A Reflective Model for First Class Dependencies

S. Ducasse ducasse@essi.fr

M. Blay-Fornarino blay@essi.fr

A.M. Pinna-Dery pinna@essi.fr

Université de Nice Sophia-Antipolis - CNRS
Bât. ESSI, 650 route des colles - B.P. 145
06903 Sophia Antipolis CEDEX

Abstract

We propose a reflective model to express and to automatically manage dependencies between objects. This model describes reflective facilities which enable the changing of language semantics. Although the importance of inter-object dependencies is well accepted, there is only limited object-oriented language support for their specification and implementation. In response to this lack of expressiveness of object models, the FLO language integrates dependency management into the object oriented paradigm. Dependencies are described as first class objects and FLO automatically maintains the consistency of the dependency graph.

In this paper, we first show how a user can declare dependencies and how the system maintains the consistency of the graph of expressed dependencies. In a second part, we focus on the implementation of this management by controlling the messages sent to linked objects. In order to make dependency management orthogonal to other application concerns, we propose an abstraction of message handling, implemented with meta-objects. We illustrate the extensibility of our language with different control behavior implementations, in particular we study different implementations of the global control of message propagation flow.

Keywords: Dependencies, Computational reflection, Control of message passing, Metaobject Protocol

1 Introduction

Although the importance of inter-object dependencies is well accepted [BC89, BELR92, SRHB89], there is only limited object-oriented language support for their specification and implementation. Confronted with this lack of expressiveness in object models, the programmer has to use traditional object features, such as attributes, to store the references to linked objects, accessors, or daemons, in order to manage constraints and interactions among objects.

Models such as MVC attempt to address this problem by integrating the notion of an object having dependents. A dependency mechanism, based upon the appropriate message sending, allows the user to manage the consistency of dependencies. However, such designs of behavioral relationships [HHG90] lead to several drawbacks. The dependency semantics is not clearly expressed, and dependencies are often hard-wired into object functionalities. Consequently, object structures and functionalities are polluted by non-intrinsic information [HO93]. This goes against the principle of modularity. It is then difficult to modify, specify, or maintain objects and dependencies. Reuse capabilities also decrease [BBB93]. Moreover, relationships between objects induce a dependency graph in which messages are propagated to maintain the global consistency of the dependencies. As relational information is mixed with intrinsic information, traditional implementations do not highlight the problem of controlling the message propagation flow and do not offer the ability to express different controls of the method propagation according to the application needs.

In response to this lack of expressiveness of object models, the FLO[1] [DF93] language is an extension of the object-oriented paradigms integrating dependency management.

In FLO, the user defines dependencies specifying which methods have to be controlled. Next, he/she can declare the objects involved in the dependencies, without altering object classes. FLO then automatically maintains the consistency of the graph of declared dependencies, by controlling the messages sent to the related objects. The language is currently used for knowledge representation [DFP91] and in the domain of User Interface Management [DF94].

In this paper, we present the language implementation based upon meta-objects. This aspect of the language is transparent to the user; more often the standard behavior of dependency management in FLO is sufficient for his/her application. However, when the application is complex, the computational reflection allows one to reason and to redefine the dynamic behavior of computation within the language framework[2]. Thus, the work presented in this paper is in the same line as [Mae87, Coi88, Fer89, Ish91, Kic92] : "*The metaobject protocol approach,..., is based on the idea that one can and should "open languages up", allowing users to adjust the design and implementation to suit their particular needs.*" [KdRB91].

To implement the mechanism of dependency consistency management, in such a way that it can be modified or even extended, we are faced with three difficulties:

- How to integrate the maintenance mechanism in a class-based language, so that only related objects are controlled?
- How to control the reacting propagation flow?
- How to adapt, trace and extend the language?

[1]FLO, standing for First class Links between Objects, is a scheme-based object-oriented language.

[2]Now, extensions are more often implemented by the FLO implementors.

This paper is organized in two main parts: we first give an intuitive view of definition and maintenance of dependencies in FLO, and then we focus on the implementation based upon computational reflection. More precisely, section 2 illustrates the relational capabilities of FLO. Section 3 justifies the choice of meta-objects in order to implement the management of dependencies and presents the standard implementation of the control of the message propagation flow. Extensions of the language capabilities in terms of relational expressiveness and of message propagation flow control end this section. Finally, we give an overview of related work and we conclude.

2 An intuitive approach of dependencies in FLO

In this section, throughout of a simple example, we describe the FLO language, i.e the nature of our dependencies, the way they are defined, and the process of consistency maintenance.

2.1 Dependency declarations

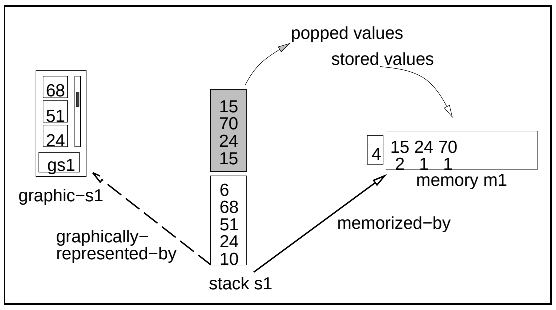

Figure 1: Stack, memory and stack representation.

Suppose the user has created two independent objects: a stack object s1 and a memory object m1. The class stack defines four methods: pop, push, empty?, and empty. The class memory defines the methods store, unstore and not-full?. Now, he/she wishes to express a dependency, called *memorized-by*, between s1 and m1, without altering the two class definitions of these objects (see Figure 1). The user defines a *link*[3], named *memorized-by*, between a stack instance and a memory instance. This link is expressed by referring to stack and memory methods, as shown in Figure 2, so that all popped values of the stack are stored in the linked memory, until the memory is full.

```
1 (deflink memorized-by ;; (:stack :memory)
2   :behavior
3    (((pop :stack) implies (store :memory :result))
   ;; a popped value is stored in the memory
4    ((pop :stack) permitted-if (not-full? :memory))))
   ;; stack can pop an element only if memory is not full

5 (define s1 (send stack :new))
   ;; we create a stack and a memory
6 (define m1 (send memory :new))
   ;; we create a link between these instances
7 (define s1−⟩m1
   (send memorized-by :new :stack s1 :memory m1))
   ;; Once the link s1−⟩m1 is created
   ;; the pop method is controlled. s1 and m1 are
   ;; considered as participants of the link s1−⟩m1
```

Figure 2: One simple dependency concerning stacks and memories: definition and instantiation of the *memorized-by* link.

Line 3 of Figure 2 shows that, when pop is called to the object denoted by the *:stack* variable[4], store must be sent to the object designated by the *:memory* variable, with the result of the pop call as argument, we call such a message, (store :memory :result), a *compensating message*. 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*. Likewise, in line 4, pop will only be performed on *:stack*, if *:memory* is not full. Thus, the semantics of the **permitted-if** operator is that the method can be applied only if the expression following such a **permitted-if** operator is true. We call such an expression a *guard*[5] (not-full? :memory).

The link *memorized-by* can be used for linking any instance of stack with any instance of memory. The instances associated to the variables s1 and m1 become dependent when an instance of the *memorized-by* link is created between them (see line 7 in Fig. 2). The system associates them respectively with the two variables *:stack* and *:memory* of this instance of *memorized-by* link. After that, FLO automatically ensures the consistency of the *memorized-by* link, controlling the messages sent to those instances, in accordance with the link definitions (see section 2.2).

Let us now suppose that the user wants to have a graphic representation of a stack. On the one hand he/she has defined the stack object s1, on the other hand he/she has defined a possible graphic (graphic-s1) representation with appropriate methods. To link these objects s1 and graphic-s1, he/she can define a new dependency, called *graphically-represented-by*, such that pop method calls imply the removal of the corresponding graphic value, push calls lead to the addition of a new graphic value, empty calls reset the representation and a selection of the bottom of the representation implies sending the pop message to the stack (see Figure 1).

This definition is independent of the *memorized-by* definition even if such links concern the same objects. Thus, the stack instance s1 can be linked at the same time to the memory instance m1 and to the graphic representation graphic-s1. Adding or removing dependencies is independent of the other links and of the other linked objects.

From this simple example, we point out some characteristics of FLO:

FLO preserves the encapsulation principles: The semantics of the links is only expressed using the interfaces of the related objects. As

[3]This term always refers to a reification of a dependency.

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

[5]The guards are used to express constraints on objects, but the *permitted-if* operator could be used to check pre-conditions as in software engineering [Mey90] (with a dependency on a single object). However, such pre-conditions are intrinsic to the behavior of the object itself, and are not relational information. For us, this possible use of this operator seems in contradiction with the initial *guard* semantics.

the dependency implementation is not buried into the object code, dependencies are defined in an independent way of the linked objects, enforcing the principle of encapsulation [Sny86]. Conversely, the code of the objects is independent from the dependencies, thus enabling higher modularity.

Links are dynamic: The creation and the destruction of links are completely dynamic. In the above example, one can dynamically add or remove new representations for a stack or relate it to other objects without interfering with the methods of the stacks or with the other links on the stack.

Links are multi-directional: Objects do not have an explicit role of master or slave in a dependency. For example, messages sent to the stack can imply *compensating messages* to the graphic representation. Conversely, messages sent to the representation can imply *compensating messages* to the stack.

Links are reified: Links are first class objects, they can be manipulated and documented. We can express links on links, such as *implication* (*subset* in [BELR92]), *inverse* or *exclusion* [NECH92].

Reusability: An immediate benefit of FLO links is the object reusability induced by the non modification of classes (contrary to MVC model, no subclass definitions are necessary). Moreover, the definition of links can be incremental using inheritance between links.

Programming: Once links are defined and instantiated, FLO ensures their consistency. The programmer is free from management of link consistency, and so his/her code is more clear and more accurate. Let us note that some primitives allow the user to ask about the existing links concerning an object.

Moreover, links are n-nary: we did not present this aspect of the language in this example, but a dependency in FLO can relate several objects through different variables. In particular, when the cardinality of a link is undefined (set of objects), FLO provides primitives to add or remove objects in the link.

Once dependencies are defined, FLO ensures their consistency. We detail FLO dependency consistency management in the following section.

2.2 Dependency consistency management

Before becoming a *"participant"* [HHG90] of a link, an object is *free*, and its messages are handled normally. On the other hand, as soon as an object is related to another one, FLO has to control some of its messages in order to enforce the consistency of the dependencies. Each object can be participant of several links. The set of links constitutes a graph. Thus, we distinguish two complementary kinds of message control: the local and the global control. When a message is sent to an object participant of some links, the local control ensures the consistency of the links on this object. The global control ensures the consistency of the whole link graph.

We describe the standard behavior of these two controls in this section.

2.2.1 Local control

The `local control` examines the *immediate* actions to perform when a message is sent to a linked object.

When a linked object receives a message, all the *guards* defined on this message in the different dependencies concerning this object have to be checked before applying the invoked method. If each *guard* is verified, the invoked method is applied, otherwise the message is not performed. After the method is applied, all the *compensating messages* are called. The standard behavior of the local control executes these messages in the order of the link instance creation.

In the example illustrated by Figure 1, a stack s1 is linked to a memory m1 and to a graphic object graphic-s1. Thus a call to the pop method of the instance s1 implies: (a) the verification of the *guard* concerning the state of the memory, and (b) if this *guard* is verified, applying the pop method to s1, and calling the store method of m1 and the remove-top method of the object graphic-s1.

However *compensating messages* may themselves be controlled. This last point induces the need of a global control of the propagation flow, that we describe next.

2.2.2 Global control

As we said above, *compensating messages* are called in reaction to a message. These *compensating messages* may in turn involve new *compensating message* calls for the re-establishment of the consistency of other dependencies. The standard behavior of the global control in FLO is to repeat the local control on all the *compensating messages*. Thus, the message propagation is realized in a depth first order.

Figure 3 illustrates the message propagation in a graph of links where a stack s1 is *memorized-by* a memory m1 and is *graphically-represented-by* an object graphic-s1; the memory is also represented by a graphic object graphic-m1. A pop message to the stack s1 implies storing the popped value in the memory m1; the execution of this last message implies reporting this modification to the object graphic-m1. Next, the pop message to the stack s1 implies sending the corresponding *compensating message* to the object *graphic-s1*.

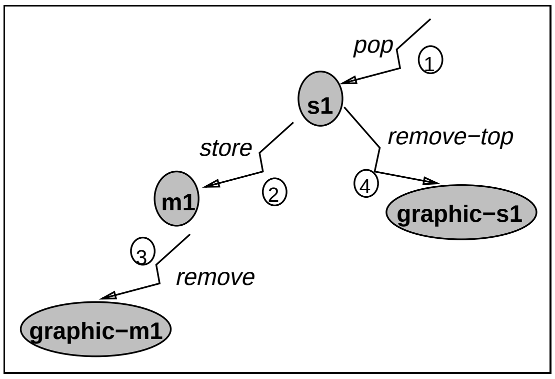

Figure 3: Example of a global propagation

The underlying graph of dependencies may be complex. Thus, causality cycles may appear due to the propagation of *compensating messages*. In order to solve this problem, we propose two orthogonal solutions.

- FLO offers the ability to cut some cycles by providing a way of detecting whether the *compensating messages* need to be called: when this is possible the programmer can statically associate an *invariant* to the link definition, such an invariant will be checked before applying the *compensating messages*.
- Some other cycles may occur during the management of dependency consistency, if one of the *compensating messages* reenters in the same propagation flow. A mechanism of interruption of these cycles is implemented as standard in FLO[6].

2.2.3 FLO standard control limitations

FLO has been used in different applications: to enforce the separation between application and graphic objects in graphic interface [DF94], for knowledge representation in the domain of building [DFP91], for studying inheritance mechanisms [Duc95], and for tracing message propagation flows. These experiments have highlighted the limitations of the standard behavior of the consistency management, leading us to “open up” the language to allow more accurate controls according to user needs.

In particular, some new operators were needed according to the applications. We present the introduction of a new operator at section 3.3.1. Moreover, the standard evaluation order chosen in FLO may not always be well adapted. Indeed, according to the user applications, a breadth first order may be more interesting because the consistency of the dependency graph is more rapidly satisfied. Moreover, as some messages may be redundant in a reaction chain, a global control could eliminate them. At the moment, we are working on this point to reduce the list of compensating messages.

Another problem arises when the propagation cannot lead to a global consistency state. It is the

[6] For the detection of cycles, we recall the sequence of calls that have already implied compensating messages during the link consistency re-establishment [DF93].

case when one of the *guards* is not verified in the chain of reaction or when there is no fixed point to stabilize the reactivity. A mechanism of backtrack could be implemented. For example with graphical objects, the actions could be performed on object "phantoms" and not objects themselves [CFZ94]. When the link graph consistency can be re-established, the system performs actions on real objects or substitutes "phantoms" to objects (when it is possible).

Finally, one drawback of this kind of "reaction oriented programming" is the difficulty in tracing and debugging such applications.

3 FLO implementation: a meta-object approach

In this section, we first explain why meta-objects are well adapted to implement the control of dependencies in FLO. Secondly, we describe the standard implementation of the language by means of meta-objects. Finally, we conclude with some examples of extensions of local and global control.

3.1 Meta-object: a concept well adapted to the implementation of an extensible control in FLO

The dependency consistency management involves checking *guards* and automatically performing *compensating messages*. Moreover, the tracing and control of message propagation are necessary, and users need to redefine the message sending algorithm according to their application needs.

The meta-objects as defined by Maes in [Mae87, Mae88] allow abstraction of the computation of an object, that is why they are well adapted to implement the FLO's dependency management. *"The meta-object 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 meta-objects), and to temporarily attach such a special behavior to an object."* [Mae87].

To manage the consistency of dependencies, we associate a meta-object with each linked object, thus conferring on FLO the following three properties:

- First, separation of object and dependency management functionalities is clear: an object exclusively represents information about the domain entity that it represents.
- Second, abstraction of the message handling allows one to implement the consistency maintenance mechanism.
- And third, as dependencies link instances, a meta-object approach allows FLO to only control necessary objects instead of penalizing the entire system.

These three features are necessary to offer an efficient and extensible management of dependencies. We will show in section 3.3 how we can use the meta-objects to extend the expressiveness of the language and to propose different controls.

Many researches which deal with procedural [Smi82] or computational reflection [BKK$^+$86, BDG$^+$88, Mae87, KdRB91] have been realized. We have chosen to base our meta-object implementation on Ferber's work [Fer89]. Ferber replies to the following question: how can we represent meta-objects in a class-based language? He proposes three solutions: using classes as meta-objects, meta-objects as instances of a class Meta-Object, and reification of communications. The second one is well adapted to our purpose because it allows us to control the dependencies at the instance level.

Indeed in FLO, the system does not penalize all the instances of a class to manage dependency consistency if only some instances of a class are related. Whereas with the first solution, controlling a single object is not impossible, but it does require a new class: every instance of a class shares the same meta-object, its class. Moreover, to quote Ferber himself, reification of communications *"does not say anything on objects: it is impossible to monitor objects, to represent specific information about receivers, or to represent the behavior of a single object."* Thus, our need of instance control leads us to eliminate this third solution. Note that, this solution is orthogonal to both the other ones and can easily be mixed with them.

With the chosen solution, classes are used for *structural* description (definition of an instance structure and a set of applicable operations) and meta-objects for *computational* description (how a message is interpreted, a method is applied, and controlled). In FLO, the meta-object of an object is called its *controller*. A controller can be shared between several objects, and these objects share the same control of methods.

3.2 Implementation of standard control in FLO

In order to present the behaviors of controllers, we use the same syntactic conventions as Ferber in [Fer89]: the object model OBJVLISP [Coi87], the syntax of FLAVORS for the definition of methods, the CLOS [Ste90] one for class definition. We choose the following syntax for message passing $\langle message \rangle$ ::= (**send** $\langle object \rangle$ $\langle selector \rangle$ $\langle arguments \rangle$) and internal slots are accessed as variables.

```
(define send (obj sel . args)
  (let ((meta (meta-of obj)))
     (if meta    ;; if there is a meta-object
        (send meta :HandleMsg ;; delegate message to it
           (send Message :new
                    :sender meta :receiver obj
                    :selector sel :arguments args))
        ;; else default message handling
        (apply (default-lookup obj sel) obj args))))

(defmethod (Meta-Object HandleMsg) (msg)
  (let* ((sel (send msg :selector))
        (args (send msg :arguments))
        (rec (send msg :receiver))
        (meth (send self :lookup (class-of rec) sel)))
     (if (null? meth)
        (send rec :doesnotunderstand sel args)
        (send meth :apply method rec args))))
```

Figure 4: Ferber's *send* primitive and **HandleMsg** method

In order to embed meta-objects, Ferber explains that some changes should be made in the kernel of the language: each object must have a slot **meta** and the **send** primitive must be modified[7]. As shown in Figure 4, the **send** primitive checks if the receiver object is bound with a meta-object. In this case, the message is delegated to the meta-object through the call to the method **HandleMsg**. Otherwise, the default message handling is performed. Note that Ferber reifies the communications.

```
(define send (obj sel . args)
   (let ((meta (meta-of obj)))
     (if meta
        (send meta :handlemsg obj sel args
                   (default-lookup obj sel))
        (apply (default-lookup obj sel) obj args))))
;; Assuming default-lookup manage unknown messages
```

Figure 5: FLO's *send* primitive

In FLO, the **HandleMsg** method definition is different. Indeed, the role of a FLO meta-object (controller) is to control the execution of messages to enforce dependency consistency. Therefore, when a message is sent to a linked object, its controller checks whether the *guards* are satisfied. If yes, after applying the accepted message, it must apply *compensating messages* (see 2.2). This standard behavior of controllers is described in the **HandleMsg** method of a controller (see Figure 6).

The **HandleMsg** method implements the main steps when a message is handled: the link search, the *guard* verification and the *compensating message* calls. These steps are key points of our protocol. Each of them is implemented by a method associated to the controllers. Thus, to adapt or to extend the standard behavior, one just has to redefine them. Let us show more precisely these methods: **get-links**, **combine-firings**, and **combine-guards**.

- **get-links** allows the controller to know whether links exist (lines 4 and 7 of Fig. 6) for a triplet: object (the receiver of the message), method (the controlled one) and operator (**implies** or **permitted-if**).

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

```
1(defmethod (StandardController HandleMsg)
                                (rec sel args method)
2   (if (null? method) ;; if no method has been found
      ;; we signal it
3     (send rec :doesnotunderstand sel args)
      ;; else before applying method, we check if
      ;; application is authorized.
4     (let ((lkgs (send self :get-links 'permitted-if rec sel)))
        ;; all guards of links are checked
        (if (and (pair? lkgs)
5                (send self :combine-guards
                           lkgs sel rec args))
          ;; method is applied
6         (let* ((result (send method :apply rec args))
          ;; and all compensating messages are performed
7                (lkfs (send meta :get-links 'implies sel rec)))
            (unless (null? lkfs)
8                (send meta :combine-firings
                       lkfs rec sel args result))
            ;; result of applied method is returned
9           result)))))
```

Figure 6: FLO's standard HandleMsg method

- combine-guards: Its role is to specify the combination of *guards* (line 5 of Fig. 6). For example, specifying this method allows one to express that only certain kinds of guards must be checked. By default, such a verification returns true if all the link *guards* are verified.
- combine-firings: Its role is to combine *compensating messages*. By default, combine-firings has no result and only sends the *compensating messages* in the link declaration order (line 8 of Fig. 6). You can see an example of another behavior in section 3.3.2.

Some remarks must also be made about the implementation of the HandleMsg method in Figure 6.

- First, we do not reify the message in the send definition. Reifying messages does not appear necessary in our use of meta-objects; however, it could easily be done as in [Fer89].
- Second, in the same way as in the default message passing handling (line 5 of Fig. 5), we distinguish, in the send definition, the lookup phase and the application method phase for the meta-level shift (line 4 of Fig. 5). Therefore, we add a fourth argument to the HandleMsg (line 1 of Fig. 6)). This new argument is the method[8] corresponding to the lookup result. As an advantage, this argument emphasizes extension facilities, avoiding calling the lookup method twice when we change the semantics of control (as shown in the following section 3.3.1).

Discussion about the standard implementation: The standard behavior of the consistency algorithm mixes local and global propagation of the reacting flows. Indeed, the proposed algorithm is an *implicitly* depth first algorithm, due to the evaluation of *compensating messages*.

As a consequence the control of the reacting flow is not simply modifiable; for example, the elimination of some redundant *compensating messages* is difficult to implement.

3.3 Examples of extensions of the language

In this section, we highlight the extensibility of our approach: first, we show how the language expressiveness can be extended for the benefit of the application concerns, by adding a new operator: the *corresponds* operator. Second, we outline how the extensibility of FLO might overcome some of the previous drawbacks of the standard approach by implementing different message sending algorithms.

3.3.1 Corresponds: a new operator in FLO for propagating messages

We agree with Ibrahim *"It is frequently desirable to modify the behavior of an object without changing the behaviors of other instances of its class ...Ad-hoc specialization should also allow the developer to create objects with new behaviors without requiring generalization to class definitions"* [IBC91], and claim that an object can have more functionalities due to the existence of dependencies. However, as

[8]This method is the one normally applied to the controlled object when there is no link.

soon as such an object becomes *free*, such functionalities must vanish. Therefore, the idea is to allow an object to answer to some new messages (defined by the dependency) as soon as it belongs to a dependency, so we propose a new operator: the *corresponds operator*.

For example, *corresponds* can be used to manage a kind of composition problem between whole-part entities, exposed by Blake in [BC87]: "*The whole protocol which a part understands ... will have to be re-implemented as the protocol of the whole. The net result is that the part hierarchy is replaced by a single monolithic whole as far as the external world is concerned*" [BC87].

The new operator, **corresponds**, allows one to declare that a message received by one object of the link has to be re-sent to another object of this link. In this sense and with the same syntax as in figure 2, (method1 :object1 argn) **corresponds** (method2 :object2 (fct argn)) means that when a *corresponding message* to the method1 is not defined, the controller of the message receiver object sends another (or the same) message, using (or not) calling arguments, to another object (or to itself).

For example if the user defines a dependency integrating the following definition : (color :whole) **corresponds** (color :part). If we link a car (the whole) to a coach-work (the part), when we ask the car for its color, this message will be re-sent to the part which is able to respond to this message.

With the *corresponds* operator, "*the whole protocol which a part understands*" need not be re-implemented, the *corresponding messages* can be introduced thanks to dependencies.

As shown in Figure 7, adding the *corresponds* operator only requires the definition of a new controller and to specialize the HandleMsg method[9]. When the method which has been sent is found, the standard HandleMsg method is performed (line 8). On the other hand, when no method is found, the controller checks whether links with a *corresponds* operator exist (lines 4 and 5 of Fig. 7). In such a case, the controller returns the result of the combination of the *corresponding messages* (line 6 of Fig. 7). If the method sent has no *corresponding messages*, an error is signaled (line 7 of Fig. 7). As we need a different combination of the *corresponding messages*, we define a new combination method, which returns the result of such a combination: by default a list of the *corresponding message*'s results.

```
1 (defclass Corresponding-Controller
                                (StandardController) ())

2 (defmethod (Corresponding-Controller HandleMsg)
                         (rec sel args method)
3   (if (null? method)
      ;; instead of saying we don't understand a message
      ;; we try to ask some other related objects
4     (let ((lkds (send self :get-links 'corresponds rec sel)))
5       (if (pair? lkds)
          ;; we return the combination of
          ;; the corresponding actions
6         (send self :combine-corresponds
                         lkds sel rec args))
          ;; if nobody can perform the asking message,
          ;; we signal it
7         (send rec :doesnotunderstand sel args)
      ;; if a method has been found,
      ;; we perform the standard behavior
8     (send super :HandleMsg rec sel args method)))
```

Figure 7: HandleMsg method for *corresponds* operator

Note that a dependency using the *corresponds* operator may be declared on an object with an inadequate meta-object, and the consistency of the dependency will not be ensured.

3.3.2 Extending the global control of dependency consistency

As explained in 2.2, the consistency of dependencies is re-established in a local way: each time an object receives a controlled message, the same control is performed without taking care of the other *compensating messages*. We now present a control which integrates a global view of the propagation algorithm, such as depth first, breadth first, or stepping propagation. Suppose for example, controlling m_1 implies m_{1i} as *compensating messages*, which implies sending messages m_{1ij}, we can then obtain

[9]Note the use of the get-links method with another operator at line 4.

a depth first or a breath first reacting flow, as shown in Figure 8.

As described in Figure 6, our standard behavior is *implicitly* a depth first algorithm. In order to implement new behaviors, such as a breadth propagation flow, we make the structure of the control apparent.

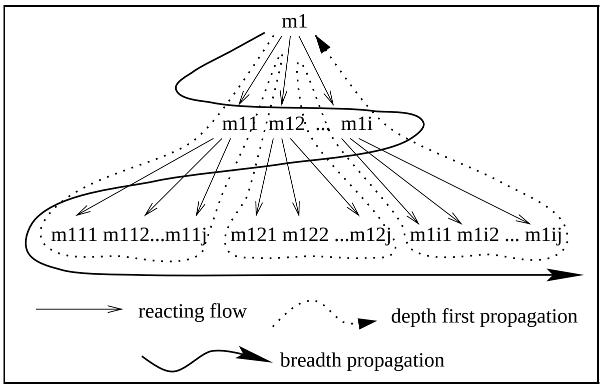

Figure 8: Two flows of reaction during compensating phase: depth first or breadth first propagation

Thus, we create a new controller class, called *AbstractController* with a new instance variable **place**, (line 2 of Figure 9). This instance variable will be used to stock *compensating messages*. Three methods are provided to manage it: **put**, **get**, and **empty?**. Defining such abstract methods allows us to specialize the behavior of message passing control, as shown in Figure 10. We redefine **combine-firing** method: instead of calculating, combining and applying *compensating messages* this method must calculate, combine and return a list of *compensating messages*. These returned messages are *put* (line 11) in the slot *place* of the controller and *get* (line 13) at convenience.

Depth first control or standard behavior In order to have depth first propagation, we create a new controller inheriting from *AbstractController*. Then, *place* must be considered as a *stack*. Therefore the **put** method must have a push semantics and the **get** method a pop one (if the stack is empty, the **get** method returns nil).

Breadth propagation of control: For a breadth propagation scheme we create a new controller

```
1  (defclass AbstractController (StandardController)
2    ((place))

3  (defmethod (AbstractController HandleMsg)
                                    (rec sel args method)
4    (if (null? method)
5      (send rec :doesnotunderstand sel args)
       ;; as in default behavior before applying method,
       ;; we check if application is authorized
6      (let ((lg (send self :get-links 'permitted-if rec sel)))
7        (if (and (pair? lg)
                (send self :combine-guards lg sel rec args))
8          (let* ((result (send method :apply rec args))
9                 (lkfs (send meta :get-links
                                    'implies sel rec)))
            ;; and all "compensating messages"
            ;; are calculated and put into place
10          (unless (null? lkfs)
11            (for-each (send meta :put x)
12              (send meta :combine-firing
                               lkfs rec sel args result)))
            ;; one compensating message is applied
13          (let ((m (send meta :get)))
14            (unless (null? m)
15              (send m :apply rec args)))
16          result)))))
```

Figure 9: HandleMsg method for propagation with apparent structure

which inherits from AbstractController. The only specifications needed are: *place* of the controller must be considered as a *queue*; the put method must have *enqueue* semantics and the get method a dequeue one. The HandleMsg method is not modified.

Stepping control: Such an implementation of control makes the addition of new functionalities easier. In particular, we introduce some stepping or debugging features in dependency control by decomposing all the *compensating messages*. As shown in Figure 10, we create a new controller subclass of DepthFirstController (or BreadthController), with a new instance variable step-mode which indicates if the control is normal or in a step by step state propagation. We define two new methods: toggle-step-mode to switch the step mode and apply-next-implied which calls the get method and applies the returned value. Moreover, to have alternatively normal and step by step propagation, the get method must be changed.

Discussion: These control behaviors are quite simple, however they exemplify the extensibility of our model. Similar behaviors have been used to eliminate redundant *compensating messages* in some simple reaction chains and to implement an algorithm of propagation with backtrack.

However, note that the use of meta-objects to dependency management is not completely safe. Indeed, the success of our approach is based on the correspondence between the dependencies on the controlled object and the behavior of its meta-object. This problem is quite similar to problems due to the metaclass compatibility presented in [Gra89, SD94] and it outlines the need of composition between meta-objects (when it is possible) [MMC].

4 Related work

Multiple attempts to express and to ensure consistency of dependencies between objects exist, and we present here the most significant ones.

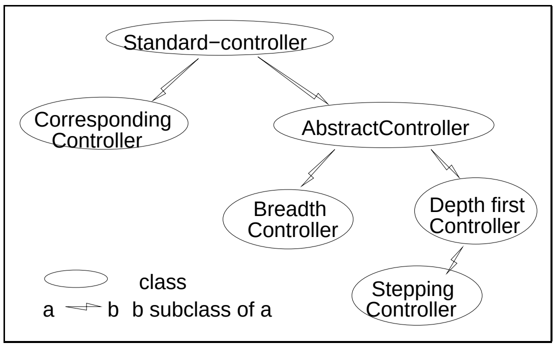

Figure 10: Hierarchy of controllers

FLO controls messages rather than only instance variable accesses, so active values [SBK86] can be seen as a subset of our control capabilities.

"Relations between objects are very important modeling entities that, unfortunately, are not supported by the conventional object model that is generally used to implement the analysis and design model" [Bos94]. In response to the lack of uniformity of the object model Jan Bosch defines relationships as first class entities. However, Bosch's relations are modeled as layers encapsulating the object whose behavior is influenced by the relation; relations are not separate entities next to objects as in FLO. Moreover, as there is no *compensating message* as in FLO, we consider that Bosch's relations are not reactive.

The MVC model [KP88] uses the Smalltalk *dependencies* which are based on message propagation between objects. In the MVC model, when a model changes (changed method), it broadcasts a change message (udpates method) to its dependents. At first glance, the philosophy of the MVC model, which was the clear independence of the different agents, is respected. However, the use of the Smalltalk *dependencies* has several drawbacks. The programmer must know *a priori* which objects are susceptible to being linked. He/she must subclass some classes only to implement the reacting behaviors. Indeed, the programmer must manage all the state modifications of the model and the reaction of its dependents: he/she must program the change notifications in all the necessary methods of the related objects (adding the self changed message and programming the update methods). Furthermore, a class of dependent objects is specific to a class of model objects: contrary to the initial MVC philosophy, these classes are strongly linked. Moreover, from a specification point of view, the Smalltalk dependencies are spread across all the classes: *"Protocols are sometimes difficult to understand because one has to browse the whole class library to track the message flow"* [BBB93]. The advantage of this approach is that the model does not explicitly know or refer to its dependents.

To make integration and evolution of tools easier, Sullivan and Notkin in [SN92] propose a model based on *mediators*, separate components designed to integrate independent tools, and implicit invocation. A component announces some events, the other components can register an interest in such events by associating procedures. Thus when one event is announced, the system itself invokes all the registered procedures of that event. This model can ensure that update methods are invoked only when necessary.

In order to mediate object collaboration, Pintado in [Pin93] proposes *gluons*. However, his approach of object collaboration is not based on dependencies between cooperating objects, as our model is. In fact, he emphasizes collaboration between objects as *exchanges of services*, which provide additional flexibility such as type conversion, by interposing an object, a *gluon*, between the provider of a service and its clients. Contrary to our approach, an object sends a message to a *gluon* in order to have services provided by another object.

In KSL [IBC91] a pure object-oriented language, the authors emphasize the importance of *ad-hoc instance specialization*. In particular, they introduce the notion of *traps* on instances: KSL *traps* can alter or add any type of behavior. Thus, KSL, as FLO, supports unique instance behaviors without implementing delegation or sacrifiing code sharing. However, FLO provides a mechanism for dependency expression and management. Moreover, FLO links are reified in objects that represents all the dependency semantics. KSL traps can be implemented in FLO using unary link (dependency on single object).

The RENDEZVOUS [HBR$^+$94] system is based upon CLOS, which authors have extended with a constraint maintenance system and an event-based scheduling to process user input. The Abstraction-Link-View paradigm [Hil92] emphasizes a clean separation of user interfaces from applications. ALV *links* are objects whose sole responsibility is to facilitate communication between abstraction objects (application) and the view objects (user interfaces). ALV *links* are bundles of constraints that maintain consistency between views and abstraction. No communication support is coded into the view or abstraction objects: they ignore each other. There are many similarities between RENDEZVOUS and FLO: we use FLO links to ensure communication between abstraction and presentation in the PAC model [Cou87, DF94]. However, FLO allows one to control any method, whereas RENDEZVOUS limits link definition to instance variable. Both approaches are valuable but we consider FLO one enforces encapsulation.

Frolund and Agha [FA93] propose *synchronizers* which allow the coordination of multi-objects in a concurrent and distributed language. *Synchronizers* are, as links, expressed with interfaces of objects, strengthening modularity. There are two important differences between *synchronizers* and *links*: first, *synchronizers* propose specific operators for the synchronization of distributed messages such as the *atomic* operator; second, the proposed operators *updates* and *disables* only take synchronizer's states into account and not the state of the related objects as with our *implies* and *permitted-if* operators.

In order to express cooperation between objects, Helm *et al.* in [HHG90] propose *contracts*. *Contracts* are specified through type obligations, which define variables and external interfaces to be supported, and causal obligations, which define a sequence of messages to be sent and an invariant to be maintained. However, our approach differs significantly from that of contracts. Indeed, with a contract, classes of related objects are structured by and around relationships. To quote the authors, *"the specification of a class becomes spread over a number of contracts and conformance declarations, and is not localized to one class definition"* [HHG90]. On the contrary, our approach emphasizes the equal importance of relationships and of the objects they relate.

On the opposite side of our approach, in languages based on constraint solvers, dependencies are expressed in terms of constraints between instance variables. When the value of such a constrained variable is modified, a propagation algorithm tries to satisfy the constraint, modifying linked variables ([FBB92, MGZ92, San93]). However, constraints are not expressed in terms of object interactions, so some inter-object dependencies are difficult to express as constraints between instance variables. Moreover, some limitations on types of components are imposed by the constraint solver. Finally, as Wilk said in [Wil91], *"Encapsulation was violated by the constraint expressions"*.

5 Conclusion

FLO language is an object-oriented language, implemented with STKLOS [Gal94], integrating the concept of dependencies in a declarative way. The user can define dependencies, declare dependencies between objects and the language automatically manages their consistency. Towards dependencies, the behavior of related objects is changed (methods are controlled). Moreover it is possible to associate new behaviors to normal object behavior through a dependency. Therefore, behaviors of the related objects are enriched. As Agha said *"...the behaviors of objects depend on the context in which they exist"* [FA93]. In FLO, this context is given by the dependencies on objects.

The language is based on the reification of dependencies, control of message passing, and a small open protocol [DF93]. As dependencies are expressed in terms of the object interfaces, and as they can dynamically be added or removed without interfering with the object implementation, modularity is strengthened. Moreover, the code of related objects is kept pure: no relational information is spread across the classes of the related objects. On the contrary, as links are first class objects, they contain this relational information. In such a way, dependency semantics is more clear and easily documented. Moreover, knowledge about dependencies can be expressed, in particular links between links. The use of these "meta-links" increases the robustness of the applications and makes the programming easier.

During the implementation, we were faced with two problems. First, how can we integrate maintenance of dependency consistency in a class-based language, in a way that only messages to related objects are controlled? Second how can we provide a dependency consistency management with extension possibilities? The response is based on the reification of the message sending algorithm. The use of meta-objects allows: a clear separation of objects and dependency functionalities, a control and extensibility of the consistency management and a minimal control of sent messages (only related objects are controlled).

Some critical points of FLO have been highlighted in this paper. The extensibility of FLO responds to some of them. We have shown in this paper for example that adding expressiveness in the dependency declaration is possible due to a new operator definition and management. We have also extended the control of the dependency graph by implementing some new propagation algorithms. To improve the FLO language, some extensions should be considered for future work. We can cite five of these research axes:

- FLO is based on control of message passing. Only one object, the receiver of the message, is controlled. As some systems, like CLOS [Pae93], STKLOS [Gal94] or DYLAN, provide generic functions which are applied to a set of objects instead of message passing on a single object, we would like to investigate if our approach is able to manage dependencies with systems based on generic functions instead of message passing.
- As dependencies tie instances together, we foresee the evaluation of our model in a prototype language with a meta-object protocol such as MOOSTRAP [MC93].
- The success of our implementation is based on the correspondence between the dependencies on the objects and the be-

havior of their meta-objects. This remark outlines the need to investigate the composition between meta-objects [MMC].

– We plan to reify the heritance mechanism of FLO by means of links. This will give FLO a uniform and reflective definition only based on two concepts: objects and dependencies. Inheritance will not be a built-in language mechanism any more, but will become described by the language itself: inheritance will be managed as other dependencies.

– A static analysis of the link graph is important in order to detect problems due to the global control of dependencies more precisely and rapidly. The semantics of FLO language is described in CENTAUR [Bor87], and we are now working on static detection of cycles and computation of reaction chains [San93].

Acknowledgement We would like to thank all the reviewers of this paper, in particular Pierre Cointe and Pierre Berlandier.

References

[BBB93] Patrick Barril, Mahmoud Boufaida, and Jean-Francois Brette. Class cooperation in a dedicated object system: the force authoring environment. In *Proceedings of the 10th TOOLS International Conference*, pages 115–123, Versailles, Mars 1993.

[BC87] Edwin Blake and Steve Cook. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, LNCS 276, pages 41–50, Paris, France, June 1987. Springer Verlag.

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89*, pages 1–6, New Orleans, Louisana, October 1989. ACM. Published as ACM SIGPLAN Notices, volume 24, number 10.

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon. Common lisp object system specification, x3j13. Technical Report 88-003, (ANSI COMMON LISP), 1988.

[BELR92] Michael Blaha, William Permerlani Frederick Eddy, William Lorensen, and James Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall, 1992.

[BKK+86] Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA '86*, pages 17–29, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Bor87] P. Borras. Centaur : the system. Rapport de recherche RR-777, INRIA, December 1987.

[Bos94] Jan Bosch. Relations as first-class entities in layom. Not Yet Published. Available on http://www.pt.hk-r.se/ bosch, 1994.

[CFZ94] Bruno Chabrier and Paul Franchi-Zannettacci. Délegation sémantique par contraintes réactives pour les interfaces graphiques. semantic delegation with reactive contraints for graphical interfaces. *Technique et Sciences Informatiques*, 13(4):539–566, 1994.

[Coi87] P. Cointe. Metaclasses are first Class : The ObjVlisp Model. In *OOPSLA'87 Proceedings*, pages 156–165, October 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.

[Coi88] P. Cointe. The ObjVlisp Kernel: A Reflective Lisp Architecture to Define a Uniform Object-Oriented System. In *Meta-Level Architectures and Reflection*, 1988.

[Cou87] Joëlle Coutaz. The construction of user interfaces and the object paradigm. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, LNCS 276, pages 121–130, Paris, France, June 1987. Springer Verlag.

[DF93] Stéphane Ducasse and Mireille 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, October 1993. ACM.

[DF94] AM. Dery and M. Fornarino. Dépendances comportementales et modèle PAC. In *IHM'94*, Lille, December 1994.

[DFP91] AM. Dubois, M. Fornarino, and AM. Pinna. A tool for modelling and reasoning. application to the building energy analysis. In *13th IMACS World Congress on Computation and Applied Mathematics*, Dublin, 1991.

[Duc95] Stéphane Ducasse. Inheritance mechanism reification by means of first class object. In *Proceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in AI*, 1995.

[FA93] Svend Frølund and Gul Agha. A Language Framework for Multi-Object Coordination. In Oscar M. Nierstrasz, editor, *Proceeding of ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360, Kaiserslautern, July 1993. Springer Verlag.

[FBB92] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 268–286, Utrecht, June 1992. Springer-Verlag.

[Fer89] Jacques Ferber. Computational reflection in class based object oriented languages. In Norman Meyrowitz, editor, *Proceedings of OOPSLA'89*, pages 317–326, New Orleans, Louisiana, October 1989. ACM.

[Gal94] Erick 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.

[Gra89] Nicolas Graube. Metaclass compatibility. In Norman Meyrowitz, editor, *Proceedings of OOPSLA'89*, pages 305–315, New Orleans, Louisiana, October 1989. ACM.

[HBR+94] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F.Patterson, and Wayne Wilner. The Rendezvous Architecture and Language for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

[HHG90] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying compositions in object-oriented systems. In *Proceedings of OOPSLA'90*, pages 169–180, Ottawa, October 1990. ACM.

[Hil92] Ralph D. Hill. The abstraction-link paradigm: Using contraints to connect user interfaces to applications. In Penny Bauersfeld, John Bennett, and Gene Lynch, editors, *Proceedings of CHI'92: the Conference on Human Factors in Computing Systems*, pages 335–342, California, May 1992. ACM.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *OOPSLA'93*, pages 411–428, Washington DC, October 1993. ACM SIGPLAN Notices Vol.28, n. 10.

[IBC91] Mamdouh H. Ibrahim, William E. Bejeck, and Fred A. Cummins. Instance specialization without delegation. *Journal of Object-Oriented Programming*, 4(3):53–56, June 1991.

[Ish91] Yutaka Ishikawa. Reflection Facilities and Realistic Programming. *SIGPLAN Notices*, 26(8):101–110, August 1991.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Metaobject protocols - why we want them and what else they can do. In *Object oriented Programming: the CLOS Perspective*, pages 121–133. MIT Press, 1992.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, pages 26–49, August-september 1988.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155. ACM, October 1987.

[Mae88] Pattie Maes. Issues in computational reflection. In D. Nardi P. Maes, editor, *Meta-Level Architectures and Reflection*, pages 21–35. Elsevier Science Publishers B.V. (North-Holland), 1988.

[MC93] P. Mulet and P. Cointe. Definition of a reflective kernel for a prototype-based langage. In Shojiro Nishio and Akinori Yonezawa, editors, *First International Symposium on Object Technologies*, volume 742 of *Lecture Notes in Computer Science*, pages 128–144, Kanazawa, Japan, November 1993. JSSST-JAIST, Springer-Verlag.

[Mey90] Bertrand Meyer. *Conception et Programmation par Objets*. Intereditions, 1990.

[MGZ92] Brad A. Myers, Dario A. Guise, and Brad Vander Zanden. Declarative programming in a prototype-instance system: object-oriented programming without writing methods. In *Proceedings of OOPSLA'92*, pages 185–199, Vancouver, October 1992. ACM.

[MMC] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for composing metaobjects. To appear in the Proceedings of OOPSLA'95.

[NECH92] D. Nanci, B. Espinasse, B. Cohen, and H. Heckenroth. *Ingenierie des systemes d'information avec Merise*. Performance. Sybex, 1992. ISBN: 2-7361-0747-7.

[Pae93] Andreas Paepcke, editor. *Object Oriented Programming: the CLOS perspective*. MIT Press, 1993.

[Pin93] Xavier Pintado. Gluons: a support for software component cooperation. In Shojiro Nishio and Akinori Yonezawa, editors, *First International Symposium on Object Technologies*, volume 742 of *Lecture Notes in Computer Science*, pages 43–60, Kanazawa, Japan, November 1993. JSSST-JAIST, Springer-Verlag.

[San93] M. Sannella. The skyblue constraint solver. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, February 1993.

[SBK86] M. Stefik, D.G. Bobrow, and K. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software (USA)*, 3(1):10–18, Janvier 1986.

[SD94] Ira R. Forman Scott Danforth. Reflections on metaclass programming in SOM. In ACM, editor, *Proceedings of OOPSLA'94*, volume 29 of *ACM Sigplan Notices*, pages 440–452, Portland, October 1994. ACM.

[Smi82] Brian Cantwell Smith. Reflection and semantics in a procedural lnaguage. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[SN92] K.J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *Transactions on Software Engineering and Methodology*, 1(3):228–268, July 1992.

[Sny86] Alan Snyder. Encapsulation and inheritance in object programming languages. In *Proceedings of OOPSLA'86*, pages 38–45. ACM, September 1986.

[SRHB89] A. Shah, J.E. Rumbaugh, J.H. Hamel, and R. Borsari. Dsm: An object-relationship modeling language. In *Proceedings of OOPSLA'89*, pages 191–202, New-Orleans, October 1989. ACM.

[Ste90] G.L. Steele. *Common Lisp The Language, Second Edition.* Digital Press, 1990.

[Wil91] Mickael R. Wilk. Equate: An object-oriented constraint solver. In *Proceedings of OOPSLA'91*, pages 286–298. ACM, 1991.