Object and Dependency Oriented Programming in FLO

Anne-Marie Dery and Stéphane Ducasse and Mireille Fornarino

{pinna,ducasse,blay}@essi.fr
Université de Nice Sophia-Antipolis - CNRS URA 1376
650 route des colles - B.P. 145, 06903 Sophia Antipolis CEDEX - FRANCE

Abstract. The FLO language integrates management of inter-object dependencies into the object oriented paradigms. In this paper, we focus on the use of reactive dependencies (*links*) in object-oriented knowledge representation. In particular, we present different *meta-links* (*links* between *links*) and show how the FLO links allow one to design some composition relationships.

1 Introduction

Although the importance of inter-object relationships (also named dependencies or interactions) is well accepted [BC89, BELR92], 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. 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. This goes against modularity. It is then difficult to modify, specify, or maintain objects and dependencies. Reuse capabilities also decrease.

In response to this lack of expressiveness of object models, the FLO[1] [DBFP95] 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. Moreover, FLO is designed as an extensible language by means of computational reflection [Mae87]. Thus, its expressiveness and the dependency management mechanism may be adapted according to the application's needs.

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

This paper is organized in three parts. Firstly, we present other work on dependency expression. Secondly, we give an intuitive view of the definition and maintenance of dependencies in FLO. Finally, we describe some examples of use and adaptations of the language for knowledge representation. In particular, we present different meta-links (dependencies between dependencies) and a new operator to design some composition relationships.

2 Motivations

"... no object is an island. All objects stand in relationship to others, on whom they relay for services and control." [BC89] The necessity of dependency definition and management has been known for a long time [Woo75, HK85, EWH85]. Nowadays, different methods of modeling and design [BELR92, NECH92] outline the existence of relationships. Many applications in several domains (graphical interfaces, knowledge representation and acquisition [DT88], hypertext, ...) require the expression of interactions between objects. However, in traditional object-oriented languages there is no way to declare the behavioral dependencies, so the user has to implement them using built-in mechanisms such as inheritance, attributes, method combination, active values, or daemons [SBK86]. Thus, some languages, such as Smalltalk-80, introduce tools for managing inter-object dependencies. Constraint languages also answer to this lack of expressiveness in object-oriented languages. Let us examine these different approaches in the following sections.

Traditional Object-Oriented Solutions. In order to illustrate the traditional implementations of inter-object dependencies, we chose the following example. Let us suppose that a 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?. We want to express a dependency between two instances of these classes, such that all the popped values of the stack instance must be stored within the memory instance associated with the stack . Such a dependency between a stack and a memory is consistent if the memory stores all the popped values of the stack.

A simple implementation consists of creating a subclass of stack to represent stack-with-memory, in adding a slot memory to this class and in specializing the pop method so that its execution implies automatically sending a store message to the memory associated with the stack. Daemons, accessors, references to linked objects, or active values [SBK86] are other similar ways to manage dependency consistency. Such implementations give a poor dependency expressiveness. The dependency semantics is not clearly expressed, and dependencies are hard-wired into object functionalities. Consequently, object structures and functionalities are polluted by non-intrinsic information. Programming and maintaining applications involving inter-object dependencies are then difficult.

The MVC Model. The MVC[2] model [KP88] uses the Smalltalk *dependencies*, which are based upon message propagation between objects: when a model

[2] Model View Controller

changes (changed method), a notification message (update: method) is broadcasted 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. 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 linked objects (by 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. The advantage of this approach is that the model does not explicitly know or refer to its dependents.

The ALV Paradigm. 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 ALV paradigm and FLO. However, as we will show FLO allows one to control any method, whereas RENDEZVOUS limits link definition to instance variable accesses.

Constraint Languages. In constraint-based languages, 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 constraints, modifying the linked variables ([FBB92, MGZ92, San93, Ber93, Kum92]). Constraints are not expressed in terms of object interactions, so some inter-object dependencies are difficult to express as constraints between instance variables. In the proposed example, it is clear that the use of constraints is not natural. The user only wants to express interactions between messages of its objects and not mathematical constraints between values of their instance variables. Moreover, some limitations on types of components are imposed by the constraint solvers. And constraint expressions violate encapsulation. However, constraints are a powerful formalism to manage some particular inter-object dependencies between objects. As the constraints are not at the same level of expression as our links, constraints are complementary to our approach.

Contracts. 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. With a contract, classes of linked objects are structured by and around dependencies. 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]. Our approach differs significantly from that of contracts, because we believe in the equal importance of dependencies and of the objects they relate.

3 Dependencies in FLO

We claim that the existence of inter-object dependencies changes the behaviors of linked objects. Thus, in this section, we describe the FLO language via a simple example, i.e. the nature of our dependencies (named links), the way they are defined, and the process of link maintenance.

Link Declarations. Let us recall the previous example concerning two independent objects: a stack object s1 and a memory object m1. To express the previous dependency as a link instance between s1 and m1, the user defines a *link*, named *memorized-by*. This link is only expressed by referring to stack and memory methods, so that all popped values of the stack are stored in the linked memory, until the memory is full. After that, he/she instantiates this link with s1 and m1 (line 6).

```
1 ( deflink  memorized-by (:stack :memory)
2   ((( pop :stack) → ( store :memory :result))
                    ;; the popped value is stored in the memory
3    (( pop :stack) |  ( not-full? :memory))))
                    ;; stack can pop an element only if memory is not full
4 (define s1 (new stack))       ;; we create a stack and a memory
5 (define m1 (new memory))      ;; and an instance of link between these instances
6 (define s1-⟩m1 (new  memorized-by :stack s1  :memory m1))
```

Definition 1: the link *memorized-by*.

Line 2 of definition 1 shows that, when pop is called for the object denoted by the *:stack* variable, store must be sent to the object designated by the *:memory* variable[3], with the result (denoted by the predefined *:result* variable) of the pop call as argument. 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 3, pop will only be performed on *:stack*, if *:memory* is not full. Thus, the semantics of the *permitted-if* operator (|) is such that the method can be applied only if the expression following such a | operator is true. We call such an expression a *guard*. The link *memorized-by* can be used for linking any instance of stack with any instance of memory. The instances s1 and m1 become dependent when an instance of the *memorized-by* link is created between them (see line 6). 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

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

consistency of this instance of *memorized-by* link, controlling the messages sent to those instances in accordance with the link definitions.

The link definitions are independent even if such links concern the same objects. 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 representation `gr-s1` with appropriate methods. To link these objects `s1` and `gr-s1`, he/she defines a new link, called *graphically-represented-by* (see definition 2), such that `pop` method calls imply the removal of the corresponding graphic value, `push` calls lead to the addition of a new graphic value, and `empty` calls reset the representation. This definition is independent of the *memorized-by* definition. Thus, the stack instance `s1` can be linked at the same time to the memory instance `m1` and to the graphic representation `gr-s1`.

```
( deflink graphically-represented-by (:stack :graphic)
 ((( pop :stack) → ( remove-top :graphic))
  (( push :stack val) → ( add :graphic ( conv val))))
  (( empty :stack) →  ( reset :graphic)))

(define gr-s1 (new single-descriptor)) ;; one graphic object is created
(define s1-graph1 (new graphically-represented-by :stack s1 :graphic gr-s1))
```

Definition 2: the link *graphically-represented-by*.

An Example of Link Definition Inheritance. Another kind of reuse is necessary: the link reuse. Flo offers a special inheritance mechanism to specialize link definitions.

Let us suppose that we define a new link, called *reactive-gr*, subclass of the link *graphically-represented-by*, such that some messages on the representation now imply modifications of the stack object: if we click on the top value of the graphic object, the stack is popped (line 3). To increase the expressiveness, we have introduced the possibility of renaming the inherited link variables within the new link (line 2 of the definition of *reactive-gr*).

Moreover, new participant objects can be added to a link (see the link *complete-gr* line 4 in the definition 3). For example, the previous representation of the stack can be completed with a button *stop-bt*. When this button is selected, no value may now be pushed or popped in the stack, and the representation of the stack may no longer be selected (lines 6, 7, 8).

```
1 ( deflink reactive-gr (:stack :react-grap)
2     :inherit (( graphically-represented-by (:graphic rename-as :react-grap)))
3  ((( click-top :react-grap) → ( pop :stack))))

4 ( deflink complete-gr (:stack :react-grap :stop-bt) :inherit ( reactive-gr)
6  ((( select :stop-bt x)  → (if (eq x t)
                                 ( inhibit :react-grap)
                                 ( reactive :react-grap)))
7   (( push :stack v) | ( if-not-selected :stop-bt))
8   (( pop :stack) | ( if-not-selected :stop-bt))))
```

Definition 3: an example of link inheritance.

4 Links for Knowledge Representation

In this section, we highlight the power of expressiveness of FLO for object-oriented knowledge representation.

4.1 Meta-Knowledge: Links between Links

Links are themselves objects. Thus as simply as other links, we can express links between links, called "Meta-links[4]". Meta-links are not specific to a particular application, and must be considered as powerfull FLO expressive tools. We first present a simple definition of the link *inverse*, and use it to define dependencies between dependencies as defined in Merise [NECH92].

The Inverse Link. If l^{-1} is the inverse of the link l, then x linked to y by l^{-1} implies that y is linked to x by l (see definition 4). In particular, we can specify the reflective aspect of the inverse link: the inverse of the inverse is the inverse (line 4 of the definition 4). Consequently, defining l^{-1} as the inverse of l is sufficient to define l as the inverse of l^{-1} (line 5).

```
1 ( deflink inverse (:masterlink :slavelink)
2     :inherit ( binary-link)
3   ((( create :masterlink obj1 obj2) → ( create :slavelink obj2 obj1))
4    (( create :slavelink obj2 obj1) | ( if-not-exist :slavelink obj2 obj1))))

5 (create  inverse  inverse  inverse)
6 (create  inverse father-of son-of)
  ;; the link inverse between son-of and father-of is automatically created
7 (create  father-of Caesar  Brutus)
  ;; the link son-of between Brutus and Caesar is automatically created
8 (create  son-of Jesus God)
  ;; And the link father-of between God and Jesus is automatically created
```

Definition 4: the link *inverse*.

Other Meta-Links. OMT [BELR92] or Merise [NECH92] methodologies characterize different constraints on relationships. The following definitions present some meta-links as defined in Merise. The main idea of this kind of meta-links is to control the link instance creation. We can notice the use of the *inverse* link for the definition of the links *exclusion* and *simultaneity*, and the use of inheritance to define the *simultaneity* link as a specialization of the *inclusion* link, as shown in the definition 5. The sole difference between the *inclusion* and *simultaneity* links is that the last one is the inverse of itself.

[4] This possibility follows the same philosophy as the Meta-Constraints of [Ber93].

```
( deflink   exclusion  (:masterlink :slavelink)  :inherit ( binary-link)
  ((( create :masterlink obj1 obj2) | ( if-not-exist :slavelink obj1))))

(create  inverse exclusion exclusion)
(create  exclusion father-of son-of)   ;; X can not be both the father-of and the son-of Y

( deflink  inclusion   (:masterlink :slavelink)  :inherit ( binary-link)
  ((( create :masterlink obj1 obj2)→ (if ( if-not-exist :slavelink obj1)
                                         ( ask-for-creation :slavelink obj1)))))

( deflink   simultaneity   (:masterlink :slavelink)   :inherit ( inclusion))
( create  inverse simultaneity simultaneity)
( create  simultaneity father-of husband-of)
```

Definition 5: some Merise links.

Meta-links facilitate the definition of links, outline some properties of links and increase reuse of link definitions. For the sake of understanding, we have just presented binary meta-links, but other links such as the composition of links can be defined.

4.2 Language Extension and Use

In the previous sections, we have presented the standard aspects of FLO. However FLO has been designed to be an extensible language [DBFP95] and we present an additional interesting feature of FLO and its use for composing objects.

A New Operator in FLO for Propagating Messages. As shown in the above examples, the behavior of a linked object is guarded or modified by the links. However, according to some applications, we noticed that linked objects may also acquire new behaviors, which are due only to the link existence. Therefore, the idea is to allow an object to answer some new messages as soon as these messages are defined by links concerning this object. So we propose a new operator, *corresponds* ($\gg$). This operator allows the declaration that a message received by one object of the link has to be re-sent to another object of this link. In this sense - (method1 :object1 arg1) $\gg$ (method2 :object2 (fct arg1)) - means that when a message corresponding to the method1 is not defined, another (or the same) message is sent, using (or not) calling arguments, to another object (or to itself).

The user of the first example wants to ask the stack *s1* which object is its representation. However, it is clear that only s1 is related to a graphic object, therefore this property is not a class property but a property of the s1 instance. Thus, the programmer defines the link *reactive-gr-new* as a specialization of the link *reactive-gr* and he/she indicates the message corresponding[5] to the representation-of call (line 2 of definition 6).

[5] Some FLO primitives allow one to find the same information, but in our sense, in a less natural way.

```
1 ( deflink  reactive-gr-new  (:stack :react-grap)  :inherit-from ( reactive-gr)
2        ((( representation-of :stack) ≫ :react-grap)))
   ;; when linked, the stack knows its representation
3 (define s1 ( new stack))   ;; we create a stack and a graphic object
4 (define gr2 ( new graphical-object))
5 ( representation-of s1) − − − − − >    error no-applicable-method
6  (define s1−⟩gr2 (new  reactive-gr-new  :stack s1 :memory gr2))
   ;; Once an instance of link s1−⟩gr2 is created the stack. s1 knows its representation.
7  ( representation-of s1)    − − − − − >  gr2
```

Definition 6: an example of *corresponding messages*

Composition and Collection. *"Aggregation is ignored by the popular object-oriented programming" [LSR87].* Some research attempts to address this lack of object-oriented programming by introducing specific dependencies between a part and a whole object (*whole/part association* WPA [Civ93], *is-part-of dependency* [WBWW90], *part dependency* [LSR87], *aggregation* [BELR92]). We do not discuss these different approaches in this paper. However, in this section, we highlight the features of the links for composing objects. Through links, the user can specify the semantics of his/her WPA. In particular, the operator ≫ can be used to manage the composition problem between whole-part entities, exposed by Blake:"*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]. If we link a car (the whole) to the coachwork (a part), when we ask the car for its color, this message has to be re-sent to the part which is able to respond to this message. With the operator ≫, *"the whole protocol which a part understands"* must not be re-implemented, the *corresponding messages* are introduced in links. In our example, we also chose to destroy the coachwork if the car is destroyed.

```
( deflink  car-composition  (:car :coachwork :wheels)
   ((( color-of :car)  ≫ ( color :coachwork))
   ( ....)
   (( destroy :car) → ( destroy :wheels) ( destroy:coachwork))))
(define carcmp ( new car-composition :car a-car  :coachwork  a-coachwork
                              :wheels ´(a-wheel wheel a-wheel a-wheel)))
```

Definition 7: an example of composition.

The originality of our approach is that the classes of the part objects need not to be changed when they are involved in a whole-part association. Moreover, as classes are first class objects in FLO, we can also define links on classes. Thus, automatic creation of parts and of links can be declared as links between classes of the whole, of the parts, and of the links.

The concept of group or collection is integrated in the FLO language, by declaring the linked objects as a list. The language offers methods to add (resp.

retract) objects to (resp. from) a collection. In definition 8, a drawing is a collection of shapes. If a drawing is moved all the shapes are moved. A shape can be moved only within the limits of its drawing. To express this dependency, we define the link *together*, by using some built-in primitives of the language.

```
( deflink  together (:master :slaves)
 ((( move :master delta ) → (for-each (x) ( move x delta) :slaves)
  (( move (x  in :slaves) delta)) | ( if-in-limits x delta :master))))

(define adraw (new  drawing))
(define l (new  together :master adraw :slaves '(s1 s2)))
( add l :slaves s3)          ;; add a shape to the drawing adraw.
( retract l :slaves s1)      ;; retract the shape s1 from the drawing adraw.
```

Definition 8: an example of group.

5 Conclusion

Current object-oriented languages are not expressive enough to represent the richness in semantic properties and roles of inter-object dependencies. In the traditional object-oriented languages, the implementation of these dependencies is buried into the object code. To respond to this problem, FLO is an object-oriented language integrating the concept of inter-object dependencies in a declarative way [DBFP95]. The user can define links and instantiate links between objects; the language automatically manages their consistency. Through links, the behavior of linked objects is changed (methods are controlled), and it is possible to associate new behaviors to linked objects. FLO allows one to clearly specify the changed object behaviors. This specification does not occur at the class level as in Smalltalk, nor in the object definition. Links are defined in an independent way from the linked objects, enforcing the principle of encapsulation. As links are expressed in terms of the object interfaces, and as they may be dynamically added or removed without interfering with the object implementation, modularity is strengthened. The code of linked objects is kept pure: no relational information is spread across the classes of the linked objects. The classes define the intrinsic knowledge of the objects and the links the relational knowledge. Moreover, knowledge about dependencies can be expressed in particular links between links. The use of these "meta-links" increases the robustness of the applications.

References

[BC87] Edwin Blake and Steve Cook. On Including Part Hierarchies in Object-Oriented Languages. In *ECOOP'87*, LNCS 276, pages 41–50, 1987.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA'89*, pages 1–6, 1989.

[BELR92] M. Blaha, W. Permerlani F. Eddy, W. Lorensen, and J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall, 1992.

[Ber93] Pierre Berlandier. The use and interpretation of meta level constraints. LNAI 727, pages 271–280, 1993.

[Civ93] F. Civello. Roles for composite objects in object-oriented analysis and design. In *OOPSLA'93*, pages 376–393, 1993.

[DBFP95] S. Ducasse, M. Blay-Fornarino, and A.M. Pinna. A Reflective Model for First Class Dependencies. In *OOPSLA'95*, pages 265–280, 1995.

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

[DT88] R. Dieng and B. Trousse. 3DKAT, a dependency Driven Dynamic Knowledge Acquisition Tool. In *3rd ISKE*, 1988.

[EWH85] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept : An extension to the entity-relationship model. In *Data and Knowledge Engineering*, pages 75–116, 1985.

[FBB92] B. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *ECOOP'92*, LNCS 615, pages 268–286, 1992.

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

[Hil92] Ralph D. Hill. The abstraction-link paradigm: Using contraints to connect user interfaces to applications. In *CHI'92*, pages 335–342, 1992.

[HK85] P. Harmon and D. King. *Expert Systems. Artificial Intelligence in Business.* Judy V. Wilson, re-edited 1985. Re-edited by Wiley Press Book.

[KP88] G.E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, August 1988.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems: a survey. In *AI Magazine*, volume 13, pages 32–44, 1992.

[LSR87] M. E. S. Loomis, A. V. Shah, and J. E. Rumbaugh. An Object Modelling Technique for Conceptual Design. In *ECOOP'87*, LNCS 276, pages 192–202, 1987.

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

[MGZ92] B.A. Myers, D.A. Guise, and B. Vander Zanden. Declarative programming in a prototype-instance system: object-oriented programming without writing methods. In *OOPSLA'92*, pages 185–199, 1992.

[NECH92] D. Nanci, B. Espinasse, B. Cohen, and H. Heckenroth. *Ingenierie des systemes d'information avec Merise.* Sybex, 1992.

[San93] M. Sannella. The skyblue constraint solver. Technical report, Dept of Computer Science and Engineering, University of Washington, 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, 1986.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[Woo75] W.A. Woods. What's in a link: Foundations for semantic networks. In Academic Press, editor, *Representation and Understanding: Studies in Cognitive Science.* 1975.

This article was processed using the LaTeX macro package with LLNCS style