Protocol for Managing Dependencies between Objects by controlling Generic Function Invocation

Stephane Ducasse and Mireille Fornarino

I3S, Université de Nice Sophia Antipolis - CNRS
250, avenue A. Einstein
Sophia Antipolis
06560 Valbonne, FRANCE

1 Introduction

The design of programming environments, hypertexts, graphical interfaces, and communications between processes all require the representation and management of dependencies between entities (dependencies between an object and its views, liaison between menu and text selection) [KR91, San89, CMP91, Rum92]. Consequently object systems allying objects and constraints have been developed [MGZ92, FBMB90, Ilo91]. For this approach, we need information on object's structures, which goes against principles of object encapsulation [Wil91], and does not allow expression of event-based actions. The standard daemon and active value mechanisms allow implementation of certain dependencies, however there is also a diffusion of information and it is not easy to define new dependencies [FPT89].

In a class-instance model, we allow a programmer to express dependencies between his previously defined objects. For this, we propose a protocol to model and manage dependencies. It is based on an object representation of dependencies (the links) and the control of function calls. This protocol is then an over system of an object system which is not modified by adjunction of dependencies.

The reification of a dependency notion permits grouping in the same entity all of the information which characterizes a dependency (when it is coherent, how to make it coherent, which functions can invalidate it, etc...). Moreover, the expression of dependencies between dependencies such as inverse and incompatibility is then natural; moreover the trace of dependencies maintenance can also be expressed as links. Coherence maintenance of each dependency is managed by a control of function calls implying either limitations in these calls or reactions necessary to re-establish coherence of dependencies. Specification of waited reactions allows an incremental maintenance of the coherence.

Our protocol uses the meta-object protocol [KdRB91]. Its associated functionalities were determined with both minimality and extensibility considerations. Particularly, we have defined link metaclasses which rule the creation of dependencies and set-up the control automatically. A generic function class allows the control of function calls. At this point there were two alternatives: a local control of each function or a global control over the system. The impossibility to dynamically specialize the generic function classes in CLOS, considerations of efficiency and extensibility, as well as the will to supervise the "sensitive" calls brought us to introduce the notion of controller.

As [KdRB91] did, we present here several extensions of our protocol to model different behaviours then those initially forseen.

We have shown the originality of this protocol as well as its use for comprehension and resolution of problems such as design of coherent graphical interface [FP90], expression of the inheritance mechanism of object languages [For91], set-up and exploitation of thermic model library[DFP91], modelization of knowledge networks in acquisition phase [DGTC92] and a study is being made to model communication between entities [CMP91].

2 Dependencies and semantic coherence

One of the prerequisites to this work was respecting the object's encapsulation. We will consider that an object is only accessible by its interface: the whole set of its methods. Therefore the structure of the object is unknown and we can not modify it. Moreover, the expression of dependencies on an object must not modify its interface.

A link is the expression of a dependency between objects[1]. The link defines the dependency semantic. We have limited our study to binary links, the extension to the n-ary links is one of our future goals.

The dependencies we manipulate are oriented. The linked objects do not play the same role. Therefore, a link is established between an **influent** object and a **dependent** object for this link. Thus on Figure 1, c2 is linked as dependent to c1 and as influent to c3.

A link is **coherent** if the semantic of the represented dependency is respected[2]. A link permits the following definitions: is a dependency effectively verified (coherent); how to make it so; which function calls (**influent functions** for the link) on the influent objects oblige a re-establishment of the coherence and how (**reaction**), and which functions on dependent objects must be controled

[1] Dependencies : object-window, scroller-editor, centering, alignment

[2] Objects are well visualized, contained objects are effectively contained, etc

(**guarded functions** for the link). All this information expresses the operational semantic of a dependency and allows an incremental maintenance. The influent functions and the guarded ones for a link constitute a whole set of **sensitive functions** for this one. Consequently, we must control these functions.

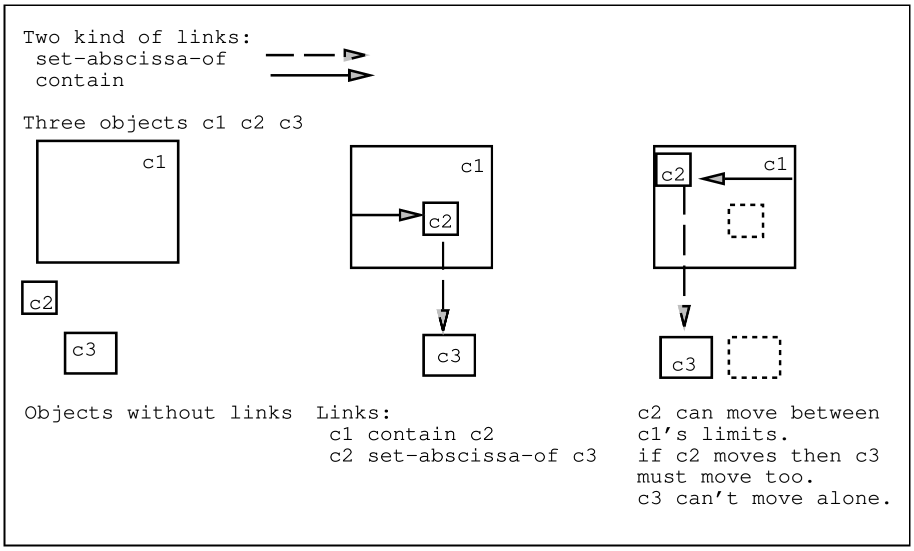

Figure 1: Example of dependencies

An object could be dependent and/or influent on several links. Certain function calls then require: the determination of the links of which one of calling arguments is a dependent object[3] and for which this function is guarded, the validation of the associated guards, the execution of the call, the re-establishment of the coherence of all the links for which one of the calling arguments is an influent object. For this last stage, we will talk of the **firing** of the links. It is important to note that the reaction to a function can fire new links[4].

The order of these updates is made by a controller which is in charge of the management and the maintenance of the all dependencies.

3 Reification of dependencies

Since dependencies have their own characteristics, we have chosen to represent them as first class objects, called **links**. This choice offers several advantages such as the factorisation of information proper to a dependency, specialization

[3] Until now we consider that in call of a guarded (resp. influent) function only one argument is an influent (resp. dependent) object.

[4] In the figure 1, If c1 moves, c2 moves and then c3 too.

of dependencies' families (conditional reaction, automaton links, etc...), facility to define dependencies between dependencies (the link " contain" reverses the link "contained-by", the link "inverse" reverses itself) (cf. 3.2.1)).

We are going to distinguish a conceptual level (class) and an object level (instance). The conceptual level defines the characteristics of a dependency (type of linked objects, changes that can interfere with the coherence of dependency, etc...). In this case, dependency representations are called **links**. At the object level, **instance-of-link** effectively establishes a dependency between two given objects.

The model we submit here may be described in three ways:

- Declarative: Definition of a link and creation of instances-of-links (cf. 3.2),
- Operational: What are the results of creating a dependency between two objects (cf. 3.3),
- Implementational: Exploitation and control of object mechanisms (cf. 3.4).

The following is a global description of the protocol. We shall then detail the three levels mentioned above.

3.1 Architecture

Our protocol is based on the following notions (cf. Figure 2):
1. *links*, used to define dependencies (*link hierarchy*).
2. *metalinks* define the structure of different types of links and set their control (*metalink hierarchy*).
3. *controllers* effectively call the control procedures and manage interactions between instances-of-links (*controller hierarchy*).
4. *generic-functions* which call current controller (*controled-generic-function*).

As does the metaobject protocol, our protocol is intended for two types of users: the one who wishes to define standard simple dependencies, who only has to define new links (cf. 4.1); the one who needs to define new non standard dependencies, who has to specialize offered functionalities (cf. 4.2).

3.2 Characteristics of a link

A link specifies the semantic of a dependency. Thus, definition of a link includes two functions. The first one, **coherent-p**, specifies when an instance-of-link is coherent. The second, **do-coherence**, specifies how to make it coherent. This definition also includes specifications about influent functions and associated reactions, and guarded functions and associated guards (cf. Figure 3).

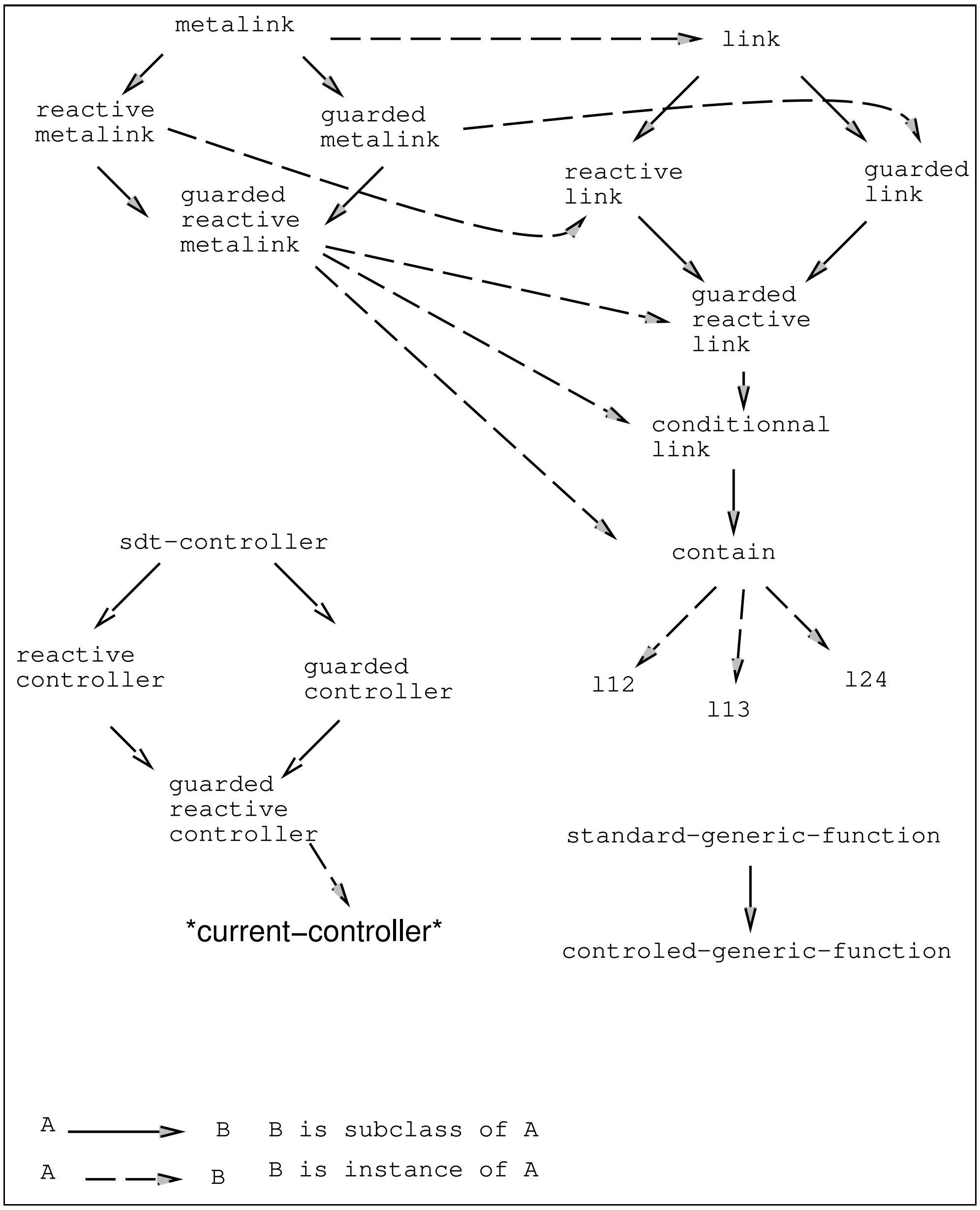

Figure 2: Hierarchies of classes of our protocol

```
(deflink CONTAIN
  :is conditional-link
  :action-reaction-list
  (((move-to (influent x y)) . (move-to (dependent x y)))
   ((translate (influent x y) . (translate (dependent x y))))
  :action-guard-list
  (((move-to (dependent x y) .
                       (cond-contient-move (dependent influent x y)))
   ((translate (dependent x y)) .
                       (cond-contient-translate (influent dependent x y))))
  :coherent-p coherent-contient-p
  :coherence coherence-contient)

; creation of link's instance between objects c1 and c2
(setq lc12 (make-instance 'contient :dependent c2 :influent c1))

; destruction of link's instance
(remove-link lc12)
```

Figure 3: An example of link definition

Some dependencies only involve influent functions and others only involve functions to be guarded, though we have distinguished these two types of links (classes *Guarded-Metalink* and *Reactive-Metalink*) (cf. Figure 4).

```
(defclass METALINK (STANDARD-CLASS) ())

(defclass REACTIVE-METALINK (METALINK)
  ((action-reaction-list
   :initarg :action-reaction-list
   :accessor action-reaction-list
   :initform ())))

(defclass GUARDED-METALINK (METALINK)
  ((action-guard-list
   :initarg :action-guard-list
   :accessor action-guard-list
   :initform ())))

(defclass GUARDED-REACTIVE-METALINK
     (GUARDED-METALINK REACTIVE-METALINK)
   ())
```

```
(defclass LINK ()
  ((influent
   :initarg :influent
   :accessor influent)
   (dependent
   :initarg :dependent
   :accessor dependent))
  (:metaclass METALINK))

(defclass REACTIVE-LINK (LINK) ()
  (:metaclass REACTIVE-METALINK))

(defclass GUARDED-LINK (LINK) ()
  (:metaclass GUARDED-METALINK))

(defclass GUARDED-REACTIVE-LINK
          (GUARDED-LINK REACTIVE-LINK) ()
  (:metaclass GUARDED-REACTIVE-METALINK))
```

Figure 4: Hierarchies of links and metalinks

The expression of the sensitive functions, as well as guards and reactions which are associated to them, is completed by **frameworks** (sequence of symbols featuring "link", "influent" and "dependent" which indicate respectively an instance of a link, the influent object and the dependent object of this instance). Frameworks allows us to do liaisons between arguments of the call[5], and the arguments of guard and/or reaction calls.

[5]Current implementation does not consider CommonLisp's keywords

The **deflink** macro makes the definition of new links easier. By default, these are then *Guarded-Reactive-Metalink* instances and *Guarded-Reactive-Link* subclasses.

An instance-of-link is simply defined by its class, the influent object and the dependent one (*link* class). An instance-of-link creation ensures automatic management of its coherence. The user only has to declare his dependencies, so that they can be maintained. He can also dynamically destroy them (*remove-link*).

The *influent-links* (resp. *dependent-links*) function allows an object to determine all the instances-of-links on which it is influent (resp. dependent).

3.2.1 A link between links: inverse-link

We often express dependencies between links. For example, the expression of an inverse-link between two links: if l^{-1} is the inverse of l, then it suffices to say that x is linked to y by l^{-1} to imply that y would be linked to x by l. Particularly, we can specify that the inverse of the inverse is the inverse. Consequently, saying l^{-1} is the inverse of suffices to define l as the inverse of l (cf. Figure 5).

```
(deflink inverse-link
  :action-reaction-list
  (((create-link (influent xi xd)) . (create-link (dependent xd xi))))
  :action-guard-list
  (((create-link (dependent xi xd)) . (cond-not-exist (dependent xi xd)))))

(defmethod create-link ( (Link METALINK) Influent Dependent)
   (make-instance Link :influent  Influent :dependent Dependent ))

(create-link (find-class 'inverse-link) (find-class 'inverse-link)
            (find-class 'inverse-link))

(create-link (find-class 'inverse-link) (find-class 'Father-of)
        (find-class 'Son-of)))
;; Implies Son-of inverse of Father-of

(create-link  (find-class 'Father-of) adam caen)
;; Implies caen son-of adam
(create-link  (find-class 'Son-of) abel adam)
;; Implies adam Father-of abel
```

Figure 5: Link between links: inverse-link

The existence of an inverse-link allows us to model bidirectional links. If we say the inverse of the *contain-link* is the *contained-link*, we can express that a move of the contained object implies, if necessary, an extending of the container, whereas a move of the container implies, if necessary, a move of the contained. In such cases, guards are no longer necessary.

3.2.2 Link and trace system

To trace a link or an instance-of-link simply consists of associating a trace as a reaction of maintenance methods of coherence (methods defined on links or metalinks). To do this, we define a new link *trace-link* (cf.Figure 6). The dynamic declaration of this instance-of-link between instances-of-link or classes of link and *trace-object* allows us to trace these links or instances-of-link and dynamically eliminate this trace. In particular, it is possible to trace instances of the link *trace-link* itself. Tracing the class *trace-link* itself leads to infinite recursion.

```
 (deflink trace-link
   :action-reaction-list
   (((firing (influent gf-name args)) .
                            (trace (dependent influent gf-name args)))
    ((firing-using-class (influent l gf-name args)) .
                            (trace-cl (dependent influent l gf-name args)))))

(setq set-abs-1-2 (make-instance 'set-abscissa-of :influent c1 :dependent c2))
(setq set-abs-3-4 (make-instance 'set-abscissa-of :influent c3 :dependent c4))

(make-instance  'trace-link :influent set-abs-1-2 :dependent object-trace)
;; for trace link set-abs-1-2

(change-x c1 10000)
> "Change-x on c1 implies change-x on c2"

(change-x c3 10000)  ;; No trace

(make-instance  'trace-link :influent (find-class 'set-abscissa-of)
                :dependent object-trace))
;; for trace class link 'set-abscissa-of

(change-x c3 10000)
> "set-abscissa-of between c3 and c4 implies reaction on c4"
```

Figure 6: Link trace by links

3.3 Controlling the function calls

Class of generic functions: Every call to sensitive functions must be checked. This is the purpose of the generic function class *Controled-generic-function*[6] which redirects any call to its instances toward a current controller (a call to the generic function *control*)(cf. Figure 7).

The impossibility of dynamically modifying a generic function class prohibited us from having a specific control according to the nature of the fonction (standard, guarded and/or influent).

[6]In Clos, for such functions, we have to specialize the *compute-discriminating-function* method. Moreover the specialization of *ensure-generic-function-using-class* allows us to oblige all new generic functions to be of this class.

```
(defclass CONTROLED-GENERIC-FUNCTION (STANDARD-GENERIC-FUNCTION) ()
  (:metaclass FUNCALLABLE-STANDARD-CLASS))
(defmethod compute-discriminating-function  ((cgf controled-generic-function))
  (let ((normal-dfun (call-next-method)))
    #'(lambda (&rest args)
 (control *current-controller* cgf normal-dfun args))))
```

Figure 7: Controled-generic-function class

Controller: While calling the *control* method, the controller determines if the call of the function must be checked (sensitive functions and existence of link's instances concerned by the call). If ckecking is unnecessary there is then simply execution of the generic function. Otherwise, the function can be sensitive to several links. If the function is guarded, the control consists in checking the whole group of guards before authorizing the execution of the call. If the function is influent, there must also be execution of the different reactions. Often, functions are of both types, requiring both actions. These two aspects of the control are done by *combine-firing* and *combine-guards* methods. By default, the order chosen for guards verification and reaction firing is the same as the instance-of-link creation order. We can easily imagine, by simple specification, a new guard combination or expression of priorities between links (cf. 4.2.1).

```
(defclass STANDARD-CONTROLLER () () (:metaclass ONLY-ONE-INSTANCE))
(defclass REACTIVE-CONTROLLER (STANDARD-CONTROLLER)
  ((gf-frameworks-of-action :accessor gf-frameworks-of-action
   :initform (make-hash-table :test #'equal))
   (reactive-table :initform firing-hash :accessor reactive-table))
  (:metaclass ONLY-ONE-INSTANCE))
(defclass GUARDED-CONTROLLER (STANDARD-CONTROLLER)
  ((gf-frameworks-of-guard :accessor gf-frameworks-of-guard
   :initform (make-hash-table :test #'equal))
   (guard-table :initform guard-hash :accessor guard-table))
  (:metaclass ONLY-ONE-INSTANCE))
(defclass GUARDED-REACTIVE-CONTROLLER (GUARDED-CONTROLLER REACTIVE-CONTROLLER)
 (:metaclass ONLY-ONE-INSTANCE))

(defmethod control ((sc STANDARD-CONTROLLER)
                    (cgf CONTROLED-GENERIC-FUNCTION) normal-dfun args)
  (apply normal-dfun args))
(defmethod control :after ((ac REACTIVE-CONTROLLER)
                    (cgf CONTROLED-GENERIC-FUNCTION) normal-dfun args)
  (firing-reaction ac cgf args))
(defmethod control :around ((gc GUARDED-CONTROLLER)
                    (cgf CONTROLED-GENERIC-FUNCTION) normal-dfun args)
 (when (guard gc cgf args)
 (call-next-method)))
```

Figure 8: Hierarchy of controllers

By default, we propose to work with the instance controller of the class *guarded-reactive-controller* (cf. Figure 8)). This manages at the same time the guarded and influent functions. However in certain applications only one aspect of the control occurs, the user then has the possibility to define a more restrained current controller. In addition, he may want to manage new behaviours and use a more specialized controller (cf. 4.2).

Guards and Reactivity: Since we want to preserve, in a link, the factorization of dependency information and also to favour specialization of coherence maintenance with regard to links, the *combine-firing* and *combine-guards* methods of the controller call *verify-guard* and *firing* methods of links. The guards verification can depend on the state of a link. For example, we may associate an automaton to an instance of a link, or we may associate a conditional reaction (cf. 4.1).

As information about influence and guard of functions is associated to class of link, *verify-guard-using-class* and *firing-using-class* method of metalinks control the execution of *verifiy* and *guard* methods.

3.4 Control set-up

We distinguish two stages for the control set-up: first, the definition of a link (class) where certain functions become sensitive, and then instance-of-link creation where dependency is established. The association of a metaclass *metalink* to link allows us to manage these two aspects.

Sensitive functions and controller: When a link has been defined, the setting of its slots *action-reaction-list* and *action-guard-list* by a list of pairs of frameworks specifies its guarded and influent functions.

The association of daemons on accessors of these slots and the initialization of a link (cf. Figure 9) allow us to automatically store this information in the controller, in order to limit control to these functions (slots *gf-frameworks-of-guard* and *gf-frameworks-of-action*).

```
(defmethod initialize-instance :after ((gl GUARDED-METALINK) &rest reste
           &key (action-guard-list () p)
           &allow-other-keys)
  (setf (action-guard-list gl) (car action-guard-list)))

(defmethod initialize-instance :after ((gl REACTIVE-METALINK) &rest reste
         &key (action-reaction-list () p)
         &allow-other-keys)
  (setf (action-reaction-list gl) (car action-reaction-list)))
```

Figure 9: Initialize-instance

Linking of two objects : The redefinition of the instance-of-link mechanism creation (*make-instance* method - cf. Figure 10) allows on the one hand the establishment of dependency (by eventually pointing out dependent object so that the coherence of this new instance-of-link might be checked), and on the other hand the recording of the existence of this new instance (*register-link* method defined on *metalink*). The proposed implementation consists then of warning the controller of the existence a new instance-of-link (slots *reactive-table* and *guard-table* and *register-reactive-link* and *register-guard-link* methods).

```
(defmethod make-instance ((ml METALINK) &rest reste)
  (let* ((new-link (call-next-method)))
    (unless (and (dependent link) (influent link))
     (error " dependent and or influent must be specified"))
    (unless (coherent-p new-link) (do-coherence new-link))
    (unless (coherent-p new-link) (coherence-error new-link))
    (register-link ml new-link)
    new-link))
 (defmethod register-link :after  ((ml REACTIVE-METALINK) (new-link REACTIVE-LINK))
   (register-reactive-link *current-controller* new-link
                          (action-reaction-list ml) (influent new-link) ) )
  (defmethod register-link :after  ((ml GUARDED-METALINK) (new-link GUARDED-LINK))
    (register-guarded-link *current-controller* new-link
                           (action-guard-list ml) (dependent new-link) ) )
```

Figure 10: Make-instance

4 Extensions of our model

The minimal core described above, including hierarchies of links, metalinks, controllers, and their specific methods, has been extended in various directions according to the desired goal. Thus we are going to distinguish specific extensions of the model of links and extensions related to the global protocol.

4.1 Extensions of hierarchy of links

We had to design various kinds of links according to the different applications we designed (graphical interfaces, modelization of inheritance, map coloring, hypothesies management). We chose to describe here two of the most used links.

4.1.1 Conditional reaction

By default, the execution of an influent function always implies the execution of the associated reaction. However, concerning some links, reaction can be executed only under specific conditions (lack of coherence of the link, non operating window, etc...). In order to do this, we only have to specialize the *firing* method associated to the link (cf. Figure 11).

```
(defclass CONDITIONAL-LINK (GUARDED-REACTIVE-LINK) ()
  (:metaclass GUARDED-REACTIVE-METALINK))

(defmethod condition-p ((l CONDITIONAL-LINK))
  (coherent-p l))

(defmethod firing ((l CONDITIONAL-LINK) gf-name args)
  (unless (condition-p l)
       (call-next-method)))
```

Figure 11: Link with a conditional reaction

The link *contain* described on Figure 3 is in fact a conditional link. The various moves of reactions occur only if necessary.

4.1.2 Links with detection of dynamic cycles

When the coherence of a link is re-established, cycles may occur as a result of other links' existence. We consider that a dynamic cycle occurs during the reaction phase of the execution of an influent method only if this call is reiterated while the firing is still running. For the detection of cycles, we remember, in the fired instance-of-link, the sequence of calls to influent functions which occured during the re-establishment of coherence. To do this, we enrich the class of links and we specialize the *firing* method (cf. Figure 12).

```
(defclass REACTIVE-LINK-WITHOUT-CYCLES (GUARDED-REACTIVE-LINK)
  ((already-firing :initform () :accessor already-firing))
  (:metaclass GUARDED-REACTIVE-METALINK))

(defmethod firing ((lk REACTIVE-LINK-WITHOUT-CYCLES) gf-name args)
    (unless (member  gf-name  (already-firing lk))
       (push gf-name (already-firing lk))
       (call-next-method)
       (pop (already-firing lk))))
```

Figure 12: Links with dynamic cycles detection

4.2 Extensions of the protocol

In the extensions above, the creation of extensions required only the specialization of functions associated to links. Others extensions need new controllers or new classes of metalinks.

4.2.1 Hierarchic links

In this part, we associate priorities to links. These priorities allow us to control the order of firing and to manage conflicts between guards.

Ordered firing: By default, the instances-of-links are fired according to the ordering of the declaration. In some cases it may be necessary to modify this order if, for example, we consider the weight associated to a link. In order to introduce this functionality, we need on the one hand, to define a new class of link having a new slot *firing-priority* and, on the other hand, a new controller which specializes the *combine-firing* method (cf. Figure 13). The specialization of this method allows us to carry out any other ordering, for example, firing selectively the highest priority links only.

The specialization of our *order-for-firing* method allows us to memorize the firing ordering in the case where this ordering may be determined statically, thus avoiding useless sorts[7].

```
(defclass FIRING-PRIORITY-CONTROLLER (GUARDED-REACTIVE-CONTROLLER) ()
 (:metaclass ONLY-ONE-INSTANCE))

(setq *current-controller* (make-instance 'FIRING-PRIORITY-CONTROLLER))

(defclass ORDORED-FIRING-LINK (GUARDED-REACTIVE-LINK)
  ((firing-priority :initarg :firing-priority :initform 0 :accessor firing-priority))
  (:metaclass GUARDED-REACTIVE-METALINK))

(defmethod combine-firing ((ppc FIRING-PRIORITY-CONTROLLER) llk gf-name args)
   (call-next-method ppc (sort llk #'(lambda (x y) (< (firing-priority x)
                                   (firing-priority y))))
                     gf-name args))
```

Figure 13: Links with ordered firing

Hierarchic guards: An object may be dependent on several instances-of-links. In this case, the resulting guard (which specifies if a given action on this object is possible or not) requires that each guard of instances-of-links, having this object as dependent, has been verified. Contradictions may emerge between guards. By default, if this happens, we signal an error. An alternative way of detecting contradictions consists in stating priorities between links. Only the highest priority guard(s) should then be verified and incoherence may then be accepted [BFBW92].

To do this, we enrich our links with a priority and we define a new controller which achieves the new guards combination (*combine-guards*) and remembers sorted instances-of-links (*register-guarded-link*) (cf. Figure 14).

4.2.2 A new composition of guards

When an object is included in another, its moves are limited in space. If an out-of-limit displacement is required, it is ordinarily rejected. In this extension, such a move is allowed, up to the influent object's limits (cf. Figure 15).

[7] It might necessary to recalculate the link's weight for each time a firing occurs, for example, considering the existence of environment variables.

```
(defclass GUARDED-PRIORITY-CONTROLLER (GUARDED-REACTIVE-CONTROLLER) ()
  (:metaclass ONLY-ONE-INSTANCE))
(setq *current-controller* (make-instance 'GUARDED-PRIORITY-CONTROLLER))
(defclass ORDORED-GUARD-LINK (GUARDED-REACTIVE-LINK)
  ((guard-priority :initarg :guard-priority :initform 0 :accessor guard-priority))
  (:metaclass GUARDED-REACTIVE-METALINK))
(defmethod combine-guards ((gpc GUARDED-PRIORITY-CONTROLLER) llk gf-name args)
  (call-next-method gpc
              (get-links-having-priority  llk
                         (guard-priority (car llk)))
              gf-name args)))
(defmethod register-guarded-link ((gpc GUARDED-PRIORITY-CONTROLLER)
                                   (new-link GUARDED-REACTIVE-LINK) agl dependent)
...)
```

Figure 14: Links with priority on guards

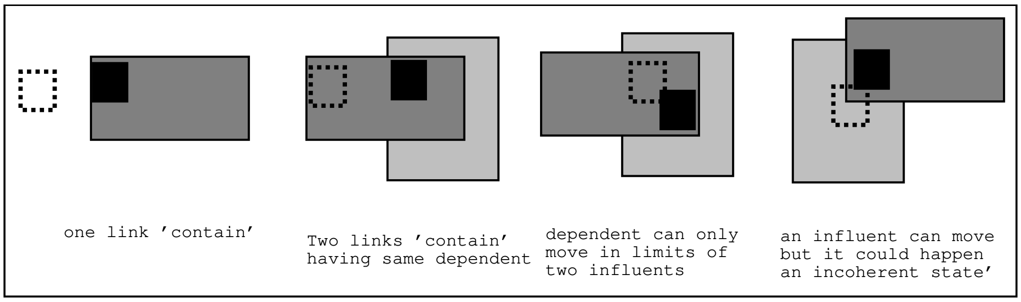

Figure 15: Example of combination: composition of guards

We authorize the modification of the calling arguments of a guarded function, so that these arguments are coherent with the guard. If several guards are encountered, these guards must undergo a composition.

Thus, guards will not simply give out a boolean value, but a list of arguments for which action is possible. Firing will be initiated from the new arguments. Therefore the action will not be completely forbidden.

In Figure 16), a new controller redefines the guard combination function (*combine-guards*) as a composition of functions, and specializes the *control* method in order to communicate the result of this composition to the reaction method. Concerning the *control* method, a novel *one-around* qualifier had to be introduced. In fact, only the most specific around method as well as all the primary and after methods must be executed.

```
(defclass COMPOSED-GUARDED-CONTROLLER (GUARDED-CONTROLLER) ()
  (:metaclass ONLY-ONE-INSTANCE))
(defclass COMPOSED-GUARDED-REACTIVE-CONTROLLER
                 (COMPOSED-GUARDED-CONTROLLER REACTIVE-CONTROLLER) ()
  (:metaclass ONLY-ONE-INSTANCE))
(setq *current-controller*
    (make-instance 'COMPOSED-GUARDED-REACTIVE-CONTROLLER))
(defmethod control :one-around ((cgc COMPOSED-GUARDED-CONTROLLER)
                  (cgf CONTROLED-GENERIC-FUNCTION)
                  normal-dfun args)
   (let ((new-args (guard cgc cgf args)))
     (if new-args
       (call-next-method cgc cgf normal-dfun new-args)
       (format t "~% REFUS EXECUTION" ))))
(defmethod combine-guards ((cgc COMPOSED-GUARDED-CONTROLLER) llk gf-name args)
    (verify-composed llk gf-name args))
```

Figure 16: Composition of guards

4.2.3 A new meta-behaviour

Until now any action over the influent object of an instance-of-link implied a reaction on the dependent one. For some dependencies, the influent and dependent object are the same, so an action requires an action on itself[8].

To do this, we create a new controller *Reactive-Dependent-Controller* which manages two tables (one for the frameworks, one for instances-of-links), a new class of link *Reactive-Dependent-Link*, and the associated metalink *Reactive-Dependent-Metalink*. This metalink carries out table management *register-link* as well as firing on dependents (cf. Figure 17).

[8] While a text is centered in an editor window, if the text is scaled, it must be re-centered.

```
(defclass REACTIVE-DEPENDENT-METALINK (METALINK)
  ((action-reaction-dependent-list
      :initarg :action-reaction-dependent-list
      :accessor action-reaction-dependent-list
     :initform ())))

(defclass REACTIVE-DEPENDENT-LINK (LINK) ()
  (:metaclass REACTIVE-DEPENDENT-METALINK))

(defclass REACTIVE-DEPENDENT-CONTROLLER (STANDARD-CONTROLLER)
  ((gf-frameworks-of-action-dependent
     :accessor gf-frameworks-of-action-dependent
     :initform (make-hash-table :test #'equal))
   (reactive-dependent-table
     :initform firing-dependent-hash
     :accessor reactive-dependent-table))
  (:metaclass ONLY-ONE-INSTANCE))

(defmethod register-link :after
    ((ml REACTIVE-DEPENDENT-METALINK) (new-link REACTIVE-DEPENDENT-LINK))
  (register-reactive-dependent-link *current-controller new-link
                              action-reaction-dependent-list dependent)) )

 (defmethod control :after ((ac REACTIVE-DEPENDENT-CONTROLLER)
       (cgf CONTROLED-GENERIC-FUNCTION) normal-dfun args)
   (firing-reaction-dependent ac cgf args))
```

Figure 17: A new meta-link: an action on dependent implies reaction on itself.

5 Conclusion and Perspectives

We propose a protocol for the expression of dependencies. Within the framework of the metaobject protocol in CLOS, our protocol is extensible and fully integrated to the object model. Since it uses meta-classes and meta-behaviour, it is addressed to two types of users: One who is willing to express the dependencies, which are managed in an incremental and automatic way; and the other who wishes to define new behaviour on his dependencies. In the first case, we have to define new classes and create instances, in the second one we have to work on metaclasses of links. The dependencies modelization as a full object allows the creation and dynamic withdrawal of instances-of-link, as well as dynamic modifications of a link (addition, withdrawal, and modification of the pairs of functions defining the semantic of a link). A simple generation of daemons on the sensitive functions is tedious and involves method duplication and more difficult management of dependency addition and withdrawal.

Moreover the expression of dependencies between instances must either be considered at the class level of the linked objects or use *eql* specializers, which is not recommended if the number of dependencies is high.

The realization of several applications, the works on constraint systems, and the advanced works on reflexivity [Mae87, Coi87, Fer89, MC93] of languages have at the same time supported the validity of our approach and open several routes of investigation toward which we are going.

- The management of the incoherences is at the present time poorly elaborated and not really integrated into the protocol. That is one of the priorities of our work.

- The links instances are numerous. It is then very important to optimize their structures.

- The definition of a link between several influent and/or dependent objects sets ordering and delay problems. We work currently for this extension of our protocol which seems to bring us back to problems of parallelism and distributivity.

- There is a narrow dependency between the class of the link and its associated controller. This dependency can be expressed with links. This approach would allow then the simultaneous management of several controllers, eventually linked together. Thus, we could implant some delays according to the controller state.

- The coherence function associated to a link seems sometimes able to be assimilated to declaration of the instance of a link. One of the ways in which we plan to continue our study is with a representation of the links by special generic functions. These functions would have most of the characteristics of our current links and could be put together, at least conceptually, by first ordered logical predicates. We hope thus to increase the declarative aspect of the protocol and its integration into the object model.

References

[BFBW92] A. Borning, B. Freeman-Beson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic computation*, (5):223–270, 1992.

[CMP91] D. Clement, F. Montagnac, and V. Prunet. Integrated software components: a paradigm for control integration. In *Proceedings of the European Symposium on Software Development Environments and CASE Technology*, June 1991.

[Coi87] P. Cointe. Metaclasses are first class : The ObjVlisp model. In *OOPSLA'87 Proceedings*, pages 156–165, October 1987.

[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.

[DGTC92] R. Dieng, A. Giboin, P.-A. Tourtier, and O. Corby. Knowledge Acquisition for Explainable, Multi-Expert Knowledge-Based Design Systems. In *Current Developments in Knowledge Acquisition : EKAW-92*, Heidelberg, 18-22 May 1992.

[FBMB90] B.N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In Norman Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 317–326, New Orleans, Louisiana, October 1989. ACM. Published in Sigplan Notices Volume 24, Number 10.

[For91] M. Fornarino. Mise en oeuvre de l'héritage au moyen de relations. Rapport de recherche 1417, INRIA, Avril 1991.

[FP90] M. Fornarino and A.M. Pinna. Un modèle objet logique et relationnel. le langage othelo. Thèse, Université de Nice - Sophia Antipolis, INRIA Sophia Antipolis, Avril 1990.

[FPT89] M. Fornarino, AM. Pinna, and B. Trousse. An original object-oriented approach for relation management. In *Lecture Notes on Artificial Intelligence*. Springer-Verlag, 1989.

[Ilo91] Ilog. *PECOS,Programmation par contraintes orientée objets*, Avril 1991. Version 1.0.

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

[KR91] J. Konstan and L. Rowe. Developing a guide using object-oriented programming. In *OOPSLA'91 Proceedings*. ACM, 6–11 October 1991.

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

[MC93] Ph. Mulet and P. Cointe. Définition d'un noyau réflexif pour un langage à prototypes. In *Representations par objets*, La grande Motte, 17-18 Juin 1993. EC2.

[MGZ92] Brad A. Myers, D.A. Giuse, and B. Vander Zanden. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. *ACM Sigplan Notice, OOPSLA'92*, 27(10):184–200, October 1992.

[Rum92] J. Rumbaugh. Horsing around with associations. *Journal of Object Oriented Programming*, 4(6):20–29, February 1992.

[San89] Elmer Sanvad. Hypertext in an object-oriented programming environment. Technical Report DAIMI PB - 280, AARHUS University, Computer Science Department, Denmark, May 1989.

[Wil91] M.R. Wilk. Equate: An object-oriented constraint solver. In *OOPSLA'91 Proceedings*, pages 286–297. ACM, 6–11 October 1991.