Executable Connectors: Towards Reusable Design Elements*

Appeared in ESEC'97

Stéphane Ducasse and Tamar Richner

Software Composition Group, Institut für Informatik (IAM), Universität Bern
(ducasse,richner)@iam.unibe.ch
http://iamwww.unibe.ch/∼(ducasse,richner)/

Abstract. The decomposition of a software application into components and connectors at the design stage has been promoted as a way to describe and reason about complex software architectures. There is, however, surprisingly little language support for this decomposition at implementation level. Interaction relationships which are identified at design time are lost as they get spread out into the participating entities at implementation. In this paper, we propose first-class connectors in an object-oriented language as a first step towards making software architecture more explicit at implementation level. Our connectors are run-time entities which control the interaction of components and can express a rich repertoire of interaction relationships. We show how connectors can be reused and how they enhance the reuse of components.

1 Introduction

In modeling software architectures Allen and Garlan distinguish between implementation relationships and interaction relationships of software modules or components: "Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system" [AG94]. Allen and Garlan propose a formal model for software design that makes explicit the interaction relationships between components using the abstraction of connector.

Describing software architectures in terms of interaction relationships between components brings us closer to a compositional view, and hence a more flexible or open view of an application [ND95]. First-class connectors allow us to view an application's architecture as a composition of independent components. We gain in flexibility, since each component could engage in a number of different agreements, increasing the reuse potential of individual components. Separating connectors from the components also promotes reuse and refinement of typical interaction relationships. It opens the possibility of the refinement of connectors and the construction of complex connectors out of simpler ones.

* This research is supported by the Swiss National Science Foundation, grant MHV 21-41671.94 (to T.R.) and project grant 2000-46947.96

But whereas implementation relationships use the primitive abstractions of a programming language such as procedure or method call, interaction relationships are rarely captured by programming language constructs. In this sense, traditional object-oriented languages provide little support for explicit representation of software architecture. Class hierarchies are the only design elements visible at the implementation level - but they represent inheritance relationships, and do not reflect an application's architecture. In contrast, interaction relationships, such as coordination and synchronization of a group of objects collaborating to achieve a task, manifest themselves as patterns of message exchanges. Such patterns of communication have a logical and conceptual identity at the design level but this identity is lost when we move from design to implementation as the information about such collaborations is spread out amongst the participating objects. The loss of this information makes the resulting application opaque with respect to its architecture: the design is no longer apparent, making the application difficult to understand, to re-use and to re-engineer.

A first step towards making an application's architecture more explicit at the code level is to enable the localization of information about interactions in an application's code. In this paper we propose one solution: enriching object-oriented languages with an explicit connector construct. As in [AG94], our connectors are first-class objects which represent the interaction relationships between components. Our contribution, however, is to provide connectors at the *implementation level*: our connectors are run-time entities that not only describe, but actually control inter-component communication. Note that we are not proposing a new object-oriented language. Rather, by presenting our model, FLO [1], we show how the traditional object model can be extended to provide explicit connectors between components and show that the reification of such entities promotes reuse of components as well as of typical interactions.

The paper is structured as follows: in section 2 we discuss the problem of language support for explicit connectors. Section 3 presents the basic concepts and notation for representing connectors in FLO. Sections 4 and 5 illustrate our approach with some examples and section 6 discusses implementation issues. Section 7 gives an overview of related work. Finally, section 8 concludes with a discussion - evaluating our contribution and pointing to directions for future work.

2 Language Support for Explicit Connectors

Our work is based on the recognition that relations between components are as important as the components themselves. Providing a construct for explicitly specifying interactions between components addresses the following common software problems:

Inability to localize interaction information: loss of design information. Some of the design of the application is lost during the implementation since we cannot localize information about interactions. This is most evident when we try to re-engineer an application. Program code contains little of the interaction relationships identified at design time, making reverse engineering a much more difficult task.

[1] The FLO model is an extension of the ObjVlisp model [Coi87] and is implemented in a CLOS-like language and in Smalltalk.

Mixing of concerns: impediment to reuse. Logically, components should have an identity independent of the different interactions in which they could engage. When no connector construct is available, component behavior includes the connector behavior, making for less reusable components. Providing a connector entity at implementation level allows abstraction and factorization of all the information about a connection and also allows for the reuse of typical interaction relationships.

To address the first problem, interaction relationships should be represented by an *explicit* construct, as in [AG94]. This is in contrast to approaches of enriching component interfaces with protocols which capture interaction information [YS94], and to the approach of Darwin [MDEK95], where the connection of components is defined as the binding of services and is found *inside* the definition of a composite component. To address the second problem, components and connectors should be independent of each other - more specifically, although connectors must specify the kinds of components which they connect, components should not be aware of the relationships in which they may engage. This is in contrast to the composition-filters approach of [AWB$^+$94], where object interfaces must be modified to allow them to engage in new kinds of interactions, and to the approach of *gluons* [Pin93], where objects must address the mediating gluon in order to collaborate with each other.

Given these basic properties, we first present specific issues which must be addressed in providing language support for such connectors, then summarize the design choices made in our approach.

2.1 Requirements for First-Class Connectors

Connector Specification. Is a connector a user-defined abstraction or is it represented by a fixed abstraction? How can connectors be specified?

Range of abstractions. A connector should be a user-defined abstraction, permitting us to represent a large range of interaction relationships, rather than restricting us to a fixed set of abstractions or mechanisms, as can be found in certain environments (e.g. pipe/filter systems). To allow a larger range of relationships to be described, a connector should be able to connect more than just two participants. A connector specification should define the kinds of participants which may engage in the interaction abstractly, in terms of the interface of those participants.

Specification process. A connector should first be defined *abstractly* then later instantiated with the actual participants. We should be able to define a connector *incrementally* - that is, a new connector could be defined from existing ones by modifying or combining connector definitions. Finally, we would like to be able to define *generic* connectors, to allow for the reuse of typical interaction relationships.

Connector Lifetime. Is a connector a dynamic entity which can be created and destroyed, or is it active throughout the application or throughout the lifetime of its participants? How is a connection activated and terminated?

Lifetime. Since it is natural that interactions between entities are formed and dissolved dynamically, a connector should be a dynamic entity which can be created to manage an interaction and destroyed when that interaction is no longer required.

Activation and Termination. A connector should be activated and terminated in such a way that the participants themselves need not be aware of the connection.

Connector behavior. What kinds of relationships can be expressed with a connector? What expressive power do connectors require to represent these relationships?

Kinds of Relationships. Connectors should be used to specify all the information relating to the interaction of components, including data conversion [AWB+94], interface adaptation [YS94], synchronization and coordination [FA93] and other patterns of collaboration and cooperation.
Expressive power required. In representing the communication behavior in relationships such as those given above, we depend on the basic communication paradigm of the language in which the connectors are implemented. As a general rule, however, we consider it important for expressive power that connectors not only be able to coordinate communication as do *synchronizers* [FA93], but also to enforce state changes in the participants.

Formal Properties. Can compatibility of a component with a connector be checked? Can we give certain guarantees on the behavior of a composition?

Ideally we would like to have a formal notion of compatibility and substitutability so that we know exactly what kinds of components can participate in a connection, and which kinds of components can replace each other as participants. Furthermore, we would like to be able to prove certain connector properties (e.g. deadlock freedom) [AG94].

2.2 Connectors in FLO

FLO is an extension of the object-oriented model with first-class connectors. Connectors in FLO are user-defined: they are first defined abstractly, then instantiated. They can be specified incrementally and, in some cases, as generic constructs. They are dynamically created and destroyed, with components remaining unaware of the connectors in which they participate. Connectors allow for the specification of a large range of relationships – they observe and control the communication between participants and can also enforce state changes in the participants. FLO is based on a *sequential* object-oriented model: we do not yet seek, therefore, to represent synchronization and coordination mechanisms which require concurrency for their expression. Our approach provides a *descriptive* and *executable* notation for connectors, in contrast to Allen and Garlan [AG94], who present a notation for connectors which has descriptive and analytical properties. FLO's connectors can provide a basic test at connector instantiation to check if the participants indeed provide the required interface, but our approach does not, at present, formalize the notion of compatibility. In the next section we present the basic concepts of FLO's connectors in greater detail.

3 FLO's Basic Concepts and Notation

3.1 Component

In the object-oriented programming context, we consider that a component is an object or a grouping of objects. A grouping of objects can be realized using inheritance, aggregation or connectors, the only restriction being that a component should provide as

an interface a set of method selectors and signatures[2]. A component's interface is then basically a set of signatures as in CORBA IDL [OMG95] and does not provide a formal description of the relationships between its different methods as do the augmented interfaces proposed by [YS94].

3.2 Connector

Connector Specification. A connector in FLO is a special object that connects components, called its *participants*. A component can participate in more than one connector. A connector is specified by a connector template, which describes all the information representing the connection between the participants by specifying how message exchanges influence the behavior of the participants. We call this specification the *dynamic behavior* of the connector. A component can participate in a connection as long as it provides an interface compatible to a *role* required by the connector. Roles are specified by variable names in the connector template declaration, but they are implicitly defined in the dynamic behavior of the connector; a role is the set of method selectors on a participant which will be intercepted or invoked by the connector, so it is a subset of a component's interface. This will be clarified further in the discussion of the dynamic behavior of a connector.

```
(defConnector  connectorAB  (:roleA :roleB)   ; a list of role names
   :inherit ((...))             ; a list of ancestors
   :var                          ; some connector variables
   :behavior                    ; interaction rules of connector
)
```

Abstraction, incremental definition, and generic connectors. Connectors are first specified abstractly by defining a connector template, then instantiated by specifying the actual participants. This abstraction of the behavior of a connector is useful for incrementally defining new connectors. A new connector template can be defined from existing connector templates by adding new interaction rules or by combining connector definitions (see example in section 5.2).

Moreover, FLO provides *generic* connectors in certain cases. A generic connector template specifies a connection schema which can then be instantiated to generate a new connector template by specifying the abstract interfaces with the effective component interfaces (see example in section 4.2).

Connector Lifetime. A connector, like other objects, has a lifetime - it is created, during its lifetime it controls the communication between entities and it can be destroyed. Different kinds of actions can be associated to these three distinct phases of the connector's lifetime:

> **creation.** Some actions are mandatory at connection creation time. For example, a component could be initialized before being connected with others components. A connector can also refuse to create the connection, depending on specified criteria - that is, the connector would be destroyed without moving on to the next phase.

[2] FLO is implemented on untyped languages such as CLOS, so a signature is only a list of symbols representing formal arguments of a method.

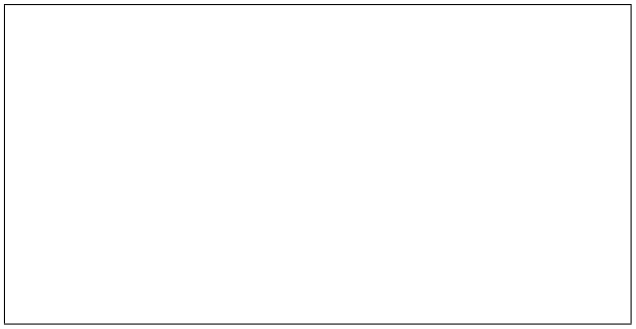

Fig. 1. Connectors and components.

active connection. During the connection, a connector intercepts the method invocations of the participants and ensures that the specification of the connection is met. For example, a connector can forbid messages according to certain conditions, or send new messages to other participants [AWB$^+$94]. This behavior is specified in a connector template definition by interaction rules (see dynamic behavior below).
destruction. Some actions can be specified when a connector is removed. A component can be informed, for example, of the closing of the connection.

Connector Behavior.

Dynamic behavior. The behavior of a connector is defined by means of a set of *interaction rules* which specify how the messages received by participant objects should be controlled: is the message allowed? Under which conditions? Does the reception of a message imply the sending of other messages? We call this the *context* of a message reception.

The rules for specifying the dynamic behavior possess the following simplified syntax, their semantics differing only according to the rule operator.

Rule	`::=`	Filter Operator Context
Filter	`::=`	Selector Rolename List-Of-Calling-Args
Context	`::=`	Message$^+$
Message	`::=`	Selector Rolename Args
Operator	`::=`	`implies` | `permitted-if` | `corresponds`

The *filter* of a rule specifies which messages (method selector with calling arguments) should be intercepted for which kinds of participants, given by *rolename*. The *operator* defines the semantics of the rule and gives meaning to the *context* of the rule. Three operators: `implies`, `permitted-if` and `corresponds` are predefined[3] in FLO. Finally, the *context* of the rule specifies the execution of messages: a list of

[3] However, FLO has an open implementation language [Duc97b] and its meta-object protocol allows for the definition of new operators.

method invocations on participants. Note here that in order for a component to participate in a connector it must provide in its interface all the selectors which are associated to its rolename in the interaction rules.
These rules allow us to specify three different kinds of semantics:

Propagation is specified using the `implies` operator. After the reception and the execution of a message, some other messages, named *compensating messages* are sent to the sender object or to other participants.
Inhibition is specified using the `permitted-if` operator. The received message is only executed if a condition, named a *guard*, is satisfied[4]. The context part is then a boolean expression giving the condition for which the execution is permitted.
Delegation is specified by the `corresponds` operator. Instead of the received message being executed a new message is sent to some of the participants.

Here we see that, in contrast to event-based connectors like Mediators [SN92], FLO's connectors can *forbid* method execution, and that, in contrast to Synchronizers [FA93], FLO's connectors can *invoke* methods on controlled components.

Special operations. The fact that a connector is an object means that its behavior is represented by means of methods, some of which can be specialized to adapt the connector behavior. Because of space limitations, we do not present here special methods linked to the creation phase of a connector and to the dynamic aspect of managing groups of participants.

As FLO has an open implementation [KdRB91], different methods defining a *meta object protocol* allow the complete connector behavior to be adapted. However, as these aspects concern the modification of the language itself and should not concern an application developer, they are out of the scope of this paper (see [Duc97b] for more information).

In summary, a connector is a run-time entity in which information about interaction (data and behavior) between components is stored. Data represents the state of the interaction, e.g. which component has received or sent a message. Connector behavior represents behavior specific to the interaction of the participants. A connector is then an appropriate place to specify all the information which is particular to an interaction: in addition to specifying constraints on message exchanges when components collaborate to achieve a task, connectors are also the right place to define conversion of data and to adapt or enrich component interfaces (see example of section 5).

Having summarized the main properties of FLO's connectors, we now present some examples. These allow us to demonstrate how connectors are defined, to illustrate their properties and to show how connectors can be reused.

4 Ensuring Exclusion

We first show how a connector template is defined and discuss connector properties, then show how FLO provides generic connectors.

[4] In case the condition is false, a default value is sent to notify the caller.

4.1 Connector Definition

We show how a template can be defined for connectors which enforce exclusion between a set of components (in a sequential setting) – that is, which allows only one component to be active at any time.

Let us suppose that we have simple components – buttons – each of which provides `select` and `deselect` methods. Moreover, following the same constraint given in [Frø94], these buttons are constructed so that each button cannot receive the same message twice. We make this assumption here only because it allows us to provide a shorter code example.

A reactive solution. To specify an exclusion situation among components, a connector is defined which ensures that as soon as a new component is selected the previous one is deselected. Another schema is presented in [Frø94], where a component can be selected only if no other component is already selected[5]. This connector requires only one kind of participant, so the list of roles is given as a set of identical role names.

In the following examples, FLO's global constructs are represented using bold font to distinguish them from other expressions that are specific to the defined connector. The keyword **connector** represents the connector itself - that is, the instantiation of the connector template.

```
1 (defConnector  reactive-exclusion  (set-of :buttons)
2  :var ((active? :initform #f  :accessor active?)   ; two connector
3        (last-button-selected  :accessor selected))   ; variables
4  :behavior
5   (((deselect  :buttons-receiver) implies (set! active? connec-
tor #f))
6     ; a button is deselected, connector state is adapted
7    ((select  :buttons-receiver) implies-before
8       (when (active? connector) (deselect (selected connector))))
9     ; before a button is selected, deselect the previous one selected
10   ((select  :buttons-receiver) implies
11      (set! active? connector #t)
12      (set! selected connector  :buttons-receiver))))
13    ; a button is selected, connector state is adapted
14 ; end of behavior definition
15 (defmethod action-before-effective ((connector  reactive-exclusion)
                                      args)
16   (for-each deselect (give-participant connector :buttons)))
17     ; before creating the connector, all the buttons are deselected
```

This template definition is an abstract definition of an exclusion schema between buttons or any other components that provide the same interface. Once defined this connector template can be instantiated on different sets of buttons as shown below (lines `17` and `18`), resulting in two independent connectors. The interface required for par-

[5] The presented solution sends message to the other participant object, hence the reactive adjective.

ticipants of this connector is that each participant provide the methods `select` and `deselect`.

```
17  (define re1 (make  reactive-exclusion :buttons (list b1 b2 b3)))
18  (define re2 (make  reactive-exclusion :buttons (list b4 b5 b6)))
```

Properties. The *reactive-exclusion* connector template groups all the information related to the connection.

- Line `1` `:buttons` is a variable representing the role of the component. The **set-of** keyword specifies that several components can take this role in the connector and defines a role-group.
- Lines `2` and `3` define two variables: `active?` that indicates if one button of the group is selected and `last-button-selected` that represents the last selected button. These variables possess initial values and accessors. As in a class/instances model, each instantiated connector possesses its own set of variable values.
- Lines `5` to `13` specify how messages should be controlled to ensure that only one button is selected. When a new button is selected, the currently selected button, if there is one, will be deselected[6].
- Lines `14` and `15` specify the actions that should be performed when the connector is created. Here all buttons should be deselected. The method `action-before-effective` invokes the `deselect` method on all the participants associated with the rolename `:buttons` using the `give-participant` method defined for all connectors. However, a connector can also check if the participants are in the right state to be connected and the creation of a connector can be refused according to certain criteria.

The fact that connectors are entities separate from components provides important benefits - connector information is no longer spread across component code. In this example, for instance, buttons do not need to keep track of the last selected button in their own `select` and `deselect` methods, as would be the case with traditional object-oriented languages. This increases the reusability of components, making their code clearer and simpler. A connector can thus be understood at code level as a logical unit and carries an important part of the design decisions down to the implementation level.

4.2 Generic Connectors

FLO allows for the definition of *generic* connector templates, currently with some restrictions. A generic connector template defines a general connection schema that can be instantiated to create a specific connector template for a particular context. Generic connector templates are defined in terms of generic interfaces that are then *instantiated* with real component interfaces. For example, a generic connector template is defined as shown in the code below (lines `1` to `12`) and then instantiated in case of buttons

[6] The `implies-before` allows for the definition of the action performed before the execution of the controlled method, here the deselection of the last selected component. Using it ensures real exclusion.

(lines 13 to 15). The resulting connector template is identical to the *reactive-exclusion* connector template in section 4.1.

```
1 (defGenericConnector  generic-reac-excl  (set-of :components)
2  :var ((active? :initform #f  :accessor active?)
3        (anchor  :accessor anchor))
4  :behavior
5  (((action1  :components-receiver) implies (set! active? connector #f))
6   ((action2  :components-receiver) implies-before
7        (when (active? connector) (action2 (anchor connector))))
8   ((action2   :components-receiver) implies
9        (set! active? connector #t)
10       (set! anchor connector  :components-receiver))))
11 (defmethod action-before-effective ((connector  generic-reac-excl) args)
12      (for-each action2 (give connector :components)))
```

```
13 (defConnector  reactive-exclusion-between-buttons14
14   :is generic-reac-excl :with-participants component = buttons
15   :with-methods action1 = select, action2 = deselect)
```

```
16  (make  reactive-exclusion-between-buttons :buttons (list b1 b2 b3))
```

A generic connector template describes a general pattern of communication between components in terms of abstract interfaces. The use of generic connectors allows for the reuse of complex connector templates that need only be defined once. The definition of generic connectors in FLO is at present limited to specific cases - to those cases when the context of a rule is independent of the filter. That is, the context of an interaction rule should not use the arguments of the controlled message. Future improvements of the FLO implementation will take into account these problems and propose a way to manipulate calling arguments.

5 A kind of Client-Server Connector

In this example we show a connector in which participants play different roles, and demonstrate how a new connector can be defined from existing connectors.
For the sake of exposition, our example is a rather simplified schema of a client-server relationship between tools. It allows us, however, to illustrate that connectors in FLO can forbid message execution according to specified criteria, in contrast to *mediators* [SN92], or to implementations of the MVC model [KP88] or the Observer design pattern [GHJV94], which only propagate messages.

Suppose that we have a simple calculator component that generates new data when the method `computes-new-value` is invoked, and we would like to display the calculated data on a graph displayer that displays a limited number of values on x-y axes, and has a method for displaying a value (`add-new-value`) and one for removing a value by clicking on the display (`remove-one-value`). We want to express that each value computed by the calculator should be displayed by the graph displayer - this means in particular that if the graph displayer is full (method `free-variables?`)

a new value should not be computed. An additional constraint is that the format of the calculator result values is not compatible with the format required by the displayer, so the values must be converted.

5.1 Connector between Calculator and Graph Displayer.

The following connector definition ensures that all the constraints mentioned above are satisfied:

```
1 (defConnector  calculator-displayer  (:calculator :displayer)
2                      ;two distinct participants
3  :behavior
4  (((compute-new-value  :calculator val) implies
5       (add-new-value :displayer (convert connector val result)))
6      ; when a new value is computed, the displayer is aware
7      ; the value is converted to be understood by the displayer
8   ((compute-new-value  :calculator val) permitted-if
9       (free-variables? :displayer))))
10 ; computing a new value is only possible if the displayer can display it
11 ; end of behavior definition
12  (defmethod convert ((connector  calculator-displayer) v1 v2)
13      (list (from-float-to-pixels v1) (from-float-to-pixels v2))
14       ; a conversion  from two floats to a list of two pixels
```

The keyword **result** at line 5 represents the value returned by the controlled method, here the method `compute-new-value`. The connector template above can be instantiated as follows:

```
17 (define c (make calculator))
18 (define d (make displayer  :x 120  :y 200))
19 (define calc-disp (make  calculator-displayer
20                          :calculator c :displayer d :x 50 :y 70))
```

This example illustrates that, in contrast to the MVC model [KP88] or the Observer design pattern [GHJV94], which are design descriptions, a connector actually *controls* the message exchange between participants, ensuring the integrity of the design. Here, the calculator cannot produce a new value if the displayer is not in the appropriate state.

Distinct roles. In this example, the connector requires participants that play different roles. Here two roles are defined – a calculator which provides a method called `compute-new-value` and a displayer which provides the methods `add-new-value`, `free-variables?`, `specify-axes` and `clear-variables`.

A place to define connector behavior. A connector allows for the definition of connection information, one aspect of which is data conversion. Line 5 specifies that the result of the call to the calculator should be converted before sending the `add-new-value` method to the displayer. Whereas with traditional object-oriented languages this conversion would be included in the interface of the component, with a new conversion method being added for each new interaction, here the conversion is a method, called

`convert`, local to the connector itself (lines `12` and `13`). The components themselves need not provide such a conversion method - since a component should not be required to anticipate its possible connection with other incompatible components.

5.2 Reusing and Composing Connectors

We now show how connector templates can be composed to create new templates. Suppose that we want to visualize with a new grapher, called *history*, all the values calculated by the calculator. In addition we would like that when a user selects a vertex in the first graph displayer the corresponding value is highlighted in the history graph. We assume that the history grapher provides a way to change the color of a value (methods `find-values` and `change-color`).

There are different possibilities to implement this new connection. We can define a separate connector template and instantiate from it a new connector which links the history grapher to the graph displayer and to the calculator, retaining the existing connector which links the calculator to the graph displayer. Or we can define a connector template for a connector which links all three components to each other (see figure 2), thus no longer requiring the existing connector. We now present these two solutions.

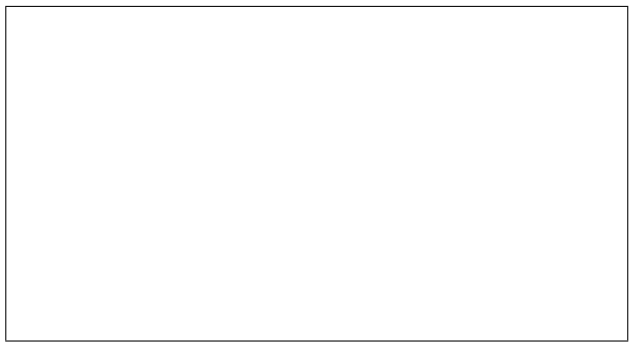

Fig. 2. Two possibilities for defining a new connector: (A) Addition of a separate connector and (B) Derivation of a global connector from the previous definition.

Adding a separate connector. We define a separate connector template called *CDH* as shown below (lines `1` to `11`) and instantiate it (line `13`) on the previous components (i.e. the graph displayer and the calculator) plus the new one (i.e. the history graph). For the sake of simplicity, we assume now that the returned values of the calculator are compatible with the history graph. There are now two independent connectors: *cdh*, instantiated from template *CDH* (line `13`), and *calc-disp*, instantiated from template *calculator-displayer*. It is possible to remove one without being concerned about the other one, and the variable names (`:calc`, `:displ` and `:history`) are also independent. The displayer should provide two new methods: `select` and `deselect`.

```
1 (defConnector  CDH  (:calc :displ :history)
2  :behavior
3  (((compute-new-value  :calc val)
4    implies (add-point :history val result)))
5    ; when a new value is computed, the history graph adds it
6   ((select  :displ point)
7    implies (change-color  :history (find-values  :history point))
8       ; when a point is selected in the displayer,
9       ; the corresponding values in the history are highlighted
10  ((deselect  :displ point)
11   implies (change-color  :history (find-values  :history point)))))
```

```
12 (define h (make history))
13 (define cdh (make CDH :displ d :calc c :history h))
```

Connector Inheritance. Instead of having two separate connectors, it is possible to define by inheritance one connector template which manages the connections described by *CDH* and *calculator-displayer* (figure B in figure 2). Here we can enrich the *calculator-displayer* connector template with new interaction rules as shown in lines 1 to 11 or use multiple inheritance between connector templates *CDH* and *calculator-displayer* (lines 13 to 16).

```
1 (defConnector  Global-CDH  (:calc :displ :history)
2   :inherit-from (( calculator-displayer
3                    (rename :calculator as :calc :display as displ)))
3   :behavior
4     (((compute-new-value  :calc val)
5       implies (add-point :history val result)))
6       ; when a new value is computed, the history graph displays it
7      ((select  :displ point)
8       implies (change-color  :history (find-values  :history point))
9       ; when a point is selected in the displayer,
10      ; the corresponding values in the history are highlight
11     ((deselect  :displ point)
12      implies (change-color  :history (find-values  :history point)))))
```

```
13 (defConnector  Global-CDH  (:calc :displ :history)
14   :inherit-from (( calculator-displayer
15                    (rename :calculator as :calc :display as displ))
16                  ( CDH)))
```

The last template definition is equivalent to the previous one. This last example shows that the inheritance mechanism provides a way to rename the variable representing the role of the participants. It also shows that when two connector templates are composed to give a new connector template through multiple inheritance, the roles of the participants are composed to give a new role.

6 Implementation Issues

The FLO model is based on full message passing control as offered by reflexivity and the open aspect of languages like CLOS [KdRB91,DBFP95] or Smalltalk [GR89]. Few object-oriented languages offer message passing control. Preprocessing the code, as used in Synchronizer [FA93] or OpenC++ [Chi95], can be a solution to introduce such control of message passing. Another approach is to use an implicit invocation mechanism that can be easily added to any language [NGGS93]. However, this solution does not allow full message passing control because implicit invocation cannot support inhibition or redirection of messages.

The connectors described in this paper are fully implemented in CLOS, and a new version is under development in Smalltalk at the University of Berne (see http://iamwww.unibe.ch/∼ducasse/). Moreover, we are now evaluating the introduction of reflexive facilities into Java [Gol97] to support connectors.

7 Related Work

Allen and Garlan propose a formal approach to connectors which is independent of object-oriented languages, their main motivation being to formalize software architectures in general [AG94]. Their formalism allows one to reason about the compatibility of a component's interface, or *port*, with a connector's *role*, and to prove properties such as deadlock-freedom of a connector. Whereas component ports in [AG94] are processes, our ports are just sets of methods. Our connectors have a dynamic behavior part, specified by interaction rules, which corresponds to Allen and Garlan's connector *glue*, but the *role* part of our connectors is implicitly defined in a connector specification as explained in section 3.2. Furthermore, though FLO's connectors can provide a basic test at connector instantiation to check if the participants provide the required interface, our approach does not, at present, formalize the notion of compatibility.

In contrast to Allen and Garlan's formal approach, our contribution is to provide an *executable* notation for connectors: connectors which not only describe component relationships, but also enforce them. *Contracts* [HHG90] are design formalisms used to express cooperation between objects, but they describe rather than enforce the constraints on the message exchange between participants. Also, many design patterns [GHJV94] can be expressed using connectors - in this way retaining and enforcing design decisions at implementation level.

The problem of providing a language construct to express and also control interaction relationships has been approached from a variety of angles. Pintado [Pin93] proposes *gluons* to mediate object collaborations. His approach emphasizes collaborations between objects as client-server protocols, and does not allow for the specification of more general patterns of collaboration, in particular for ones where no server is required, such as the example in section 4. Similarly, components in Darwin [MDEK95] interact through services required and services provided, so that the unit of connection is basically a binding of services of two components. In the Composition-Filters approach of SINA [AWB$^+$94], *Abstract Communication Types* are proposed for enforcing invariant behavior among objects. But, object interfaces must be modified before an object can engage in a new kind of interaction – an impediment to the reuse of components in new contexts.

Yellin and Strom [YS94] represent interaction relationships using protocols specified in the object's interface. When two components are functionally compatible, but their interface protocols are not compatible *adaptors* are used to translate the interfaces. Adaptors are connector-like constructs but in the context of augmented interfaces they require a limited expressive power and can represent only two-party relationships.

Sullivan and Notkin [SN92] separate connectors from the components at the implementation level by providing *mediators*, proposed to facilitate tool integration. Our connectors are close to mediators in the sense that they are based on an implicit message passing mechanism which allows components to remain truly independent [SG96]. In contrast to mediators, however, our connectors are explicit programmable entities which not only relay messages, but can forbid message delivery, redirect messages or send compensating messages to the participants.

Frølund and Agha [FA93] propose *synchronizers* for multi-object coordination in a concurrent language. Synchronizers are similar to our connectors but there is a main difference between the two approaches. A synchronizer only updates its own state on receiving a message from a participant, but it cannot itself send messages to its participants and alter their state. A synchronizer only coordinates communication. Our connector is, in this sense, more active - it can enforce state changes in the participants by sending messages to components.

8 Conclusions

We have proposed first-class connectors in a sequential object-oriented language as dynamic user-defined abstractions which coordinate and control component communication. We consider FLO's connectors a powerful construct for expressing interaction between components and for structuring software in a way which gives equal importance to relationships and to the components they relate. Our contribution is to provide a descriptive and executable notation for connectors and thus enable the localization of information about interaction of components at the level of implementation.

We have also shown how connectors themselves can be reused: through inheritance or through generic connectors which describe schematic interaction relationships, and have argued that the presence of connectors promotes the reuse of components in new contexts.

Executable connectors are a first step towards the goal of making software architecture more explicit at implementation. Connectors enrich the design vocabulary and carry some design information to the application code. They do not, on their own, lead to better designs, but they are an aid in the codification and enforcement of design idioms (e.g. design patterns [Duc97a]). We further plan to investigate the use of connectors in enforcing more general design constraints, as in architectural styles [SG96]. We see this use of connectors as a contribution to the goal of composing applications from existing software artefacts.

We plan to pursue our work on connectors in three main directions:

Genericity. In particular, improving generic connectors by developing constructs to manipulate calling arguments and by introducing parameters for connectors which would allow generic connectors to be instantiated with code fragments to tailor a generic interaction schema to a particular context.

Concurrency. We are currently extending our approach to concurrent objectoriented languages with active objects. A connector is then an entity that synchronizes concurrent objects and controls their communication. This leads us to introduce new kinds of operators to handle different synchronization schemes.

Formal approach. The current approach lacks a formalization of componentconnector compatibility and of component substitutability. To formalize these notions we may need to revise our notion of component interface and connector role, perhaps in the line of augmented interfaces proposed in [YS94].

9 Acknowledgments

We are grateful to Oscar Nierstrasz for his encouragement and advice, and for many helpful comments on the manuscript. Our thanks also to Serge Demeyer, Theo Dirk Meijler, Markus Lumpe and the anonymous referees for their comments on an earlier draft.

References

[AG94] R. Allen and D. Garlan. Formal connectors. Cmu-cs-94-115, Carnegie Mellon University, Mar. 1994.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Object-Based Distributed Programming (ECOOP'93 workshop)*, LNCS 791, pp. 152–184, 1994.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, pp. 285–299, 1995.

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

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

[Duc97a] *Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation*, 1997. Language Support for Design Patterns and Frameworks ECOOP'97 Int. Workshop.

[Duc97b] S. Ducasse. *Intégration réflexive de dépendances dans un modèle à classes*. PhD thesis, Université de Nice-Sophia Antipolis, 1997.

[FA93] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *ECOOP'93*, LNCS 707, pp. 346–360, 1993.

[Frø94] S. Frølund. *Constraint-Based Synchronization of Distributed Activities*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gol97] M. Golm. Design and implementation of a meta architecture for java. Master's thesis, IMMD at F.A. University, Erlangen-Nuernberg, 1997.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

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

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

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

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. ESEC'95, LNCS 989*, pp. 137–153, 1995.

[ND95] O. Nierstrasz and L. Dami. Component-oriented software technology. In *Object-Oriented Software Composition*, pp. 3–28. Prentice Hall, 1995.

[NGGS93] D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Proc. ISOTAS'93, LNCS 742*, pp. 487–510, 1993.

[OMG95] OMG. *The common object request broker: architecture and specification*, 1995. Revision 2.0.

[Pin93] X. Pintado. Gluons: a support for software component cooperation. In *Proc. ISOTAS'93*, LNCS 742, pp. 43–60, 1993.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

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

[YS94] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proc. of OOPSLA'94*, pp. 176–190, 1994.