iv`. Each class `B′` has its own hash table.

The access to a shell instance variable is made using `iv` through a single indirection. Even if the shell is removed by the shell method, `iv` will continue to point to the shell memory. It will be garbage collected as any other object.

A dynamic shell of class `B` is attached to an object of class `A` through an expression "`Reflect.attachShell(a, B.new())`". Method `attachShell`:

3. tests if `A` belongs to the allowed set of `B`. If it does not, the expression returns `2`. Note that:
   1. the class of object `a` will only be known at run time;
   2. this test would be unnecessary if class `B′` were dynamically created.
4. allocates memory for the class-`B` instance variables. If there is not enough free memory, the expression returns `1`. If class `A` is reflective(shell), `attachShell` makes pointer `sv` of the object points to this memory. See Fig. 7 for the object layout. If class `A` is not reflective(shell), the pair (object, allocated memory address) is inserted into a hash table. The object address is used as key. In the shell methods this table is used to retrieve the address of the shell instance variables.
5. makes pointer `mt` of the object point to the class-`B′` method table created at compile time. Then the object will now use `B′` methods.

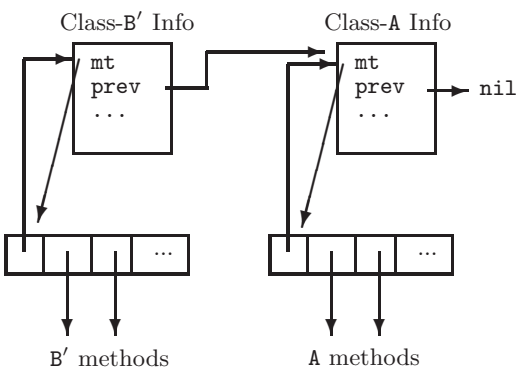

Fig. 8. Relationship between classes `B′` and `A`

Figure 8 shows the relationship between classes `B′` and `A`. The first pointer of class-`B′` method table points to an object with information about this class including a pointer `prev` to class `A` [2] and a pointer `mt` to `B′` method table.

[2] Object class before it was attached to the `B′` shell.

To remove a B′ shell from an object, method `removeShell` of `Reflect`:

1. makes pointer `mt` of the object points to class-`A` method table. This table is found following the links shown in Fig. 8.
2. assigns `nil` (`NULL` in C++) to instance variable `sv` of the object if class `A` is reflective(shell). Otherwise the shell memory is removed from the hash table in which it was inserted. In any case the shell memory will only be deallocated by the garbage collector.

```
proc m'( n : integer )
  var vetArg : array(Object)[];
begin
  { allocates memory for an array of one element }
vetArg = array(Object)[1].new();
  { creates a class-INTEGER object that wraps parameter n }
vetArg[0] = INTEGER.new(n);
  { method_A_m is a pointer to an object describing method m
    of class A }
self.interceptAll( method_A_m, vetArg );
end
```

Fig. 9. Method m′ calling `interceptAll`

We will not describe the implementation of the `interceptAll` feature but we will give some hints about that. The compiler creates and inserts a method m′ in class B′ for each method `m` defined in `A` but not defined in the shell class `B`. This means class `B` has a method for each method of class `A`, either a method like m′ or a method defined in `B` itself. Method m′ has the same arguments as `m` and packs the parameters and information about `m` in objects that are used as arguments in a call to `interceptAll`. The object returned by this method is also returned by m′ if it has return value. This mechanism redirects messages sent to the object to shell methods like m′ that packs the message and calls `interceptAll`. Then the implementation of `interceptAll` does not rise any special problem. A method m′ with an integer parameter is shown in Fig. 9.

Dynamic Extension Implementation

Suppose an extension class `B` is attached to a class `A`. By the definition of dynamic extension, the methods of `B` replace the corresponding methods of `A`. This is achieved by making all pointers that point to a class-`A` method[3] to point to the corresponding class-`B` method. There are pointers to class-`A` methods in method tables of this class and all of its subclasses (which includes classes B′

[3] We consider a class-`A` method any method defined in class `A` or its superclasses.

created by the implementation of dynamic shell). If a class C inherits from class A that defines a method m, class-C method table will point to class-A method m if C does not redefine this method.

If extension B defines instance variables, memory to these variables must be allocated to each object of class A and its subclasses. This is made in a lazy way. Only when a class-B shell method will use shell instance variables the run-time system allocates memory for them.

Let us see in details how dynamic extension is implemented. An extension can only be plugged into classes belonging to its "allowed set" which was specified through a compiler option. For each class A belonging to the allowed set of a dynamic extension B, the compiler

1. makes a copy of class B, renames it B′, and makes B′ inherit from A. Each message send to `object` inside class-B′ methods is changed to a static call to the corresponding class-A method. If dynamic extension implemented a message send to `object` as a call through class-A method table, there could be an infinite loop in a method m of class B if it had a statement like
   ```
   object.m();
   ```
   The entry in class-A method table corresponding to m would point to class-B method m after B is attached to A. Then the above statement would be a call to class-B method m, which is the method in which this statement is. This could result in an infinite loop and certainly does not obey the definition of dynamic extension.
2. inserts a hash table lookup in the beginning of each class-B′ method that uses instance variables.[4] The object that received the message (`self`) is given as key to retrieve the shell memory that is assigned to a variable `sv`. If the lookup fails, the run-time system allocates the shell memory and inserts its address in the hash table. It is worthy noting that each class B′ has its own hash table.

When the compiler finds an expression
```
Reflect.attachExtension(A, B)
```
which attaches a dynamic extension B to class A, it confirms if A belongs to the allowed set of B.

At run time, `attachExtension` makes all pointers that point to a class-A method[5] to point to the corresponding class-B method. That is, if a method table has a pointer to class-A method m, `attachExtension` makes this pointer point to class-B method m (if this method is defined in B). There are pointers to class-A methods in method tables of this class and all of its subclasses, which includes classes B′ created by the implementation of dynamic shell.[6] The method table of a A subclass points to method m of class A if the subclass does not redefine this method.

[4] Of course, class-B instance variables since B′ is a copy of B.
[5] Methods of class A are those defined in A or inherited from superclasses.
[6] Then dynamic extensions can be mixed with dynamic shells in any order without problems.

When a superclass is declared as reflective(extension), message sends to `super` are made through the superclass method table. If an extension is attached to the superclass, its method table will be overwritten by the extension method addresses. Then a message send "`super.m()`" will call the extension method `m` if a method `m` is defined in the extension class.

Accesses to shell instance variables must be made through a pointer got from a hash table. To use a pointer `sv` in the object, as is made in the dynamic shell implementation, does not work. An object of a class `C` that inherits from a class `A` would need two pointers `sv` if an extension were attached to `A` and another to `C`. One pointer `sv` would point to a memory with the instance variables of the extension attached to `A` and the other pointer `sv` would point to a memory with the instance variables of the extension attached to `C`. When an extension is attached to a class `A`, all subclasses (as `C`) are affected because a subclass object is also considered a superclass object. Then to attach an extension to `A` affect class-`C` objects. This problem can be solved by putting several `sv` pointers in the object but that seems too expensive.

To remove an extension `B′` from a class `A`, method `removeExtension` makes all pointers to class-`B′` methods point to the corresponding class-`A` method. Class `B′` hash table is deallocated and the memory for the shell instance variables will be garbage collected.

We have implemented a dynamic shell and dynamic extension using a compiler that translates our language code to C. In the current status of the compiler, a dynamic shell cannot be attached to an object that already has a shell object and an extension cannot be plugged into a class that already has another extension. The compiler, its source code, and some examples of dynamic shells and extensions are available at [16].

6 Discussion and Related Work

One can see the type correctness of dynamic shell considering a shell $\mathcal{F}$ plugged into an object $\mathcal{Q}$ as a composite object $\mathcal{Q}$-$\mathcal{F}$. The method signatures[7] of $\mathcal{Q}$-$\mathcal{F}$ are the union of those of $\mathcal{Q}$ and $\mathcal{F}$. Since the method signatures of $\mathcal{F}$ are a subset of those of $\mathcal{Q}$ (by definition), the set of method signatures of $\mathcal{Q}$-$\mathcal{F}$ and $\mathcal{Q}$ are equal. Then, to attach a shell to an object is to replace an object by another that can respond to the same set of messages, which of course does not introduce type errors.

Dynamic shell and dynamic extension have many of the non-introspective functionalities of metaobjects and metaclasses, respectively. Dynamic shell was not based in other language constructs although it is essentially the trap mechanism of KSL [18] and the metaobject construct of Foote and Johnson [10] introduced in Smalltalk. However, KSL and Smalltalk are untyped languages that allow run-time type errors. Dynamic shell and dynamic extension can be

[7] The signature of a method is its name, types of formal parameters and type of the return value (if any).

supported in a statically-typed language without damaging the language type system.

Predicate classes of Chambers [3] can make an object inherit from some classes dynamically if it satisfies some boolean expressions. The inheritance is removed whenever the boolean expression evaluates to false using the object. Then the methods of an object can be changed at run time by predicate classes which is related to dynamic shell. The main difference is that a dynamic shell is attached/removed by the programmer and the dynamic inheritance of predicate classes is automatically managed by the run-time system.

Related to predicate classes is the *metaCombiner* model [21]. A metaobject called metaCombiner combines *adjustments* that are roughly shells. *Adjustments* are activated (attached to an object) according to events that may depend on the object or program state. The metaCombiner coordinates the way the *adjustments* work together and with the object. The reflective architecture Guaran'a for Java supports a similar feature. A metaobject may coordinate other metaobjects associated with the object.

Aksit et al. [1] defines *Composition Filters* to abstract communications among objects. Each message received or sent by an object can be intercepted by filters that have a metaobject-like behavior. By controlling the object communication with the external world, filters can enforce the cooperative behavior of a set of objects. Ours shells filter messages sent to an object intercepting some of them and there is no way to intercept the messages sent by an object.

A mixin class [8] can refer to its superclass even though it does not inherit from anyone. This class is not intended to create objects but to be composed with other class that plays the role of its superclass. In the composition of a mixin with a class `A`, the references to a superclass in the mixin methods are rebound to class `A`. This is similar to have a subclass that can be plugged to several superclasses. That is why mixin classes are called "abstract subclasses".

The language Agora [27] supports dynamic application of mixins. A class can be dynamically created by combining an existing class with a mixin. This is very similar to dynamic extension. A class `B` defined as a dynamic extension class can send messages (through the keyword "`object`") to the class it will be attached to. If `B` is attached to `A`, any messages sent through "`object`" in `B` methods will invoke class-`A` methods.

In fact, dynamic extension was based in the statically-typed mixin extension to Modula-3 proposed by Bracha and Cook [8].

Feature-oriented programming [25] is much like a compile-time version of dynamic extension. A feature is like a class that can be combined with other feature through a class-like construction called *lifter*. This is similar to attach an extension `B` into a class `A` using a class `C` to define how `B` methods will call `A` methods. We believe that to create a compile-time version of dynamic extension emulating feature-oriented programming would require minor modifications in the language syntax and in the compiler.

Gil and Lorenz [12] proposed *environmental acquisition* to allow an object to behave accordingly to the composite object in which it is connected. So, a

`Door` object could acquire the `getColor()` method from the `Car` object that contains it. Environmental acquisition resembles dynamic shells since methods of individual objects can be replaced by others. The difference is that, in environmental acquisition, the object methods are automatically changed according to the surrounding object whereas one must explicitly attach a shell to an object in order to change its methods.

Contexts [26] are similar to shells. When a context is attached to an object, some methods of the context can replace the object methods. Although shells and contexts have some similarities, we believe shells have more functionalities of metaobjects than contexts:

1. by our knowledge, a context method can only replace a method of a specific class, preventing it to be used with subclasses;
2. the attachment of several contexts to an object is not addressed;
3. no feature similar to method `interceptAll` is defined.

No performance evaluation of contexts is discussed in [26]. Then, we cannot compare the performance of shells and contexts.

The old version of Open C++ [4] [5] is a reflective C++ that supports metaobjects. Its goal is to allow programmers extend Open C++ with facilities that cannot be easily added to C++ as support to distributed and parallel programming. In Open C++, a message sent to an object of a reflective class causes the execution of method `Meta_MethodCall` of the object metaobject, whichever is the message selector. A message send using a metaobject is 10 times slower than a normal one if no optimization is done.

A new version of Open C++ [6] uses metaobjects to control the compilation of classes. The metaobjects exist only at compile time and are not directly related to our proposals.

There are some Java extensions that support metaobjects [14] [22] [31] and a feature similar to dynamic extension. In all these meta architectures a single method (like `Meta_MethodCall` of Open C++) traces all message sends to an object with an attached metaobject, which results in poor performance.

In the old version of Open C++ the call to a reflective method is ten times slower than a call to a normal method if no optimization is done. In MetaJava a call to a method of an object with a metaobject takes 170 μs according to tests made by Golm [14]. A normal call takes 6.1 μs, which gives a ratio of 28 between the two options. Golm used a method without parameters and a metaobject that implemented the default behavior (to call the object method). In our implementation of dynamic shell there is no overhead in the call to a shell method or to any method of an object attached to a shell.

By defining method `interceptAll`, to shells are given the full functionality of metaobjects. Then, it is worthwhile to compare the performance of metaobjects with that of shells with `interceptAll`, which is made in Figures 10 and 11. We use a class `Store` that stores a class-`B` object with methods to store (`put`) and retrieve (`get`) the object. There is also an empty method `m` without parameters. Figures 10 and 11 give the ratio `tshell/tnormal` where `tshell` and `tnormal` are the execution time of a message send using a shell with `interceptAll` and

Fig. 10. Shell performance in a Pentium

	Optimiz.	Non-optimiz.
`a.m()`	2.7	4.2
`b = a.get()`	2.5	3.4
`a.put(b)`	2.7	9.2

Fig. 11. Shell performance in a Sparc 10

	Optimiz.	Non-optimiz.
`a.m()`	3.3	4.6
`b = a.get()`	2.4	3.5
`a.put(b)`	3.0	11.6

normal message send, respectively. Method `interceptAll` simply delegates the message send to the object, which is the default behavior. Figures 10 and 11 show the results of the measurements made in a Pentium and a Sparc 10.

In Fig. 10, a call "`a.m()`" through a shell with `interceptAll` takes 4.2 times the execution time of the normal call. See column "Non-optimiz.". A simple optimization in method `interceptAll` (column "Optimiz.") reduces this overhead to 2.7. This optimization is to allocate array `vetArg` (`interceptAll` real parameter) statically and replace the call "`mi.invoke(self, vetArg)`" by a switch statement with a case label for each method that can be called.

Performance measurements of metaobject is not available for most languages. Albeit that, we can say shells with `interceptAll` are faster than the implementation of metaobjects with available performance data.

Templ [28] introduced reflection in the Oberon-2 language. Each procedure has an associated metaprocedure that knows how to call the procedure. By acting on the metaprocedure, the program can change the procedure behavior. For example, the metaprocedure can call other procedure instead of the one it is associated with. If a metaprocedure could be associated to a method, Oberon-2 would have a feature similar to dynamic extension. The metaprocedure would control a class method in the same way dynamic extension does.

7 Conclusion

The reflective model described in this paper does not really have a "meta level" — it uses the method look-up algorithm and type system of the language. As a result, dynamic shell and dynamic extension are statically typed and easy to understand. They use a syntax very similar to normal classes, do not suffer the "infinite regression problem", and the performance penalty caused by them is minimal.

The reflective features described here do not require any modification in objects of non-reflective (normal) classes. Objects of reflective(shell) classes need one more pointer per object.

Message sends to objects or shell objects are not slowed down by dynamic shell or dynamic extension. In particular, message sends to shells are as fast as message sends to objects. Performance degradation only occurs in two cases:

1. an assignment "`iv = self.sv`" or a hash table lookup is inserted at the beginning of each shell/extension method;
2. when the superclass is reflective(extension) message sends to `super` are implemented by indexing the method table of the superclass. So, the overhead corresponds to an array indirection plus a function call through a pointer.

However, that is not too bad. In most cases shells will be plugged into objects of reflective(shell) classes since their use will be *planned*. In this case just an assignment will be inserted in each shell method that uses shell instance variables. If the object class is not reflective(shell), there will be an extra hash table lookup in each shell method. This lookup always exist when an extension method accesses an extension instance variable.

This hash table lookup is expensive for a statically-typed language but difficult to avoid since shell and extension instance variables must be added to some structure that cannot be the object itself. This inefficience is due to the problem itself and not to the definition of shells and extensions. However, if we prohibit dynamic shells and dynamic extensions to act on the same classes, the shell instance variables can be allocated in the method table the object points to. Each object attached to a shell would have its own method table. If few shell objects are created, this scheme may be more space efficient than to add an instance variable `sv` in each object of reflective(shell) classes. An optimized compiler could analyze the source code and use this implementation instead of the one described in this paper if dynamic extensions were not used with the same classes as dynamic shells.

Message sends to `super` by indexing the method table is 24% slower than a static call.[8] This is less expensive than a normal message send although slower than the normal implementation, which calls the method directly. However only a few percent of all classes are reflective(extension) thus making the overhead of calling `super` using a table very small.

In the reflective architecture Guaran'a [22] for Java and languages as Reflective Java [31], MetaJava [14], and the old version of Open C++ [4] [5], the method `Meta_MethodCall` of the metaobject is called whenever the corresponding object receives a message. The message is packed as an object passed as a parameter to `Meta_MethodCall` that then may call methods of the original receiver. There are special constructs to handle the parameters of a message, which are not necessary in order to use dynamic shell and dynamic extension.[9] When a message is sent to an object and intercepted by a metaobject, the most

[8] It was used a empty method without parameters in a Sparc 10.

[9] We are not considering method `interceptAll`.

common case is that the metaobject delegates the message to the object without doing any computation. Then the message is packed when passed to the metaobject and again unpacked when delegated to the object, resulting in two wasteful operations. With dynamic shells, the object method is called directly in this case at the speed of a normal message send.

Although dynamic shells and extensions were implemented in Green [17] they do not need any particular feature of this language. In fact, we are adding shells to Java [30] with a implementation different from the one described in this paper.

Dynamic shell and dynamic extensions are among the five constructs proposed by Guimarães in [15]. The others are *class view*, *adapters* and *class extension*. All of these constructs share the idea of *shell*. A shell can be attached to a single object using dynamic shell and to a whole class using dynamic extension.

Adapters change the type of an object through a shell and are used to glue otherwise type-incompatible classes. They play a role similar to the Adapter pattern [11]. *Class extension* is a kind of mixin that can be applied to classes without editing the classes themselves. It is based on mixins and extensions of Ossher and Harrison [24]. *Class view* solves the problem of the misinterpretation of a class semantics in the code of other classes. Shells are used to correct the semantic of the objects. Using *class view*, the programmer can restrict the use of shells to specific regions[10] of the program. This is similar to use metaobjects to specific parts of the code.

The dynamic extension construct defined in this paper is simpler than the one proposed in [15]. This one allows a kind of parameterized dynamic extension. The dynamic extension class can use as type for variables and method parameters the name of the class in which the dynamic extension will be attached to.

Acknowledgments. We thank Ralph Johnson for many helpful comments, Brian Foote for suggesting some bibliography to this article and Mamdouh Ibrahim by responding some questions about the KSL language and sending us some papers.

References

1. Aksit, Mehmetl Wakita, Ken; Bosch, Jan; Bergmans, Lodewijk, and Yonezawa, Akinori. Abstracting Object Interactions Using Composition Filters. *Proceedings of ECOOP'93. Workshop on Object-Based Distributed Programming.* Lecture Notes in Computer Science No. 791, 1993. 455
2. Borning A. and O'Shea, T. DeltaTalk: An Empirically and Aesthetical Motivated Simplification of the Smalltalk-80 Language. *Proceedings of ECOOP 88. Lecture Notes in Computer Science* No. 322. 440, 441
3. Chambers, Craig. Predicate Classes. *Proceedings of ECOOP'93. Lecture Notes in Computer Science* No. 707. 455
4. Chiba, Shigeru and Masuda, Takashi. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceeding of ECOOP'93. Lecture Notes in Computer Science* No. 707, 1993. 442, 456, 458

[10] Code composed by methods, procedures, and functions.

5. Chiba, S. Open C++ Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, Tokyo, Japan, 1993. 456, 458
6. Chiba, S. A Metaobject Protocol for C++. *SIGPLAN Notices*, Vol. 30, No. 10, October 1995, pg. 285-299, OOPSLA'95. 456
7. Cointe, Pierre. Metaclasses are First Class : the ObjVlisp Model. *SIGPLAN Notices*, Vol. 22, No. 12, December 1987. OOPSLA 87. 440, 441
8. Cook, W. and Bracha, G. Mixin-based Inheritance. *SIGPLAN Notices*, Vol. 25, No. 10, October 1990, OOPSLA 90. 455
9. Ferber, Jacques. Computational Reflection in Class Based Object Oriented Languages. *SIGPLAN Notices*, Vol. 24, No. 10, October 1989. OOPSLA 89. 441
10. Foote, Brian and Johnson, Ralph. Reflective Facilities in Smalltalk-80. *SIGPLAN Notices*, Vol. 24, No. 10, October 1989. OOPSLA 89. 454
11. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994. 459
12. Gil, Joseph and Lorenz, David. Environmental Acquisition: A new Inheritance-Like Abstraction Mechanism. *SIGPLAN Notices*, Vol. 31, No. 10, October 1996, OOPSLA'96. 455
13. Goldberg, Adele and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. 441
14. Golm, M. Design and Implementation of a Meta Architecture for Java. Diplomarbeit im Fach Informatik, Friedrich-Alexander Uniersität, Erlangen-Nürnberg, Jan 1997. 442, 456, 458
15. Guimarães, José de Oliveira. *Filtros para Objetos*. PhD thesis, 1996. An English version of the thesis is also available with the title *Shells to Objects*. 459
16. Guimarães, José de Oliveira. Shells: The Green Metaobjects. http://www.dc.ufscar.br/~jose/shell.html. 454
17. Guimarães, José de Oliveira. The Green Language. http://www.dc.ufscar.br/~jose/green.html. 442, 459
18. Ibrahim, Mamdouh and Bejcek, W. and Cummins, F. Instance Specialization without Delegation. *Journal of Object-Oriented Programming*, June 1991. 454
19. Ibrahim, Mamdouh. Reflection in Object-Oriented Programming. *International Journal on Artificial Intelligence Tools*, Vol. 1, No. 1, 1992.
20. Maes, Pattie. Concepts and Experiments in Computational Reflection. *SIGPLAN Notices*, Vol. 22, No. 12, December 1987. OOPSLA 87. 440, 441
21. Mezini, Mira. Dynamic Object Evolution without Name Collisions. *Proceedings of ECOOP'97. Lecture Notes in Computer Science* No. 1241, 1997. 455
22. Oliva, Alexandre. The Reflexive Architecture of Guaran'a. http://www.dcc.unicamp.br/~oliva/guarana.ps.gz. 442, 456, 458
23. O'Shea, Tim. Panel: The Learnability of Object-Oriented Programming Systems. *SIGPLAN Notices*, Vol. 21, No. 11, November 1986, OOPSLA 86. 440, 441
24. Ossher, Harold and Harrison, Willian. Combination of Inheritance Hierarchies. *SIGPLAN Notices*, Vol. 27, No. 10, October 1992, OOPSLA'92. 459
25. Prehofer, Christian. Feature-Oriented Programming: A Fresh Look at Objects. *Proceedings of ECOOP'97. Lecture Notes in Computer Science* No. 1241, 1997. 455
26. Seiter, Linda; Palsberg, Jeans, and Lieberherr, Karl. Evolution of Object Behavior using Context Relations. Accepted to publication in *IEEE Transactions on Software Engineering*. Available at http://www.ccs.neu.edu/research/demeter/biblio/context.html. 456

27. Steyaert, Patrick at al. Nested Mixin-Methods in Agora. *Proceedings of ECOOP'93. Lecture Notes in Computer Science* No. 707, 1993. 455
28. Templ, Joseph. Reflection in Oberon. In *ECOOP'92 Workshop on Object-Oriented Reflection and Metalevel Architectures*. 442, 457
29. Templ, Joseph. Metaprogramming in Oberon. Swiss Federal Institute of Technology, Zurich, 1994. 442
30. Tomioka, Elisa; Guimarães, José de Oliveira; Prado, Antônio Francisco do. R-Java: uma Extensão de Java com Metaobjetos. *In English: R-Java, a Java Extension with Metaobjects*. Federal University of São Carlos, SP, Brazil. Unpublished report, 1998. 459
31. Wu, Z. and Schwiderski, S. Reflective Java: Making Java Even More Flexible. FTP: Architecture Projects Management Limited (apm@ansa.co.uk), Cambridge, UK, 1997. 442, 456, 458
32. Java Core Reflection: API and Specification. Java Soft, Mountain View, CA, USA, October 1996. 446