Reifying Inheritance in a Reflective Language

Stéphane Ducasse
Université de Nice – Sophia Antipolis
Laboratoire I3S, CNRS-URA 1376
email:ducasse@unice.fr

Abstract

The inheritance mechanism is one of the most important concepts of object-oriented programming. The idea presented in this paper is to investigate the inheritance mechanism as a "relational information" between classes and to express it in a distinct way from class structure and functionalities. Reifying inheritance by means of links gives FLO a complete reflective definition only based on two concepts: objects and dependencies. Inheritance is no longer a global built-in language mechanism, but becomes described by the language itself: inheritance is managed as other dependencies. FLO is self described, in the sense that inheritance semantics is expressed in terms of class methods. We illustrate this approach by re-implementing the inheritance mechanism of the ObjVlisp model.

Keywords: Reflective Language, ObjVlisp Model, Inheritance Reification, Inter-Object Dependencies, Behavioral MetaObjects, Handling Message

1 Introduction

Nowadays, the inheritance mechanism is one of the most important concepts of object-oriented programming and is, according to Wegner[35], intrinsic to the definition of *object-oriented*. The literature has innumerable works on the semantics of the inheritance mechanism[4, 5, 9], on its use [21], on multiple inheritance problems [6, 16, 17], on new models of inheritance [2, 24], and on alternative solutions [27, 34]. We consider inheritance as a mechanism for incremental programming. The idea presented in this paper is to investigate the inheritance mechanism as a "relational information" between classes and to express it in a distinct way from class structure and functionalities.

In this sense, we reject the inheritance mechanism as a built-in process. On the contrary we propose a reification of such a mechanism by means of inter-object dependencies. This leads to opening up our language, so the work presented is in the same line of [28, 8, 18, 23, 30]. Our language, named FLO[1], is an open object-oriented programming language in the sense of CLOS's MOP[25]. It extends the object-oriented paradigms by integrating dependency management. FLO is based on reification of dependencies (links), control of message passing (thanks to behavioral metaobjects[32]), and an open protocol [15, 13, 14]. Reifying inheritance by means of links gives FLO a complete reflective definition which is only based on two concepts: objects and dependencies. Inheritance is no longer a built-in language mechanism any more, but becomes described by the language itself: inheritance is managed as other dependencies. FLO is self describing, in the sense that inheritance semantics is

[1] standing for "First class Links between Objects"

expressed in terms of class methods. Reintroducing inheritance into an object model is related to other work, such as the re-introduction of delegation in a prototype based language [32].

We illustrate this approach by re-implementing the inheritance mechanism of the ObjVlisp model[2]. Throughout this paper, we highlight the implementation of the inheritance reification from an implementor point of view. It is important to notice that a final user of FLO does not take into account such inheritance aspects.

We first briefly present the major features of the FLO language - we only describe the necessary features of the link management for inheritance reification. Then we recall the main characteristics of the ObjVlisp model highlighting its inheritance mechanism and the structural aspects implied by this mechanism. In section four, we explain the FLO model and how inheritance is reified into links. Section five presents some bootstrapping issues of the FLO kernel. In section six, we introduce a first extension of the inheritance mechanism. Finally, we review related works and conclude.

2 A presentation of FLO and its standard message handling control

Programming with FLO is slightly different that with others OO languages. Indeed with FLO, a programmer defines intrinsic object behaviors by means of classes, and the interactions between instances of these classes by means of links [14]. So after defining such links (by specifying which methods have to be controlled), she/he declares the instances involved in the links, without subclassing instance classes (contrary to the MVC[26] model, no subclass definitions are necessary). Thus FLO automatically maintains the consistency of the link graph by controlling the messages sent to the linked instances [14].

Using a simple example, we will describe the FLO language, i.e. the nature of links, the way they are defined, and the process of link maintenance. We then justify the use of metaobjects and describe the standard control message handling. A presentation of this aspect of the FLO language is to found in [14].

2.1 A FLO link

We use FLO links to ensure communication between abstraction (application) and presentation (view) objects in the same philosophy as the PAC model [10, 12]. This emphasizes a clean separation of user interfaces from applications. In particular, no communication support is coded into the view or application objects: they ignore each other.

We present here a link taken from the FLO's implementation of tic-tac-toe game (TTT[3]). The idea and benefits of such a link based implementation is detailed in [22]. As shown in the figure 1, the main idea is that links ensure communication and consistency between the two views of the TTT's players and the shared abstraction (TTT's core). In such an implementation different links were defined, for example, between abstract and view cells (<Cell-to-Cell> on the Fig. 1), board and TTT game. We only give the <Between-Board-and-Tic> link definition here.

So, on the one hand, the TTT game is represented by the <Tic>[4] class. This class defines some methods, such as next-turn, reset, win, next-player, player-can-play?, game-finished? and current-player. <Tic> class is autonomous, i.e. it suffices

[2] FLO has been implemented with different object-oriented languages: CLOS[15], STKLOS[19, 14], and in particular with a scheme-based implementation of the ObjVlisp model.

[3] We will use this acronym in the rest of this paper.

[4] In this paper, we chose to indicate classes by the following convention : < class-name >.

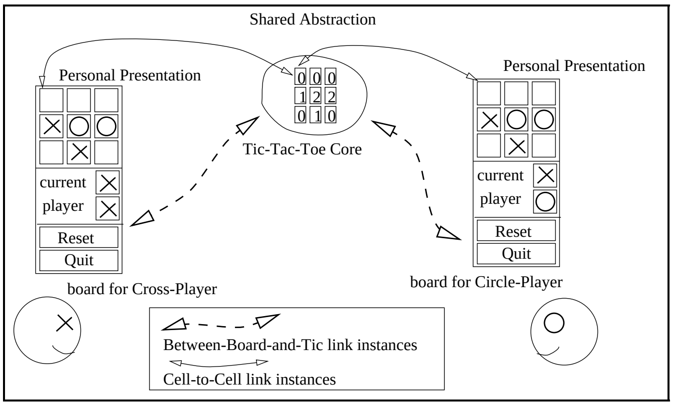

Figure 1: Tic-Tac-Toe architecture: a shared abstraction, two different presentations and some links

to play tic-tac-toe with no graphical representation. On the other hand, the personal views of the two players are represented by the <Board> class. This class makes any references to the tic-tac-toe abstraction and it is responsible for the representation of the game and the player interaction. Thus methods such as reset-game, current-player, and cells-policy are defined. Moreover, by clicking on the button named "Reset", each player can reset the game. However, there are different choices for this reset of the game; for example, one can imagine that only the player who has the turn can reset the game or that the game can only be reset at the end of play. A link can modelize these different choices. Here, in addition to the management of the consistency between the core and the game board, the link <Between-Board-and-Tic> ensures that the game will not be reset before the end of a game. Others choices might be implemented in the same way without modifying classes.

```
1 (deflink <Between-Board-and-Tic>
2  :inherit ´((<One-Entry-Firing-Link>))
3  :behavior  ´((( reset (:abstract)) implies ( reset-game (:graphic)))
4               (( reset-game (:graphic)) implies  ( reset (:abstract)))
5               (( reset-game (:graphic)) permitted-if ( game-end? (:abstract)))))

7 (define Crossboard-to-coretic (send <Between-Board-to-Tic> ´new :abstract Core-Tic :graphic CRossBoard))
8 (define Circleboard-to-coretic (send  <Between-Board-to-Tic> ´new :abstract Core-Tic :graphic CIrcleBoard))
```

Line 3 shows that, when the message reset is called on the object designated by the variable[5] *:abstract (TTT's core)*, the message reset-game must be sent to the object designated by the variable *:graphic (a game board)*. The **implies** operator associates a *compensating message* to a method so that, after applying a specified method, the system automatically performs the associated *compensating message* (reset-game :graphic). In the same way, line 4 expresses that, when the message reset-game is called on the graphical object, the message reset must be sent to the abstraction. It is important to notice that this flow of messages will not loop according to inherit specification line 2. Link cycle detection or specific behaviors are out of the scope of this paper, see [13] for more details. Line 5 expresses that the message reset-game is only permitted when the game is ended. Such a kind of condition is called a *guard*.

Once an instance of <Tic> *(*core-tic*)* and two instances of <Board> *(*crossboard *and* circleboard*)* are created, we create two link instances between these instances lines 7 and 8. Now each time, a user click on the "Reset" button, both views and

[5] These variables look like Lisp keywords because we want the user to keep in mind that such variables refer to objects which are associated with these initarg keywords at creation time.

core are reset. We chose this simple example to clearly present the intrinsic aspects of FLO's links, a curious reader may refer to [13, 14].

2.2 The standard message passing control by means of metaobjects

Amongst the multiple OO meta-level proposals [1, 8, 3, 25, 7, 11, 31], the metaobjects as defined by Maes (in [28]) are well adapted to implementing the FLO dependency management because they allow abstraction of the computation of an object. *"The metaobject holds information about the implementation and interpretation of the object... It is possible to create abstraction of the behavior of an object (i.e. ready-made metaobjects), and to temporarily attach such a special behavior to an object."* [28].

The default proposed behavior of dependency consistency management involves checking *guards* and automatically performing *compensating messages*. This implies controlling the message passing algorithm. In FLO, this control is achieved by associating with each linked object a metaobject (in the sense of Maes's metaobjects [28]). In a previous paper [14], we have justified this introduction of metaobjects in a class-based language [18] and outlined the benefits of this approach. In particular, this gives FLO the following three properties:

* first, separation of object and dependency management functionalities is clear; an object exclusively represents information about the domain entity which it represents,
* second, abstraction of the message handling allows the implementation of the consistency maintenance mechanism, and
* third, as links tie instances, a metaobject approach allows FLO to control only necessary objects instead of penalizing the entire system.

In FLO, classes are used for *structural* description (definition of instance structure and a set of applicable operations) and metaobjects for *computational* description (control of instance behaviors). In FLO, the metaobject of an object is called its *controller*. And FLO's controllers can also be shared between objects.

2.3 Standard message passing control in FLO

To embed metaobjects, we modify the ObjVlisp kernel: each object must have a slot **meta** and the **send** primitive must be modified[6]. Moreover, as classes are objects, they can have metaobjects. This point is really important for the FLO inheritance management, as we will explain in the subsequent sections.

In order to manage inheritance with links, there is no longer a lookup primitive as in the original ObjVlisp implementation[8], a lookup method is defined on the <Class> class. This change introduces possible infinite regression of the lookup phase: sending a message implies to send a lookup message, that implies to send a lookup message, and so on.

Therefore, the below send definition mixes two needs: (a) only messages for linked objects are controlled, and (b) handling the infinite regression of the lookup phase is implemented without interfering with link management. The infinite regression of the lookup phase is stopped (lines 3 and 4) by testing the only fact that we are sure of: the <Class> class has the lookup method in its method dictionary. In such a call, we extract the method applying directly the primitive get-method (line 3) else the method is looked up in the class of the receiver (line 4). Thus, when

[6] In order to avoid an infinite loop, it is important to notice accessing **meta** and **class** slots must not use message passing.

the object receiver is not bound with a metaobject (line 6) the message is handled[7] normally (line 7). If the receiver has a controller, the message is handled by its controller through the call to the method HandleMsg (line 8).

```
1  (define send (lambda (recvr selector . args)
2    (let ((m (if (and (equal? selector 'lookup) (equal? recvr <class>))
3                 (get-method class 'lookup)           ;; base case
4                 (send (class-of obj) 'lookup         ;; the method is looked up
                                 selector recvr)))    ;; in the receiver class
5           (a-meta ( meta recvr)))
6      (if (not-meta? a-meta)                          ;; if there is no meta
7           (apply-method m recvr args)                ;; the message is nor-
mally handled
8           (send a-meta 'HandleMsg
                m selector cons recvr args)))))       ;; the message is controlled
```

What exactly does the HandleMsg method ? When a linked object receives a message, the system sends a HandleMsg to its controller: all the *guards* defined for this message in the different dependencies concerning this object are checked. If each *guard* is true, the invoked method is applied, otherwise the message is not performed. After applying the method, all the *compensating messages* are called[8]. And the result of the invoked method is returned.

An example of such a message passing control. In the context of the previous example, figure 2 illustrates the shift to the meta level in the message passing control. Let us suppose that one sends the message (send CRossBoard 'reset-game) (noted with the number 1 in Fig. 2). As CRossBoard is involved in a dependency with Core-Tic, this message is converted into a message (noted with the number 2) to the controller of CRossBoard: CRB. This controller has no controller, so this message (2) is applied normally: CRB checks the *guards* (here it checks if the game is ended), applies the reset-game method and fires[9] the link. This leads to sending the reset message to Core-Tic (noted with the number 3). This last message is converted to a HandleMsg message (noted with the number 4) to the controller of Core-Tic which checks the *guards* (absent here for this message), applies the reset method, and as Core-Tic is involved in another dependency with CIrcleBoard, another *compensating message* is sent: reset-game noted with number 5 and the same control occurs (number 6).

3 The ObjVlisp Model

We briefly recall the ObjVlisp model [8] and especially focus on its management of the inheritance mechanism.

3.1 Architecture of the ObjVlisp model

"The ObjVlisp model supports a simple, clean and minimal architecture for explicit metaclasses." [3] ObjVlisp classes are explicitly and uniformly instances of some other classes: metaclasses. Moreover, ObjVlisp is minimal in the sense of being self-defined by two classes: the root of the instantiation tree (the class <Class> which is instance of itself) and the root of the inheritance tree (the class <Object> which is instance of the class <Class>) as shown in Fig. 3.

[7] The apply-method primitive checks if the looked up method has been found before applying it. When the method is not found an error message is sent to the receiver.

[8] The standard behavior control executes these messages in the order of the link instance declaration.

[9] We use this expression as a generic that all the compensating messages are performed.

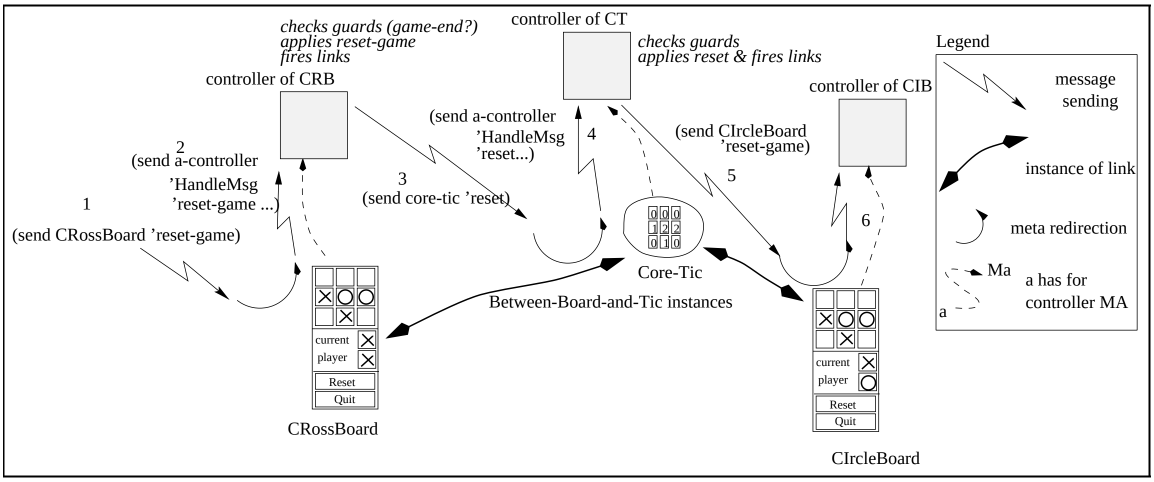

Figure 2: Metaobject control of message passing

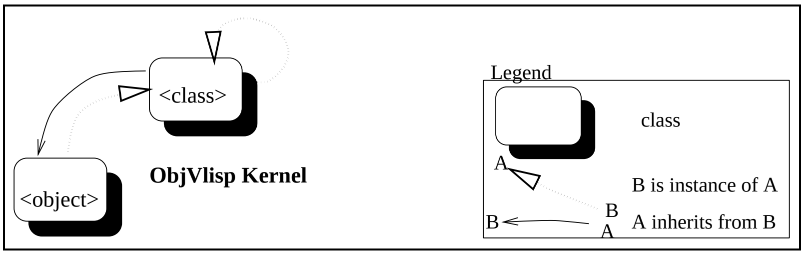

Figure 3: Architecture of the ObjVlisp core.

Structure of a class. A class is described by a name, an instance variable list, a method dictionary and a list of its direct superclasses. This class description is defined by the instance variables of the class which describes all the classes: the metaclass <Class>. In particular, the supers instance variable allows one to place a class in the inheritance graph.

3.2 Inheritance in ObjVlisp

As in Smalltalk-80, the inheritance mechanism in ObjVlisp is the combination of two distinct behaviors: the dynamic inheritance of methods and the static inheritance of instance variables.

Method inheritance. Method inheritance is dynamic. Message sending is composed of two distinct phases: first, the lookup of the method corresponding to the message selector, and second, the application of this method. Each time a message is sent to an object, the lookup phase goes through the inheritance graph, supported by the **supers** instance variable of the class <Class>, looking up the method: when it fails in the receiver class then the search continues in a depth-first/breath-first way. This lookup phase raises an error when the method is not found, i.e., when the lookup fails in the <Object> class. It is important to notice that the lookup algorithm is defined by a primitive in a global way for all the classes.

Instance variable inheritance. On the contrary, inheritance of instance variables is static and done once at creation time. When a new class is defined, its instance variables are computed as the union of all the instances variables of its superclasses and the instance variables specified at creation. In particular, all the objects inherit the **isit** instance variable from the <Object> class. This instance variable represents the instantiation link: an instance always knows its class.

4 The FLO model and inheritance reification

The model FLO is strongly based on the ObjVlisp model. The main difference between a traditional implementation of ObjVlisp and the FLO one is that in FLO the inheritance mechanism is reified by means of links: the inheritance between two classes is represented by an inheritance link instance. Moreover, the only restriction we chose is that managing inheritance by means of links does not imply changing the send semantics, i.e. the definition given in section 2.2 must not be modified due to this special inheritance management. In the subsequent sections, we present the consequences of our design choice at the class description and functionality level, and the implementation of inheritance links.

4.1 Structure and behaviors of objects

In the same way that the <Stack> class of the first example was not modified to be linked with the memory object (no instance variables were added to the stack definition to refer to a memory object), we take care to remove all the inheritance information of the class description. The class structure no longer needs to contain inheritance information: the instance variable **supers** is removed from the class description (line 3 in the below definition). The following definition describes the self instantiation of the metaclass of all the classes: the <Class>[10]. Object structure is simply extended to take into account the meta reference (line 8).

The class <Class> has a lookup method which only performs a **local** lookup of the method using the get-method primitive, which searches the method in the method dictionary (line 5). This method returns false (♯f) if no local method is found. We consider that the lookup phase is ensured at the metaobject level as in Ferber's implementation [18], but this lookup is managed by means of links.

```
1 (define <Class>
        (send <Class> 'new
2             :name '<Class>
3             :iv '(name iv keywords methods)
              ;; the superclass instance variable is removed from the list of the instance vari-
ables that defines a class.
4             :methods  '(basicnew (lambda () (allocate (name self)))
5                         lookup   (lambda (selector obj) (get-method self selector))
6                         new      (lambda initargs (send (send self 'basicnew) 'initialize initargs))
                          ...)))

7 (define <Object>
        (send <Class> 'new :name '<Object>
8             :iv '(isit meta)
9             :methods   '(?       (lambda (iv) (....))
10                         ?<-     (lambda (iv new-val) (...))
11                         error   (lambda ...)
12                          ...)))
```

4.2 Single inheritance mechanism as relationship management

Now, we present from an implementor's point of view how a single inheritance mechanism is expressed using links. This mechanism is hidden for a common user.

[10] The self instantiation of the class <Class> is possible due to a bootstrap mechanism similar to the one of ObjVlisp.

4.2.1 The inherit-from links

Using links, the semantics of the inheritance mechanism is explicitly expressed: dynamic lookup of methods and static instance variable inheritance.

Dynamic inheritance: two basic cases. For the dynamic method inheritance, the idea is simple: a link controls the class lookup method (in the same way that the reset-game method was controlled in the first example); a link tests whether a method has been found, if it is not the case, it looks up the method in the superclass. As an inheritance link locally ties classes, its semantics only requires the local behavior of the inheritance mechanism (for example, the error management is simply defined for links that are instantiated with inheritance graph root classes[11]). According to the class position in the inheritance graph two basic cases appear as shown by the two following link definitions (these links will be instantiated between two classes designated by the **:class** and **:superclass** variables.).

The semantics of the <Inherit-from> link is the following one: if the looked up method is not found in the class, we looked it up in its superclass (line 10), as shown in the definition below. Assuming the method could be defined in one of the ancestors of the superclass, the definition is purely local: the error notion is not managed by this inheritance link.

```
1 (deflink-inh <Inherit-from>
2      :behavior
3      ´(( lookup (:class selector object))
4         implies
5       (lambda (:class :superclass object selector :result)
6           (if (is-method? :result)
7               (begin                       ;; a method has been found
8                     (stock :class)         ;; we stock the class for the run-super
9                     :result)               ;; and we return it
10              (send :superclass ´lookup selector obj))))) ;; else the method is looked up in the superclass.
```

The other situation occurs when a class directly inherits from a root of the inheritance graph: for example, the class <Object>. In such an alternative, the semantics of link differs slightly, as shown in the definition below: when the method is not found in the class, the method is looked up using the get-method primitive in the superclass (line 10). If the method is found (line 11), it is returned, else an error message is sent (line 13). Using the primitive get-method instead of sending a message to the superclass is justified by the fact that we are sure to stop the lookup phase: the looked up method is either defined in this class or not defined.

```
1  (deflink-inh <Inherit-w-error>
2     :behavior
3     ´(( lookup (:class selector object))
4       implies
5       (lambda (:class :superclass object selector :result)
6           (if (is-method? :result)        ;; a method has been found
7               (begin
8                 (stock des)               ;; we stock the class for the run-super
9                 :result)                  ;; and we return it
10              (let ((m (get-method :superclass selector)))
                                 ;; the method was not found and the superclass is the bot-
tom of the inheritance graph
11                (if (is-method? m)
12                    m                                  ;; the method is found in the in-
heritance graph root
13                    (send object ´error selector)))))))    ;; no method has been found in the graph: an error
```

[11] In FLO, the class <Object> is not the only root of the inheritance graph. We will explained this particularity in section 5.

Both <Inherit-from> and <Inherit-w-error> link classes (noted in the following <Inherit-F/wE>) are instances of the <Metalink> class, which is the class of all the links. The <Inherit-F/wE> classes inherit from the <Link> class. All of this is done by the deflink-inh macro. It important to notice that a FLO user does not take into account this inheritance aspect, she/he simply programs in her/his habits and the system infers these different choices.

Static Inheritance. The link management of FLO provides an open protocol in the sense of the CLOS MOP[25]. Among all the methods which manage link behavior, the action-at-creation-time method is called each time a link is instantiated. This method allows one to specialize the behavior of a link by doing a specific action when an instance of such a link is created.

To manage instance variable inheritance, we specialize the action-at-creation-time method. When this method is called (an instance of <Inherit-F/wE> classes is created between two classes (associated with the variable **:class** and **:superclass** in this link instance)), the instance variables are computed as the union of instance variables of these two classes (line 5 in the definition below). Keywords are computed during the instance variable inheritance. In the original ObjVlisp model, instance variable inheritance was strongly associated with class creation, in the initialize method of <Class> class, whereas in FLO this occurs during link inheritance instantiation (line 4). In this paper, we only manage single inheritance; however, specializing such a method may permit the expression of different inheritance semantics (for example, management of instance variable name conflicts with multiple inheritance).

```
1 (add-method  <Inherit-from> ´ action-at-creation-time
2    (lambda (self)                                              ;; self denotes a link instance
       (let ((a (send self ´give :superclass))            ;; to know which class is the superclass,
3             (d (send self ´give :class)))                      ;; for this instance of in-
heritance link.
4          (keywords! d (add (keywords a) (keywords d))            ;; keywords are computed
5          (iv! d (remove-duplicates (append (iv a) (iv d)))))))   ;; union of instance vari-
ables is done
```

As FLO inheritance is separated from class creation, the presented code has a strong precondition: the superclass must already inherit from its superclass. For example: if A inherits from B, and B from C, declaring first that A inherits from B, and then B from C, leads to an error. In section 6 we propose a solution to this problem.

4.3 An example of dynamic inheritance by means of links

Let us suppose that we create a simple hierarchy: the <Point> class inherits from <Object>, and the <Colored-point> class inherits from <Point>, as shown in Fig. 4. This is done by defining: first, these two classes (lines 1-4 and 6-9 in definition below), second, the instance of <Inherit-w-error> between <Point> and <Object> (line 5), and third the instance of <Inherit-from> between <Point> and <Colored-point> (line 10).

```
1 (define <point>
          (send <Class> ´new :name ´<Point> :iv ´(x y)               ;; we create the <point> class
2               :methods ´(x       (lambda ...)
3                          display (lambda ...))
4               :controller (send <Controller-firing-result> ´new)))
5 (send <Inherit-w-error> ´new :class <Point> :superclass <Object>)
  ;; this class inherits from <Object> class. An instance of link is created
6 (define <Colored-point>
          (send <Class> ´new :name ´<Colored-point> :iv ´(color)         ;; the <Colored-
point> class is defined
```

```
7               :methods ´(color   (lambda ...)
8                          display (lambda ...))))
9               :controller (send <Controller-firing-result> ´new))
10 (send <Inherit-from> ´new :class <Colored-point> :superclass <Point>)
;; this class inherits from the <Point> class

11 (send <Colored-point> ´? ´iv)       >> (isit meta x y color)
  ;; static inheritance of instance variables is done and dynamic inheritance is performed to look up the '? method.
12 (define cp2 (send <Colored-point> ´new :x 33 :y 45 :color "blue"))
13 (send cp2 ´x)                        >> 33
```

As the ?[12] method is defined in the <Object> class, the message (line 11 of the following definition) illustrates the dynamic lookup of methods, as well as the inheritance of instance variables: isit and meta from <Object>, x and y from <Point>, and color from <Colored-point>.

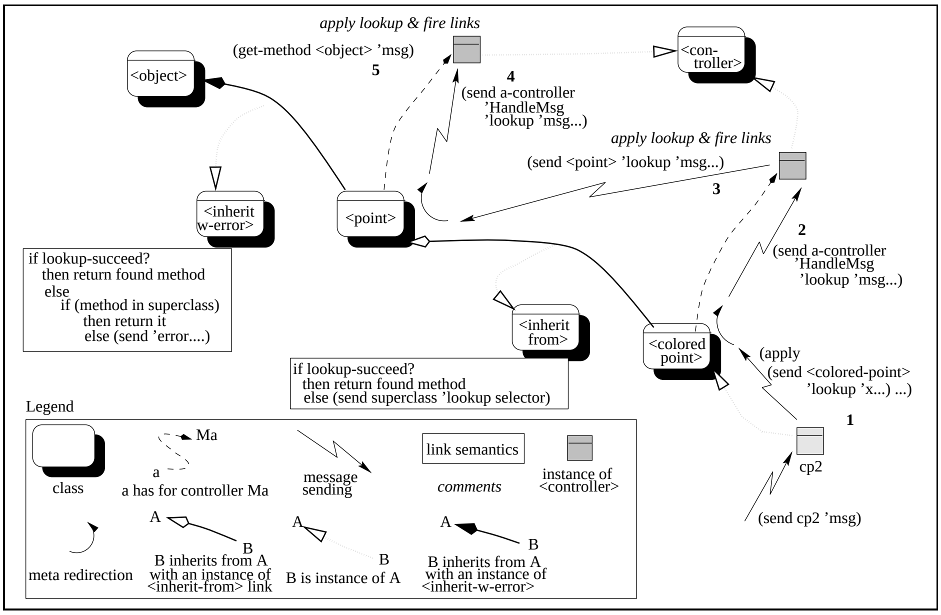

Figure 4: Using reified inheritance links: two basic cases.

Figure 4 illustrates the lookup of a message (msg) sent to the cp2 instance. According to the message three situations appear:

* when the invoked method is defined in the <Colored-point> class (e.g. color), this method is looked up (noted with the number 1 in Fig. 4) in the class <Colored-point> . This lookup message to the <Colored-point> class is controlled (noted with the number 2). However, as the method is defined in <Colored-point>, the lookup terminates.
* When the invoked method is defined in the <Point> class (e.g. x y...), the control (2) of the lookup message (1) sends a lookup message (3) in the superclass. This message is in turn controlled (4), but as the method is defined in the <Point> class the lookup terminates.
* When the method is neither defined in <Point> nor in <Colored-point> the previous case is extended: as the method is not found in the <Point> class, the controller of this class performs (5) the lookup in <Object> using the get-method primitive according to the

[12] This method defined on <Object> allows one to find the value associated with a given instance variable. It corresponds to the CLOS slot-value method.

<Inherit-w-error> semantics. So, either a method is found or an error is raised.

4.4 A different message passing control

By default, an instance of <StandardController> is associated with any linked object. However, in the previous definition of <Point> and <Colored-point> classes, we specify another controller using the :controller keyword[13] (lines 4 and 9 of the previous definition). The controller associated with the classes in the above definitions is different than the one which ensures the standard control of message passing (see 2.3). Indeed, the default controller is not adapted to the inheritance link management. Semantics of the <Inherit-F/wE> links implies the return resulting from the lookup method, the local method if it exists, or else the result of the *compensating message*. However, as we already said, the FLO final user is not concerned by all these details, which are hidden using syntactic sugar.

So, we define a specific controller which returns the result of the *compensating message* when a link[14] exists, instead of returning the result of the controlled method.

```
(define <Controller-firing-result>
   (send <Class> 'new
        :name ' <Controller-firing-result>
        :methods  '(HandleMsg (lambda (recvr selector args  method)
                                (let ((res (apply method args))  ;; we apply the method
                                      (lkf (send self :get-links 'implies recvr selector)))
                                  (if  (no-link? lkf)  ;; if there is no link
                                       res   ;; we return the result of its correspond-
ing message
                                       (send lkf 'firing recvr selector args res)))))))
                                  ;; else we return the controlled method  result
```

5 Bootstrapping issues and the FLO architecture

In this section, we discuss the bootstrap of the FLO kernel and present its architecture.

5.1 A simple infinite meta regression

To justify our bootstrapping choices, we present a first simple infinite meta regression in the message passing control. Let us suppose we want to represent the fact that the <Class> class inherits from the <Object> class by an instance of link. This implies that messages sent to <Class> must be controlled. However, this simple choice immediately leads to an infinite regression as illustrate in Fig. 5.

Let us suppose that <Class> receives a message msg (noted by the number 1 in Fig. 5 and in the decomposition below). As <Class> is self-instantiated, this selector is looked up in <Class> itself (2). As <Class> has a controller, the message HandleMsg is sent to it (3). This implies to looking up the HandleMsg method in the <Controller> class (4). Then, <Controller> class is an instance of <Class>, the lookup of the lookup is sent to <Class> (5). And the system infinitely loops.

```
(send <Class> 'msg)
   (send <Class> 'lookup 'msg)
       (send a-controller 'HandleMsg 'lookup 'msg <Class>)
           (send <controller> 'lookup 'HandleMsg)
               (send <Class> 'lookup 'lookup)
...
```

[13] This keyword allows one to explicitly specify the controller of an object.

[14] The presented code assumes that there is only one superclass for a given class, i.e., only one link has such a class for :class

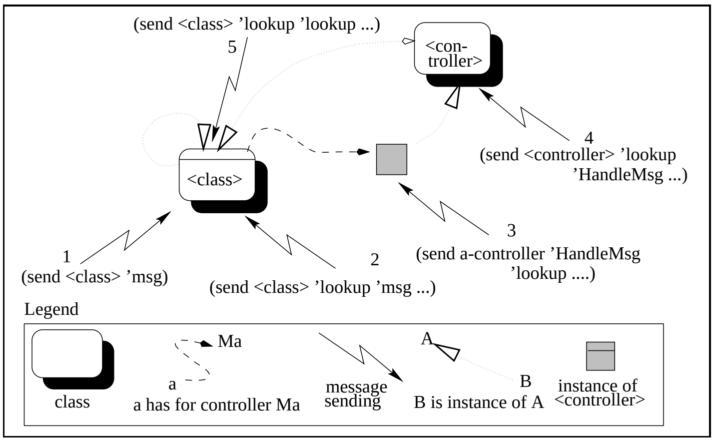

Figure 5: A first basic configuration that leads to an infinite loop.

Some different solutions can be found to stop this infinite regression. For example, we can test the lookup of the lookup method on <Class>. However, this type of solution is an ad-hoc solution. Indeed, a more subtle loop will appear as soon as the <Controller> class will inherit from the <Object> class thanks to links. In the same way, when <Metalink> will inherit from <Class> and <Link> from <Object>, trickier regressions should be stopped. Each time we would like to stop these infinite regressions, some new specific stop tests would have to be implemented by modifying the send primitive.

In fact, the observation of this mechanism leads us to chose another solution. As the fact that <Class> has a controller leads to an infinite loop, <Class> must not have a controller. As a consequence, inheritance between <Object> and <Class> cannot be managed by links, so <Class> must be autonomous from the inheritance point of view (this choice takes into account that we do not change the semantics of our send primitive (see section 2.2)). As shown in Fig. 6, all the classes responsible of the simplest message passing control are autonomous. For these classes, the message passing control cannot be itself controlled. These five classes conceptually inherit from each other, but they are self-contained. So, we can consider all of these classes as roots of the inheritance graph and that they could only be superclasses of <Inherit-w-error> link instances. However, we consider the bootstrap of the FLO kernel can be improved. This will be part of our future work to minimize the number of classes in the FLO kernel.

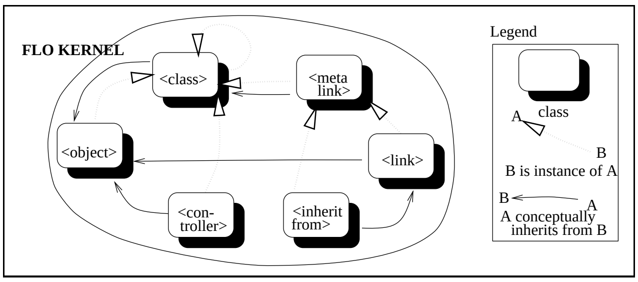

Figure 6: The kernel of the FLO language.

5.2 Architecture of FLO

According to the previous considerations, we illustrate the FLO architecture with figure 7. In addition to the ObjVlisp model, four classes appear: the <Metalink> class is the metaclass of all the link classes, the <Controller> class is the inheritance graph bottom of the controller classes, the <Link> class is the inheritance graph bottom of the link classes, and the <Inherit-w-error> class is the inheritance graph bottom of all the inheritance link classes.

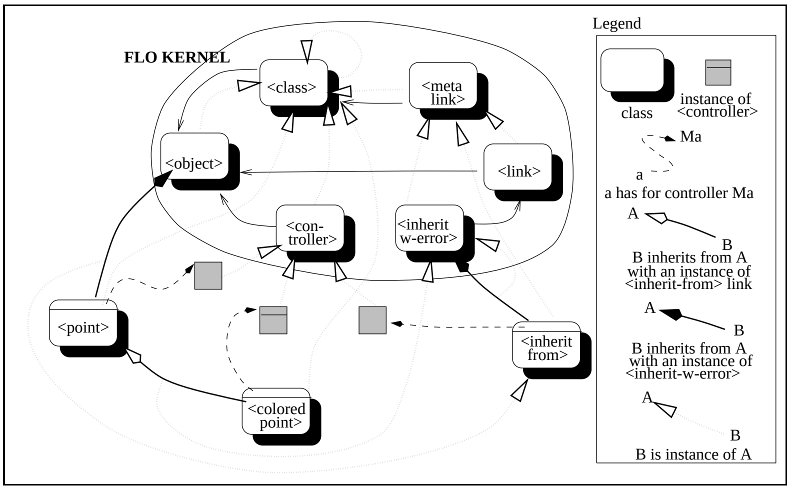

Figure 7: The kernel of the FLO language and an example of explicit inheritance between the classes <Object>, <Point> and <Colored-point>.

6 A first extension: dynamic instance variable inheritance

As we exposed in section 4.2.1, instance variable inheritance implies instantiating the inheritance links in a top down way: from <Object> to its subclasses. The idea of dynamic instance variable inheritance is the following: when we change the instance variables of a class (:superclass), we must propagate the change to its subclasses (:class) (line 5 in the definition below).

First, we define the method set-ivs for the new metaclass <Class-dynamic>. Second, we create two new link classes by completing[15] the definition of the inheritance links <Inherit-from> and <Inherit-w-error>, and third we specialize the action-at-creation-time method such that, instead of calling a primitive for instance variable management as in ObjVlisp, the set-ivs method will be called (line 10). The code below only describes the definition of the <Inherit-from-with-dynamic> link.

This first extension is simple and does not take into account the state of previously created instances. A similar mechanism to the CLOS update-instance-for-different-class could be implemented.

```
1 (deflink-inh  <Inherit-from-with-dynamic> :inherit-from ´( <Inherit-from>)
2   :behavior ´(((set-ivs (:superclass new-ivs))
3                 implies
4                (lambda (:class new-ivs)
5                   (send :class ´set-iv (remove-duplicates (append new-ivs (iv :class))))
```

[15]FLO provides a really special inheritance between link classes. So guards and compensating message of links can be reused.

```
            (keywords! :class (make-keywords (iv :class)))))))

(add-method <Inherit-from-with-dynamic> ´action-at-creation-time
  (lambda (self)
     (let ((s (send self ´give :superclass))
           (c (send self ´give :class)))
       (send d ´set-ivs (remove-duplicates (append (iv s) (iv c))))
       (keywords! d (make-keywords (iv d))))))
```

Figure 8 shows a simplified vision of the different inheritance links that manages dynamic instance variable inheritance.

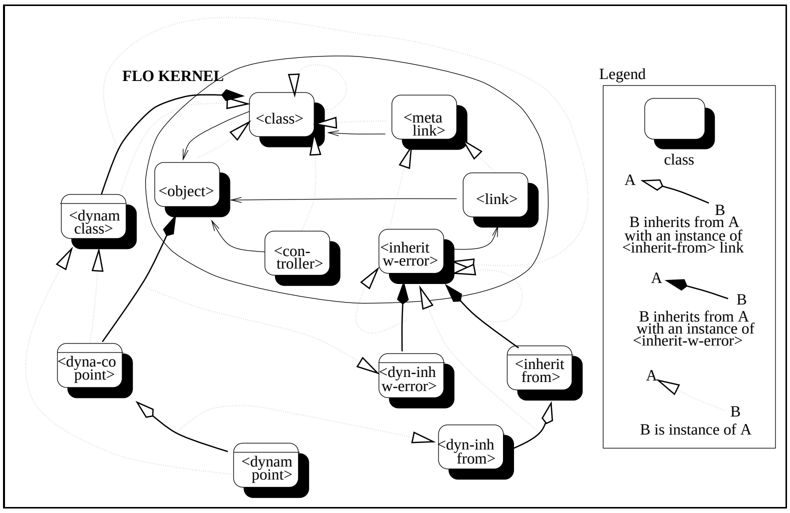

Figure 8: A first extension of inherit-from links: dynamic inheritance of instance variables.

7 Related Work

Controlling inheritance semantics and management has already been done in languages such as CLOS or MOOSTRAP, by means of MOPs, so we discuss these different approaches.

An open inheritance mechanism: the CLOS solution The MetaObject Protocol of CLOS [25] is currently considered as the quintessence of MOPs. It exhibits protocols responsible for all the aspects of program execution, amongst them being: class definition, instance structure, generic function invocation, and method combination.

The inheritance mechanism of CLOS is strongly based on the linearization of the inheritance graph, building a precedence list of superclasses for each class. First, instance variable *(slot)* inheritance is performed during the "Class Finalization Protocol" *(p93,p155 of [25])*. *"Class finalization is the process of computing the information a class inherits from its superclasses and preparing to actually allocate instances of the class"*[25]. The finalize-inheritance generic function is responsible for computing the class precedence list, calling the compute-class-precedence-list generic function *(p 78)*, the instance variable inheritance, calling the compute-slots generic function, and the full set of initialization arguments for the class. More precisely, the generic function compute-slots calls the generic function compute-effective-slot-definition to coalesce same-named slot definitions. Second, although CLOS is based on generic function application (and not on message passing like ObjVlisp and Smalltalk), the CLOS MOP allows one to specify the generic function invocation *(p160 in [25])*. Each time a generic function is called, its associated discriminating function is also called to provide the generic function behavior. Thus, the generic function compute-discriminating-function determines the ordered set of applicable methods, by calling compute-applicable-methods-using-classes or compute-applicable-methods, and determines the effective method, by calling compute-effective-method. Through all these functions, CLOS inheritance semantics can be adapted to user needs. However, the linearization of the class graph is the keystone of CLOS inheritance and provides a global semantics for the instance variable and method lookup. Unlike CLOS, FLO follows the same philosophy used by MOOSTRAP[32] in which delegation is kept within local metaobject behavior.

Moostrap and the Lookup ∘ Apply protocol MOOSTRAP [32] is a reflective prototype-based language which distinguishes itself from classical prototype-based languages. Indeed, instead of providing a built-in delegation mechanism, it provides a meta-link with which different delegation semantics can be re-introduced. MOOSTRAP is based on the lookup ∘ apply protocol as defined in [29]. The idea is to reify message passing through two phases: first the lookup of the requested slot[16], and then the application of its result. Furthermore, any object is associated with a metaobject, to which the lookup phase is delegated. The basic lookup semantics (defined for the basic-meta-object) only performs a local research of the slot in the receiver of the message, so no delegation is taken into account. Then, in [32], authors reintroduce delegation by defining a new meta-object (SD-meta-object), which performs delegation as a succession of elementary redirections.

8 Conclusion

This work follows the same philosophy as the re-introduction of the delegation mechanism in prototype based languages. Mulet and Cointe [32] propose an extensible model: although there is no delegation (the sharing mechanism in prototype languages) in the kernel of MOOSTRAP, they introduce delegation as a first extension using new behavioral metaobjects. Using a meta approach allows us to reintroduce inheritance without modifying our initial choices. In this sense, this tends to prove that *the meta link* is more primitive than *the inherit link* (respectively *the delegation link* for prototype-based languages [32]).

FLO semantics of inheritance mechanism can be modified thanks to link definitions and protocol. Inheritance mechanism is separated from the class definition (structure and functionalities) and creation. This mechanism is expressed in two dimensions: space (structure and functionalities) and time (when does it occur). Inheritance mechanism semantics is expressed by class methods themselves. Moreover, reifying inheritance by means of links gives FLO a uniform and reflective definition based only on two concepts: objects and dependencies. FLO only proposes objects and dependencies as a minimal core of object-oriented programming. Inheritance is defined as other dependencies. In particular, this allows one to specify and mix different inheritance semantics in the same language.

The following points will require our attention:

* *"AccessClass: This metaclass models classes which may provide automatic (read-write) accessors to their instance variables."*[3]. With the separation of instance variable inheritance from class creation,

[16] MOOSTRAP provides an unified solution for methods and instance variable representation: slots.

new problems appear. Actions that must be performed at class creation may be more subtle: for example, implementing AccessClass is not so simple as in the ObjVlisp or ClassTalk model. We seek to provide solutions to these problems.

* Using the expressiveness of our language, we foresee mixing different inheritance semantics and testing their compatibilities and advantages. Introducing multiple inheritance may be a test of extensibility of this work.
* As dependencies control the behavior of individual instances, we wish to compare our approach to the prototype one. For example, how can we express delegation by means of links and how can we mix delegation and inheritance.
* As of now, the system described in this paper is implemented. The system must be optimized (for example, cache mechanisms for the lookup phase). Testing only if an object has a metaobject for calling the HandleMsg method can be improved by controlling only necessary messages. Moreover, we plan to improve the FLO kernel bootstrap (minimizing the number of classes in the kernel).
* The use of metaobjects in link management is not completely safe. Indeed, the success of our approach is based on the correspondence between the links on the controlled object and the behavior of its metaobject. For example, how to manage at the same time an inheritance link and a link that implies standard control ? This problem is quite similar to problems due to the metaclass compatibilities presented in [20, 11] and it outlines the need of composition between metaobjects (when it is possible) [33].

Acknowledgments We like to thank Jacques Malenfant for his remarks on the earlier version of this paper and Philippe Mulet for our exciting discussions on the bootstrapping choices of the FLO kernel.

References

[1] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA'86*, pages 17–29, Nov. 1986.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90*, pages 303–311, Oct. 1990.

[3] J.-P. Briot and P. Cointe. Programming with Explicit Metaclasses in Smalltalk-80. In *Proceedings OOPSLA'89*, pages 419–432, Oct. 1989.

[4] L. Cardelli. A semantic of multiple inheritance. In *Lecture Notes in Computers Science, Semantics of data types*, volume 173, pages 51–67. Springer-Verlag, New-York, 1984.

[5] L. Cardelli and P. Wegner. On understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–521, December 1985.

[6] B. Carré and J.-M. Geib. The Point of View notion for Multiple Inheritance. In *Proceedings of OOPSLA/ECOOP'90*, pages 312–321, Ottawa, October 1990. ACM.

[7] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings ECOOP'93, LNCS 707*, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[8] P. Cointe. Metaclasses are first Class: The ObjVlisp Model. In *OOPSLA'87 Proceedings*, pages 156–165, October 1987.

[9] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA'89*, pages 433–443, Oct. 1989.

[10] J. Coutaz. The construction of user interfaces and the object paradigm. In *Proceedings of ECOOP'87, LNCS 276*, pages 121–130. Springer Verlag, June 1987.

[11] S. Danforth and I. R. Forman. Reflections on metaclass programming in SOM. In ACM, editor, *Proceedings of OOPSLA'94*, volume 29, pages 440–452, Portland, Oct. 1994. ACM.

[12] A. Dery and M. Fornarino. Dépendances comportementales et modèle PAC. In *IHM'94*, Lille, Dec. 1994.

[13] S. Ducasse. *FLO: the Manual Reference.* I3S, 1995. RR-95-04.

[14] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna. A Reflective Model for First Class Dependencies. In *Proceedings of OOPSLA'95*, pages 265–280, Austin, Oct. 1995. ACM. RR–95–24.

[15] S. Ducasse and M. Fornarino. Protocol for managing dependencies between objects by controlling generic function invocation. In *OOPSLA'93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, Washington, Oct. 1993. ACM. RR–94–62.

[16] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA'92, ACM SIGPLAN Notices*, pages 16–24, Oct. 1992.

[17] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. A proposal for a monotonic multiple inheritance linearization. In *Proceedings OOPSLA'94, ACM SIGPLAN Notices*, pages 164–175, Oct. 1994.

[18] J. Ferber. Computational reflection in class based object oriented languages. In N. Meyrowitz, editor, *Proceedings of OOPSLA'89*, pages 317–326. ACM, October 1989.

[19] E. Gallesio. Stklos: A scheme object oriented system dealing with the tk toolkit. In ICS, editor, *Xhibition 94, San Jose, CA*, pages 63–71, June 1994.

[20] N. Graube. Metaclass compatibility. In *Proceedings of OOPSLA'89*, pages 305–315. ACM, October 1989.

[21] D. C. Halbert and P. D. O'Brien. Using types and inheritance in object-oriented languages. In *Proceedings ECOOP'87, LNCS 276*, pages 20–31. Springer Verlag, June 1987.

[22] R. D. Hill. The abstraction-link paradigm: Using contraints to connect user interfaces to applications. In *Proceedings of CHI'92: the Conference on Human Factors in Computing Systems*, pages 335–342. ACM, May 1992.

[23] Y. Ishikawa. Reflection Facilities and Realistic Programming. *SIGPLAN Notices*, 26(8):101–110, August 1991.

[24] S. Jagannathan and G. Agha. A Reflective Model of Inheritance. In *Proceedings of ECOOP'92, LNCS 615*, pages 351–371, Utrecht, June 1992. Springer-Verlag.

[25] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[26] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, pages 26–49, Aug. 1988.

[27] H. Lieberman. Delegation and inheritance: Two mechanisms for sharing knowledge in object-oriented systems. *Bigre + Globule*, 48:79–89, 1986.

[28] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155. ACM, October 1987.

[29] J. Malenfant, C. Dony, and P. Cointe. Behavioral Reflection in a prototype-based language. In A. Yonezawa and B. Smith, editors, *Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo, Nov. 1992. RISE and IPA(Japan) + ACM SIGPLAN.

[30] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings of ECOOP'92, LNCS 615*, pages 231–250. Springer-Verlag, June 1992.

[31] J. McJaffer. Meta-Level Programming with CodA. In *Proceedings of ECOOP'95, LNCS 952*, pages 190–214. Springer-Verlag, Aug. 1995.

[32] P. Mulet and P. Cointe. Definition of a reflective kernel for a prototype-based langage. In *Proceedings of ISOTAS'93, LNCS 742*, pages 128–144. JSSST-JAIST, Springer-Verlag, Nov. 1993.

[33] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Oct. 1995.

[34] L. A. Stein. Delegation is Inheritance. In *Proceedings of OOPSLA'87*, pages 138–146, October 1987.

[35] P. Wegner. Dimensions of object-based language design. In *Proceedings OOPSLA'87*, pages 168–182, Dec. 1987.