Analyzing and Comparing Architectural Styles

Nicole Lévy, Francisca Losavio

Analyzing and Comparing Architectural Styles

N. Levy
PRISM
Universite de Versailles
10, 12 Av. de l'Europe, 78140 Velizy, France
Nicole.Levy@prism.uvsq.fr

F. Losavio
Centro ISYS
Universidad Central de Venezuela
Ap.47567 Los Chaguaramos
1041-A Caracas, Venezuela
flosavio@isys.ciens.ucv.ve

Abstract

In the existing catalogues of either design patterns or architectural styles, numerous are very analogous. They show little differences because they have been developed and used by different people and for different applications. Therefore it is really very difficult, for practical use, to select the right pattern or style for a specific design problem. In general, the criteria given for selection are based on examples or case studies. The formalization of these patterns provides a criteria of comparison.

In this paper, a context for style formalization that takes into account characteristic properties, is described first. Two styles, Mediator and Broker, are formalized. Then the relation of specialization is defined and applied to show that Broker is a particular case of Mediator. As an example, taking advantage of the distribution characterized by Broker, it is shown that in the PAC (Presentation-Abstraction-Control) architectural pattern, the usage of Broker as the control of the application, instead of Mediator, allows to define a distributed architecture for interactive applications, a distributed PAC style.

1. Introduction

Design patterns [GHJV95] represent frequently used ways to combine classes or associate objects to achieve a certain purpose. An architectural style [SG96] or a framework provides a skeleton of an overall system architecture [BCK98]. They are based on design patterns.

In the existing catalogues of either design patterns or architectural styles, numerous are very analogous. They have differences essentially because they have been developed and used by different people and applications. Therefore, it is really very difficult to choose a pattern. In general, the criteria given for selection are based on examples or case studies. The existing classifications [KG98, CM98] are based on behavioral characteristics expressed pragmatically.

Formalizing the characteristics of the patterns provides the user with more precise criteria of comparison. A relation of specialization can be defined to precise when a pattern can be replaced by another one. To do so, the latter must satisfy the characteristics of the former. We will first describe a context for style formalization using the LOTOS [BB87] specification language. This formalization has to be considered as complementary to the existing definitions. Two styles will be formalized: Mediator and Broker.

Then we will define the relation of specialization. We will use it to show that the Broker, in its indirect communication variant [BMR$^+$96], is a particular case of the Mediator. So, the usage of Broker instead of Mediator could take advantage from the distribution characterized by Broker. As an example, in the PAC (Presentation-Abstraction-Control) [BC91] architectural style, the usage of Broker playing the role of the control component of the application instead of Mediator, allows to define a distributed architecture for loosely coupled interactive applications, i.e. a distributed PAC style. In this style, Broker is responsible of communicating the clients displaying the graphical user interface (GUI), with the persistent data of the application on the server. An abstract of this paper has been published in [LLM98] without the PAC example.

Besides the Introduction and the Conclusion, this paper is structured into two main sections: Section 2 describes a technique for formalizing architectural styles using the LOTOS specification language. A brief introduction to the LOTOS specification language used in this paper is given. This formalization is an enrichment to the usual patterns description. As examples, both Mediator and Broker styles are formalized. Section 3 defines a specialization relation for comparing styles. Mediator and Broker are compared according to this relation yielding to the fact that Broker is a specialized Mediator. The Broker is then used instead of Mediator in the PAC style.

2. Formalization of architectural styles

2.1. How to formalize

As shown in [RMD90, HLLM97, Mik98], it is possible to formalize some aspects of design patterns. These aspects are specified here using the formal description language LOTOS [BB87], thus establishing a formal semantics of the communication between the components constituting a design pattern. This formalization is to be considered as an *enrichment* of the pattern description, because there are different aspects of design patterns than those concerned with communication.

Besides providing a formal semantics, the use of LOTOS has the advantage that existing tools, such as CADP (Caesar/Aldebaran Distribution Package) [FGM+92], can be employed to analyze and animate instances of patterns. In particular, several relations such as the observational equivalence [Mil86] that we will use in our specialization relation, can be automatically checked. Furthermore, LOTOS is an ISO standard so a widespread familiarity can be assumed among scientific community.

The basic ideas underlying our formalization are:

- Objects, which exhibit behavior, are modeled as LOTOS processes.
- Messages between objects are expressed by LOTOS communication patterns.

Each formal software architecture description consists of three parts:

- an Intuitive Description
- the Components Characteristics: requirements on the processes specifying the objects contained in an instance of the design pattern,
- A Communication Pattern: a LOTOS communication pattern defining its top-level behavior
- An Invariant: constraints, which provide sufficient conditions for a design description to be an instance of the design pattern. These conditions can be checked mechanically.

Formalizing the communication aspects of design patterns has revealed a strong relation of design patterns to the concept of an architectural style as it is used in software architecture. Architectural styles can be formalized in much the same way as communication aspects of design patterns [HL97].

2.2. Introduction to LOTOS

LOTOS [BB87, LOT87] is a formal specification language developed to specify open distributed systems. A LOTOS specification describes the global behavior of interacting processes. A process can be parameterized by algebraic abstract data types, and it can exchange typed values with other processes and call functions to transform data. Communication between processes in LOTOS is synchronous, i.e., two processes must participate in a common action at the same time. *Gates* are used to synchronize processes and to exchange data. Each process definition has following the syntactic form:

```
process process_name[GateList](params): func:=
    behavior
        behav_expr
    where
        local_def_list
endproc
```

where `GateList` is the list of its communicating gates, `params` is its typed parameters and `func` indicates whereas the process may terminate (`func`=`exit`) or not (`func`=`noexit`). The behavior expression `behav_expr` describes the sequences of observable actions that may occur at the gates of the process. Process definitions included in `local_def_list` may include instantiations of processes.

The behavior expression `behav_expr` can include several operators. The choice operator `[]` is used when alternative behaviors are allowed. The expression `P1 [] P2` expresses that exactly one of the two processes `P1` or `P2` will be executed, depending on a choice of the environment.

The behavior expression `P1 ||| P2` (interleaving) expresses that the two processes `P1` and `P2` behave independently and in parallel.

The behavior expression `P1[g] |[g]| P2[g]` (parallel composition) expresses that the two processes `P1` and `P2` must synchronize on the gate `g`. During the synchronization, they may exchange data. To synchronize, two processes must contain an action on the same gate `g`. To exchange data, one of them must contain an action `g ? v: t` which reads a value `v` of type `t` on gate `g`. The other process must contain an action `g ! exp` that writes a value `exp` of type `t` on the gate `g`. It is also possible to read or write more than one value in the same action.

Behaviors may be made conditional by using the guard operator `[`*pred*`] ->`*beh* . The behavior expression *beh* will take place only if the predicate *pred* is satisfied.

In LOTOS, data are described using algebraic abstract data types with conditional equations and an initial semantics. Abstract data types are used for describing process parameters and values exchanged by the processes.

Note that an asymmetric communication in the object-oriented world usually corresponds to a symmetric communication in LOTOS. If object *A* sends a message to object *B*, then *A* must have a reference to *B*, but not vice versa. In the LOTOS process modeling this communication, the processes `A` and `B` corresponding to the two objects must contain a common gate onto which `A` writes the service request, which is read by `B`. The result of the service is then sent back to `A` from `B`.

2.3. Design Patterns in LOTOS

A valid design description expressed in LOTOS must be a valid LOTOS expression, regardless of the pattern it is an instance of. Each design description consists of two parts. The *behavior* part describes the overall behavior of the design, i.e., the interaction of its parts. The *local definitions* part contains the definition of the processes involved in the behavior part and the necessary definitions of abstract data types. The syntactic structure of a design description is

```
behavior
      behav_expr
where
      local_def_list
```

LOTOS *patterns* are obtained from LOTOS specifications by abstraction, i.e. by replacing concrete LOTOS expressions by meta-variables. Both parts of a design description, i.e., *behav_expr* as well as *local_def_list*, can be subject to abstraction. In the following, concrete LOTOS expressions are set in `teletype`, and meta-variables are set in *italics teletype*.

3. Formalization of Mediator style

3.1. Intuitive Description

The intent of the Mediator design pattern is [GHJV95] to:

> define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Collective behavior of a group of objects may be encapsulated in a Mediator object, responsible for controlling and coordinating the interactions of the group. The objects do not know one another, but only their Mediator; thereby the number of interconnections is reduced.

From the description of this pattern, it does not become clear if the colleague objects communicate with other objects than the mediator. For our formalization, we clarify this ambiguity by adopting the more general choice and assume that the colleague objects may communicate with their environment. The communication among them, however, is performed exclusively through the Mediator object as shown in Fig. 1.

3.2. Components Characteristics.

A distinguished `Mediator` object manages the communication between different colleague objects. We consider here that the set of collegues will not change dynamically. The `Mediator` process has the following form:

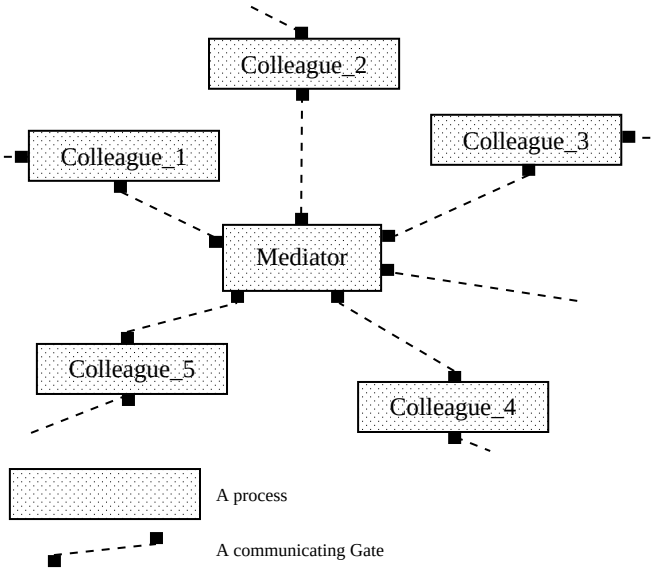

FIG. **1 –.** *The LOTOS Mediator Pattern*

```
process Mediator[InOut1,...InOutn,GateList]:
  func :=
   receive_message
   >> accept  m: message in
        pass_on_message
endproc
```

where the variable *GateList* denotes the gates that connect the Mediator object with its environment, and *InOut1*,... *InOutn* the ones that connect the Mediator object with its colleague objects.

Because the `Mediator` process reads messages from several gates, a receipt of a message is modeled by a behavior expression instead of a simple read action. Thus, the process definition consists of two successive behaviors, *receive_message* and *pass_on_message*, which are separated by `>>`. The `accept` clause means that a message *m* is passed from the behavior *receive_message* (via `exit` clauses) to the behavior *pass_on_message*. The `Mediator` process may terminate (*func* = `exit`) or not (*func* = `noexit`), and the data type *message* is defined algebraically.

In the behavior *receive_message*, the mediator reads incoming messages from some colleague object via some gate *InOuti* or from the environment. Accordingly, this behavior must contain the pattern

```
    InOut1  ? m: message; exit(m)
 [] ...
 [] InOutn  ? m: message; exit(m)
```

In the behavior *pass_on_message*, the mediator passes on the message to the colleague object to which the message

is addressed, according to some predicates. This behavior must contain the pattern below:

```
   [p_1(m)] -> InOut1 ! m ;
   Mediator[InOut1, ... InOutn,GateList]
[] ...
[] [p_n(m)] -> InOutn ! m ;
   Mediator[InOut1, ... InOutn,GateList]
```

Each concrete Mediator design consists of a process called `Mediator` as described above and an arbitrary number of independent colleague processes. Each such colleague must communicate with the Mediator object in the same way as described below:

Colleague_i has a gate *InOuti* and contains an action

InOuti ? m: message

If the colleague object generates messages, the same gate *InOuti* is used to send messages to the mediator. In this case, the process behavior contains actions of the form:

InOuti ! m

3.3. Communication Pattern.

The communication between the mediator and the independent colleagues takes place according to the following LOTOS pattern. All colleague objects behave independently.

```
hide  InOut1, ... InOutn in
    Mediator [InOut1, ... InOutn, GateList]
       |[InOut1, ... InOutn]|
  ( Colleague_1[InOut1, GateList1]
          |||  ...  |||
    Colleague_n [InOutn, GateListn] )
```

3.4. Invariant.

The instantiations of the meta-variables *behav_expr* and *local_def_list* making up the description of a concrete Mediator design must satisfy the following constraints:

- *behav_expr* must conform to the communication pattern given above.
- Each of the processes that occurs in *behav_expr* must show an equivalent behavior [1] with the instantiated components which we have described.

The Mediator style is used to define the Control role [BLMM97] in the PAC (Presentation-Abstraction-Control) [BC91] model of user interfaces.

1. for example with respect to the observational equivalence [Mil86] or to the safety equivalence [Fer89].

4. Formalization of Broker style

4.1. Intuitive Description

The Broker architectural pattern [BMR+96] is used to:

> structure object-oriented distributed software systems with decoupled components that interact by remote service invocations. It is responsible for coordinating communications.

Broker is a complex framework that involves several components: a broker, clients, client-side proxies, servers, server-side proxies and eventually a bridge.

A broker plays the role of a messenger that decouples clients and servers, registers and locates servers, locates client, forwards requests to servers, transmits results and exceptions back to clients. Moreover, it offers an API (Application Programming Interface) to clients and servers that include operations for registering servers and for invoking servers methods.

A server implements functionality or services and offers them through interfaces consisting of operations and attributes.

A client is an application that accesses the services of at least one server. Clients do not know servers, but they must know their broker.

Proxies make the clients of a server communicate with a representative rather than with the server itself. Internet gateways are examples of provided built-in capabilities that behave like proxies. Applications using these facilities do not need proxies. A client-side proxy makes a remote service appear as a local one to the client, and it forwards client requests to the broker. On the other hand, a server-side proxy makes the client requests appear local to the server, and it forwards responses to the broker for the client.

A bridge component is required for hiding communication details when two brokers, belonging to heterogeneous network systems, inter-operate. This case will not be considered here.

4.2. Components Characteristics

The components specified below as LOTOS patterns, corresponds to the one shown in Figure 2.

Broker The broker behavior is to register servers and to pass on messages from a client to the corresponding server. The broker calculates the message target server. If it is not registered, then the broker sends back the message to the client together with the message `false`. When the response from the server arrives, the broker passes it on to the client. All the messages are packed. The Broker patterns

is parameterized by the types of its memory, of the servers identifiers and the packed messages. The type *BROKER_MEM* is provided with the operations `register` and `is_registered` with a server identifier as parameter and `the_server` and `the_client` with a message as parameter.

```
process BROKER
   [BrokerServerProxy,BrokerClientProxy]
   (mem: BROKER_MEM): noexit
:=
  (BrokerServerProxy ?id:ID_SERVER
          [is_registered(mem,id)];
   BrokerServerProxy !false;
   BROKER[BrokerServerProxy,
          BrokerClientProxy](mem))
[](BrokerServerProxy ?id:ID_SERVER
          [not(is_registered(mem,id))];
   BrokerServerProxy !true;
   BROKER[BrokerServerProxy,
          BrokerClientProxy]
         (register(mem, id)))
[](BrokerClientProxy ?pmess:PACKED_MESS;
 (([is_registered(mem,
        the_server(mem,pmess))]
    ->(BrokerServerProxy
             !pmess !the_server(mem,pmess);
       BROKER[BrokerServerProxy,
             BrokerClientProxy](mem)))
 [][not(is_registered(mem,
        the_server(mem,pmess)))]
    ->(BrokerClientProxy  !pmess !false;
       BROKER[BrokerServerProxy,
             BrokerClientProxy](mem))))
[](BrokerServerProxy
             ?pmess: PACKED_MESS;
   BrokerClientProxy
             !pmess !the_client(mem,pmess);
   BROKER[BrokerServerProxy,
          BrokerClientProxy](mem))
endproc
```

Client In the client side there may be several clients behaving independently. All communicate via the same gate `BrokerClientProxy` with the broker. A client initialize itself and then behave as the composition of the two processes `CLIENT_PROXY` and `CLIENT`. The client sends a request to a server via its proxy. The proxy packs the message. It is charged of satisfying the request through the broker facility. The parameters are the types of the client proxy memory, of the requests and of the packed and unpacked responses. The types *REQ* and *PACKED_RESP* are respectively provided with the operations `pack` and `unpack`. The memory is provided with the operations `save_req`, `unsave_req` and `is_for_me`.

```
process CLIENT_SIDE[BrokerClientProxy]:noexit
:=
  A_CLIENT[BrokerClientProxy]
    |||
  OTHER_CLIENT[BrokerClientProxy]
where
  process A_CLIENT[BrokerClientProxy]:noexit
  :=
    initialize >>
    (hide ClientProxy in
    (CLIENT_PROXY[BrokerClientProxy,
                  ClientProxy]
                 (init of CLIENT_PROXY_MEM)
       |[ClientProxy]|
     CLIENT[ClientProxy] ))
  where
    process CLIENT_PROXY
              [BrokerClientProxy,ClientProxy]
              (mem:CLIENT_PROXY_MEM):noexit
    :=
      (ClientProxy ? req: REQ;
       BrokerClientProxy !pack(req) ;
       CLIENT_PROXY
         [BrokerClientProxy,ClientProxy]
         (save_req(mem, req)))
    [](BrokerClientProxy
         ?packed_resp:PACKED_RESP
         [is_for_me(mem, packed_resp)];
       ClientProxy !unpack(packed_resp) ;
       CLIENT_PROXY
         [BrokerClientProxy,ClientProxy]
         (unsave_req(mem,unpack(packed_resp)))
      )
    endproc
  endproc
endproc
```

Server After having initialized itself, a server registers with the broker and then behave as the composition of `SERVER_PROXY` and `SERVER`. The server proxy process receives the packed messages, unpack them and pass them on to the server process to be executed. The execution may take some time so the server manages a queue of requests. When the server sends the answer, the broker sends it packed to the broker. The parameters are the types of the server memory, of the server identifier, of the packed and unpacked requests and responses. These types are provided with the same operations as mentioned for the clients.

```
process A_SERVER[BrokerServerProxy]: noexit
:=
  initialize >>
  register   >>
  hide ServerProxy in
 (SERVER_PROXY[BrokerServerProxy,ServerProxy]
             (id_server of ID_SERVER,
              init of SERVER_PROXY_MEM)
    |[ServerProxy]|
  SERVER[ServerProxy](init of SERVER_MEM))
where
```

```
process SERVER_PROXY
          [BrokerServerProxy,ServerProxy]
          (my_id_server:ID_SERVER,
           mem:SERVER_PROXY_MEM):noexit
:=
  (BrokerServerProxy ?packed_req:PACKED_REQ
                     [id_server=my_id_server];
   ServerProxy !unpack(packed_req);
   SERVER_PROXY[BrokerServerProxy,ServerProxy]
     (my_id_server,
      save_req(mem,unpack(packed_req))))
[](ServerProxy ?response:RESP;
   BrokerServerProxy !pack(response);
   SERVER_PROXY[BrokerServerProxy,ServerProxy]
     (my_id_server,unsave_req(mem, response)))
endproc
endproc
```

4.3. Communication Pattern

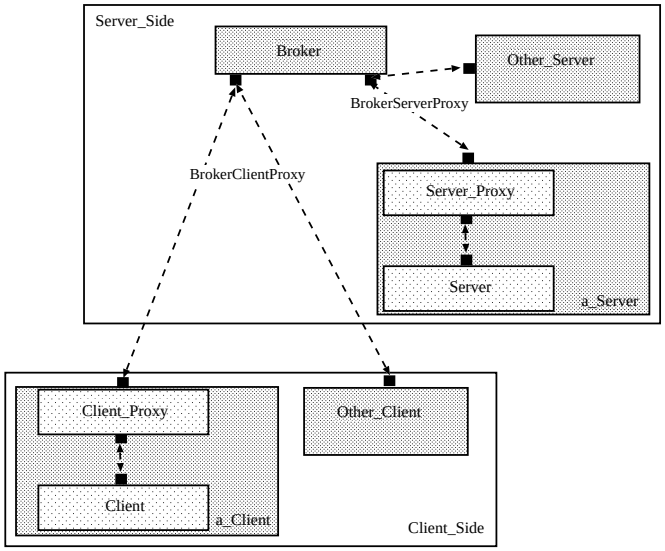

FIG. **2 –.** *Broker Architectural Pattern*

The broker process is *attached* to the server side. We have the following pattern as shown in figure 2:

```
hide BrokerClientProxy in
( SERVER_SIDE[BrokerClientProxy]
    |[BrokerClientProxy]|
  CLIENT_SIDE[BrokerClientProxy])
where
process SERVER_SIDE[BrokerClientProxy]:noexit:=
  hide BrokerServerProxy in
 (BROKER[BrokerServerProxy,BrokerClientProxy]
        (init of BROKER_MEM)
    |[BrokerServerProxy]|
 (A_SERVER[BrokerServerProxy]
    |||
  OTHER_SERVER ))
```

4.4. Invariant

The instantiations of the meta-variables *behav_expr* and *local_def_list* making up the description of a concrete Broker design must satisfy the following constraints:

- *behav_expr* must conform the communication pattern given above.
- Each of the processes that occurs in *behav_expr* must show an equivalent behavior [2] with the instantiated components which we have described.

Let us note that the Broker variant considered here is of indirect communication, where clients always communicate with servers through their broker. There are cases, such as some WWW applications, where clients must communicate directly with servers, thus establishing a direct communication between a client and the server.

5. Comparing styles

When comparing two styles, the communication pattern and the components must be analysed.

> **Definition**: A style S1 is a specialization of another style S2 **iff**
>
> - each component of the style S2 can be matched to a component of the style S1, verifying the characteristics of this component;
> - the communication pattern must be the same modulo the matching;
> - the invariant of the style S2 must be satisfied by the style S1.

Matching the components

- A_CLIENT $\longrightarrow$ Colleague
  A client communicates (via the gate ClientProxy) with the broker via one gate and it (the proxy) contains the action corresponding to the behavior of a colleague. The clients are independent as required in the Mediator style.
- A_SERVER $\longrightarrow$ Colleague
  A server communicates (via the ServerProxy) with the broker via one gate and it (the proxy) contains the action corresponding to the behavior of a colleague. The different servers are independent as required in the Mediator style.
- BROKER $\longrightarrow$ MEDIATOR
  The Broker can be matched with the distinguished Mediator object managing the communication between clients and servers. Using CADP to compare Mediator and

2. for example with respect to the observational equivalence [Mil86] or to the safety equivalence [Fer89].

Broker, it appears that the broker and the mediator have equivalent behavior once the server is registered with the broker.

The communication pattern of Mediator and Broker are equivalent [3]. Finally, as the Broker and the Mediator invariants are similar, the Broker satisfies the Mediator invariant.

Conclusion: Broker is a specialized Mediator.
Using the Broker pattern instead of Mediator, will provide a distributed behavior.

6. Using Broker instead of Mediator in the PAC style

The PAC (Presentation, Abstraction, Control) multi-agent model [BC91] is inspired on the MVC (Model-View-Controller) approach for the development of GUI [Gol84]. Both approaches are based on the idea that the presentation or physical user interaction is kept separate from the semantics or conceptual part of the application.

The PAC model allows the architecture of an interactive system to be structured recursively . The agents are organized according to three basic components:

(i) the *Presentation*, defining the appearance of the system, reflecting its behavior with respect to user input/output. It corresponds to the view-controller pair of MVC.

(ii) the *Abstraction*, or the MVC model, defining the concepts and functionalities of the system, independently of its graphical presentation, and

(iii) the *Control*, absent in MVC, maintaining the coherence and communication between the Presentation and the Abstraction perspectives. These are not allowed to communicate with each other. Communication among PAC agents is only performed by means of the respective controls of the different subsystems of the GUI.

A framework for PAC agents is defined in [BLMM97]. Two main patterns characterize the PAC framework [BLMM97]: Mediator and Strategy. The control class, which implements communication among other PAC agents and maintains consistency between abstraction and presentation, is modeled by the Mediator pattern. The Strategy pattern models the presentation, attaching a view to a controller, allowing to change the way a view responds to user input.

Each PAC agent is specified by instantiating the Mediator patterns as shown in Figure 3.

The use of the PAC model implies to structure the architecture of an interactive application into hierarchical layers. The upper level is constituted by a distinguished interface agent, representing the semantics or data model of the whole application. The presentation or GUI component of the application may be considered absent at this level, and it is represented by the agents of the underlying levels. So, the Broker, playing the role of the Mediator, is located on the server, whose abstraction contains the application data model [LM99]. The Broker shown in Figure 2, is used to decouple the server from the clients or GUI PAC agents as shown in Figure 4. In this way the GUI functionality is distributed among different clients, according to the application requirements. Notice that the Broker must be programmed with the APIs corresponding to the clients and servers involved, and that the data model could be also distributed among other servers. Moreover, on the client GUI agent, the control is constituted by a Client-proxy and by a Mediator communicating with the proxy. Mediator is still used for modeling the interactions between the presentation and the abstraction and for communicating with other sub-agents of the client, according to the usual PAC hierarchy, for non WWW interactive applications. In this case, a client-dispatcher-server [BMR+96] can be used to replace Mediator.

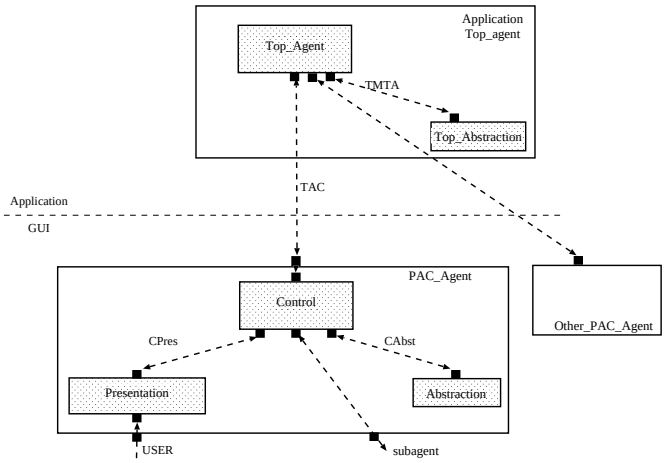

FIG. **3 –.** *PAC Architectural Pattern*

The use of Broker instead of Mediator, as the control of the application agent, has permitted in a quite natural way, the conversion of a standard interactive application into a distributed application.

7. Conclusion

The goal of this paper is to explore the mechanisms of comparison of architectural styles. The objective of comparing styles is to provide criteria of selection of an adequate style, according to the application requirements. Moreover, it enables to explore the combination of two styles or the replacement of a style by another one.

We have proposed a formalization schema for architec-

3. with respect to the observational equivalence [Mil86] tested using CADP

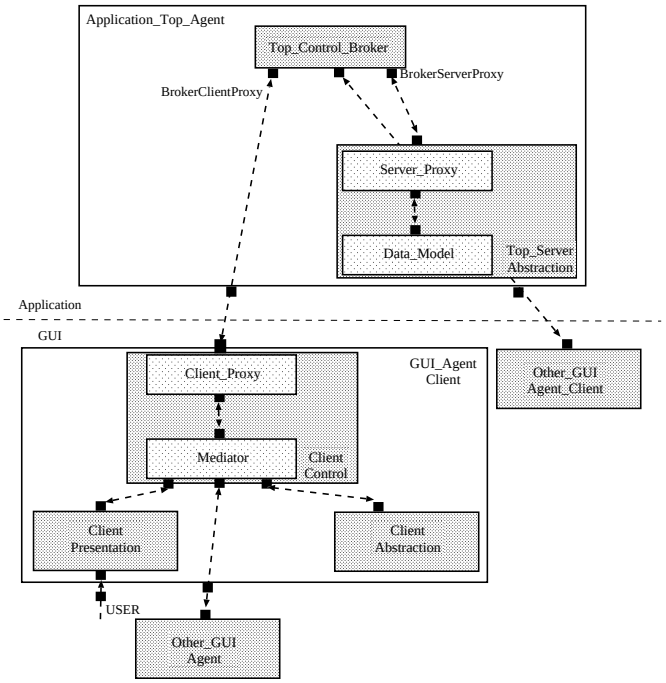

FIG. **4 –.** *Combination of Broker and PAC Architectural Patterns*

tural styles. We have applied it to describe the Mediator and the Broker. In order to compare them, we have defined the notion of specialized style. We have demonstrated that the Broker in its indirect communication variant, is a specialized Mediator. Let us that in [KG98], the classification is based on the interoperability characteristics of components. In this taxonomy, Mediator is a translator-controller and Broker is in addition an extender. Then, we have shown how to obtain a loosely coupled distributed interactive application by replacing Mediator by Broker in the PAC framework.

In case of web applications, a modified PAC style should be considered using client-dispatcher-server pattern to allow direct communications between clients and the web server, in case for example of applets. Or even the application of another architectural pattern involving the couple <subject, observer> pair such as MVC should be studied.

The Observer pattern can be also compared with the Mediator pattern. Mediator is used to encapsulate collective behavior, centralizing communications. Mediator decouples objects by having them refer to each other indirectly. Observer is more used to distribute communication between the <subject, observer> pair. It is better for decoupling objects when they are data dependent, keeping data consistency between the cooperating objects.

Références

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BC91] L. Bass and J. Coutaz. *Developing Software for the User Interface*. Addison-Wesley Co., 1991.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Pratice*. Addison-Wesley, 1998.

[BLMM97] N. Bencomo, F. Losavio, F. Marchena, and A. Matteo. Java implementations of user-interface frameworks. In *Proceedings TOOLS USA'97*, Santa Barbara, August 1997.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, a System of Patterns*. J. Wiley and Sons, Inc, 1996.

[CM98] P. Ciancarini and C. Mascolo. A catalog of architectural styles for mobility. Technical report, Departement of Computer Science, University of Bologna, April 1998.

[Fer89] J.-C. Fernandez. Aldebaran: A tool for verification of communicating processes. Rapport SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, September 1989.

[FGM+92] J-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A toolbox for the verification of lotos programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[HL97] M. Heisel and N. Lévy. Using LOTOS patterns to characterize architectural styles. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development, (TAPSOFT'97-FASE)*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, April 1997.

[HLLM97] M. Heisel, N. Lévy, F. Losavio, and A. Matteo. Formalizing communication aspects of design patterns using LOTOS. 97-R-202, 1997.

[KG98] R. Keshav and R. Gamble. Towardds a taxonomy of architecture integration strategies. In J. Magee and D. Perry editors, editors, *Proceedings of the Third International Software Architecture Workshop (ISAW 3)*, Orlando, Florida, USA, Nov 1998. SIGSOFT, ACM Press.

[LLM98] N. Levy, F. Losavio, and A. Matteo. Comparing architectural styles: Broker specializes mediator. In J. Magee and D. Perry editors, editors, *Proceedings of the Third International Software Architecture Workshop (ISAW 3)*, pages 93–96, Orlando, Florida, USA, Nov 1998. SIGSOFT, ACM Press.

[LM99] F. Losavio and A. Matteo. Multiagent models for designing object-oriented distributed systems. *To appear in Journal of Object-Oriented Programming*, 1999. ISYS, RI/04/98 No. 43-ISSN 1316-6247, Universidad Central de Venezuela, Caracas.

[LOT87] ISO. LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft international standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1987.

[Mik98] T. Mikkonen. Formalizing design patterns. In *Proc. of the 20 th International Conference on Software Engineering, ICSE'98*, pages 115–124, Kyoto, Japan, 1998. IEEE.

[Mil86] R. Milner. A calculus of communicating systems. Technical Report ECS-LFCS-87-7, Laboratory for Foundations of Computer Science, August 1986. First published by SPRINGER VERLAG as Vol 92 of LNCS.

[RMD90] Helm R., Holland I. M., and Gangopadhyay D. Contracts: specifying behavioral compositions in object-oriented systems. In *Proc. OOPSLA'90*, Ottawa, Canada, 1990.

[SG96] M. Shaw and D. Garlan. *Software Architecture, perspectives on an emerging discipline*. Prentice Hall, 1996.