Formalizing Communication Aspects of Design Patterns Using LOTOS

M. Heisel
FG Softwaretechnik
Technische Universität Berlin
Sekr. FR 5-6, Franklinstr. 28/29,
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de

N. Lévy
CRIN-CNRS
Université Henri Poincaré
BP. 239,
F-54506 Vandœuvre-lès-Nancy, France
nlevy@loria.fr

F. Losavio and A. Matteo
Centro ISYS
Universidad Central de Venezuela
Ap.47567 Los Chaguaramos,
1041-A Caracas, Venezuela
{flosavio,amatteo}@anubis.ciens.ucv.ve

ABSTRACT
In the research of design patterns, the main activities are the discovery or invention of new patterns, the application, and the specification or formal description of design patterns. This paper aims at formalizing the communication aspects of a subset of the design patterns defined by Gamma et al. [8]. The LOTOS specification language is used to formalize these communication aspects, where the objects are modeled as processes and the messages between objects are expressed by LOTOS communication patterns. Our formalization not only contributes to a semantic foundation of design patterns, but also supports validation and rapid prototyping.

KEYWORDS
Architectural style, Design Patterns, Formal Methods, LOTOS, Object Oriented Technology, Software Architecture.

1 INTRODUCTION

Patterns [1] are widely applied in software construction to describe a problem and the core of its solution. Object-oriented design patterns as the ones described by Gamma et al. [8] represent frequently used ways to combine classes or associate objects to achieve a certain purpose. One problem with design patterns is that their descriptions are often ambiguous, and additional semantics is frequently explained by examples.

Buschmann et al. [4] claim formal methods do not apply to patterns, because "Formalisms ... tend to describe particular issues very precisely, but do not allow for the variation that is inherently embedded into every pattern". Moreover, they remark that there is no formalism "suitable for describing the benefits and liabilities of a pattern".

In contrast to this opinion, we deem it to be worthwhile to attempt a formalization of design patterns for the following reasons:

- A formal description of (aspects of design patterns) clarifies the ambiguous points of the informal description.
- Unambiguous descriptions make patterns better comprehensible.
- Designers are provided with criteria to decide on the applicability of a certain pattern.
- Concrete designs based on patterns may be subject to proofs, analyzes, and other forms of validation [13], e.g., animation
- The development of pattern tools [6] is facilitated by formal descriptions.
- A formal description of patterns does not prevent their variation and evolution.

This paper aims at demonstrating that it is possible to formalize some aspects of the design patterns. These aspects are specified using the formal description language LOTOS [3], 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) [7], can be employed to analyze and animate instances of patterns. 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 processes.
- Messages between objects are expressed by LOTOS communication patterns.

Each formal pattern description consists of three parts:

- requirements on the processes specifying the objects contained in an instance of the design pattern,
- a LOTOS communication pattern defining its top-level behavior, and
- constraints, which provide sufficient conditions for a design description to be an instance of the design

pattern. These conditions can be checked mechanically.

The contribution of this approach is two-fold: First, it provides a semantic foundation of the communication aspects of design patterns. Second, it supports the practical use of design patterns and the validation of concrete designs in the following way:

- Design takes place on the level of design patterns. Designers need not be bothered with formal details.
- The formalization enables a routine translation of the design into a LOTOS specification.
- This specification is an executable prototype of the design that can be animated and analyzed using existing tools.

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. Indeed, each of the patterns presented in the following is a variant of an architectural style. Architectural styles can be formalized in much the same way as communication aspects of design patterns [10].

This paper is structured as follows: in Section 2, we briefly present the LOTOS specification language. In Section 3, we present the formalizations of three design patterns that all make use of a distinguished object that serves to distribute messages: Facade, Mediator, and Strategy. These patterns are used for the development of graphical user interfaces, as sketched in Section 4. Sections 5 and 6 present the formalizations of the patterns Chain of Responsibility and Observer, which are based on different principles. A discussion of what has been achieved concludes the paper.

2 INTRODUCTION TO LOTOS

LOTOS [3] 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 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 the syntactic form

```
process process_name[gate_list](params): func:=
  behaviour behav_expr
  where local_def_list
endproc
```

where *func* indicates whether the process may terminate (*func*= `exit`) or not (*func*= `noexit`). The behavior expression describes the sequences of observable actions that may occur at the gates of the process. Process definitions may include instantiations of processes.

The choice operator `[]` is used when alternative behaviors are allowed. The behavior expression `P1 [] P2` expresses that exactly one of the two processes 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 via 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` via gate `g`. The other process must contain an action `g ! exp` that writes a value `exp` of type `t` onto 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 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`.

Design Descriptions 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

```
behaviour behav_expr where local_def_list
```

LOTOS *patterns* are obtained from LOTOS specification by abstraction, i.e. by replacing concrete LOTOS expressions by metavariables. 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 metavariables are set in *italics teletype*.

3 PATTERNS WITH A DISTINGUISHED ADMINISTRATIVE OBJECT

We now present the formalizations of three patterns, each of which is based on a distinguished object that serves to pass on messages or requests to the other components of the pattern. Recall that each formalization consists of (i) a characterization of the components of the pattern, (ii) a LOTOS pattern describing how the components communicate, and (iii) constraints that provide sufficient conditions for a concrete design description to be an instance of the formalized design pattern.

3.1 The Facade Design Pattern

The intent of the Facade design pattern is [8]:

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The Facade design goal is to minimize communication and dependencies between subsystems. Facade can be applied, for example, for layering a subsystem. It does not prevent applications from using subsystem classes if they need to, leaving the choice between ease of use and generality.

3.1.1 Component Characteristics. As shown in Fig. 1, a distinguished `Facade` object receives service requests from the environment and passes them on to other colleague objects. The colleague objects can communicate with each other and also with the environment.

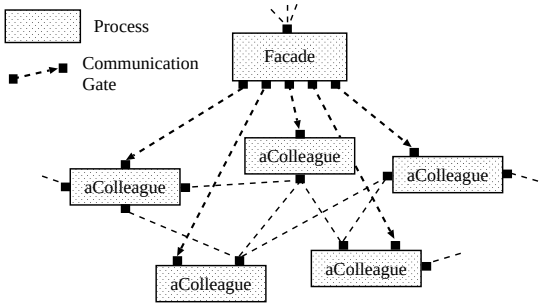

Figure 1: The LOTOS Facade Pattern

The following process models the `Facade` object:

```
process Facade [REQUEST, IN_1, ..., IN_n]: func:=
   REQUEST ? r: request;
   pass_on_request
endproc
```

This process reads service requests from the environment via the gate *REQUEST*[1] and then passes them on to the colleague object (via one of the gates *IN_1, ..., IN_n*) that can process the request, details of which are defined in the behavioral expression *pass_on_request*. The `Facade` process may terminate (*func* = `exit`) or not (*func* = `noexit`). The data type *request* must be defined algebraically. It can be structured to allow the handling of complex requests.

In the behavioral expression *pass_on_request*, the Facade object passes on the request to a colleague object that can serve it, according to some predicates *p_j*. Accordingly, the definition of this expression must contain the following pattern:

```
    [p_1(r)] -> IN_1 ! r ; beh_1
[] ...
[] [p_n(r)] -> IN_n ! r ; beh_n
```

where the metavariables *beh_i* usually will contain a recursive call

```
Facade [REQUEST, IN_1, ... IN_n]
```

so that an indefinite number of requests can be treated.

Each Facade design consists of a `Facade` object modeled as described above and an arbitrary number of colleague objects. Each such object *Colleague_i* has a gate *IN_i* which it uses to communicates with the `Facade` object. To receive requests from the `Facade` object, it must contain an action

```
IN_i ? r: request
```

This concludes the characterization of the involved components.

3.1.2 Communication Pattern. The communication between the `Facade` object and the colleague objects takes place according to the pattern

```
hide  IN_1, ... IN_n in
  Facade [REQUEST, IN_1, ... IN_n]
    |[IN_1, ... IN_n]|
  behav
```

where *IN_1, ...IN_n* is the list of all gates that connect the Facade and the colleague objects. The `hide` clause hides the communication between the Facade and the colleague objects from the environment, i.e., for the environment only the gates *REQUEST* and the gates that connect the colleague objects with one another or the environment are visible.

The behavior *behav* represents the communication between the colleague objects, which is a parallel composition according to the pattern

[1] In this definition, there is only one gate *REQUEST*. The LOTOS pattern can easily be generalized to allow for several external gates.

```
(Colleague_1[IN_1, int_gate_list_1, env_gate_list_1]
    |[int_gate_list_1]|
 ...
    |[int_gate_list_n-1]|
 Colleague_n[IN_n, int_gate_list_n, env_gate_list_n]
)
```

If some colleague objects need not communicate, their synchronization list is empty, and the interleaving operator `|||` instead of the parallel operator `|[]|` can be used.

3.1.3 Constraints. The instantiations of the metavariables ***behav_expr*** and ***local_def_list*** making up the description of a concrete Facade 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 conform to the description given in the component characterization.

The Facade design pattern will be used in Section 4 to design the overall structure of the interface of a graphical editor.

3.2 The Mediator Design Pattern

The intent of the Mediator design pattern as shown in Fig. 2 is [8]:

> 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.

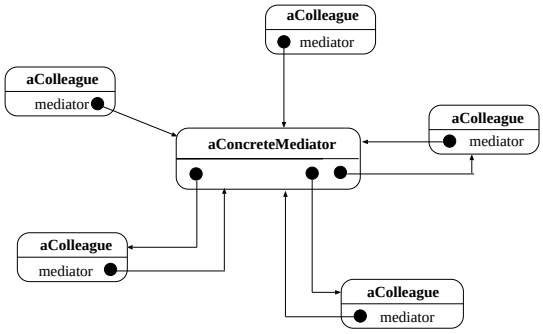

Figure 2: The Mediator Pattern of [8]

From the description of this pattern given in [8], 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. 3.

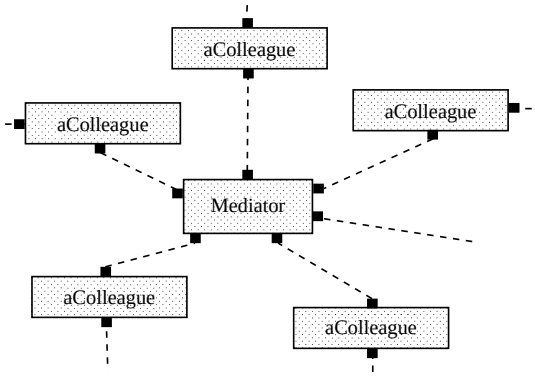

Figure 3: The LOTOS Mediator Pattern

3.2.1 Component Characteristics. A distinguished `Mediator` object manages the communication between different colleague objects. The `Mediator` process has the following form:

```
process Mediator [IN_OUT_1, ... IN_OUT_n, gate_list]:
  func :=
   receive_message
   >> accept  m: message in
        pass_on_message
endproc
```

where ***gate_list*** denotes the gate that connect the Mediator object with its environment, and ***IN_OUT_1 ... IN_OUT_n*** denote the gates that connect the Mediator object with its colleague objects.

Because the `Mediator` process—in contrast to the `Facade` 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 components, ***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***. As in the Facade formalization, 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 ***IN_OUT_i*** or from the environment. Accordingly, this behavior must contain the pattern

```
    IN_OUT_1  ? m: message; exit(m)
 [] ...
 [] IN_OUT_n  ? 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 a pattern similar to the pattern given for the behavior ***pass_on_request*** of the Facade pattern in Section 3.1.1.

Each concrete Mediator design consists of a process **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 in Section 3.1.1.

3.2.2 Communication Pattern. The communication between the mediator and the independent colleagues takes place according to the following LOTOS pattern. In contrast to the Facade design pattern, all colleague objects behave independently.

```
hide  IN_OUT_1, ... IN_OUT_n in
    Mediator [IN_OUT_1, ... IN_OUT_n, gate_list]
       |[IN_OUT_1, ... IN_OUT_n]|
  ( Colleague_1[IN_OUT_1, gate_list_1]
       |||  ...  |||
    Colleague_n [IN_OUT_n, gate_list_n]
  )
```

3.2.3 Constraints. These are the same as for the Facade pattern, see Section 3.1.3.

In Section 4, we will use the Mediator design pattern to define the Control role in the PAC (Presentation-Abstraction-Control) [5] model of user interfaces.

3.3 The Strategy Design Pattern

The intent of the Strategy design pattern is [8]:

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that uses it.

For example, different geometrical shapes, e.g., triangle or rectangle, may make up different contexts. In each context, one may click the mouse once or twice, which calls different algorithms, depending on the context.

3.3.1 Component Characteristics. A distinguished **Strategy** object serves to pass on calls issued in different contexts to the *concrete strategy* objects that implement the appropriate algorithm to be called, as shown in Fig. 4.

The pattern of the process **Strategy** is the following:

```
process Strategy [CONT_1, ..., CONT_n,
                  IN_1,1, ... IN_1,k1,
                  ...,
                  IN_n,1, ..., IN_n,kn] : func :=
    CONT_1 ? c: call; pass_on_call_1
```

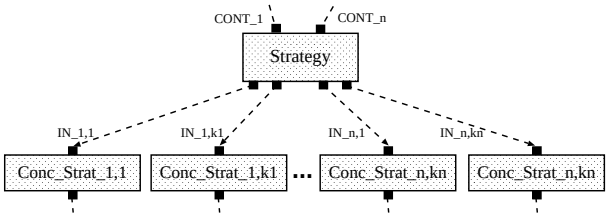

Figure 4: The LOTOS Strategy Pattern

```
  [] ...
  [] CONT_n ? c: call; pass_on_call_n
endproc
```

The **Strategy** object first reads incoming calls from the different contexts via one of the gates ***CONT_1, ..., CONT_n*** and then passes on the call to the various concrete strategies that define the algorithm called by the context. Each behavior ***pass_on_call_i*** must contain a pattern analogous to the one given in Section 3.1.1.

Each concrete Strategy design consists of a process **Strategy** as described above and an arbitrary number of concrete strategies that are associated with contexts. Each concrete strategy communicates with the Strategy object in the same way as described in Section 3.1.1.

3.3.2 Communication Pattern. The communication between **Strategy** and the concrete algorithm implementations takes place according to the pattern

```
hide  IN_1,1, ... IN_1,k1,
      ...,
      IN_n,1, ..., IN_n,kn in
  Strategy [CONT_1, ..., CONT_n,
            IN_1,1, ... IN_1,k1,
            ...,
            IN_n,1, ..., IN_n,kn]
         |[IN_1,1, ... IN_1,k1,
           ...,
           IN_n,1, ..., IN_n,kn]|
( Conc_Strat_1,1[IN_1,1,env_gate_list_1,1]
         ||| ... |||
  Conc_Strat_1,k1[IN_1,k1,env_gate_list_1,k1]
         ||| ... |||
  Conc_Strat_n,1 [IN_n,1, env_gate_list_n,1]
         ||| ... |||
  Conc_Strat_n,kn [IN_n,kn, env_gate_list_n,kn]
)
```

3.3.3 Constraints. These are the same as for the Facade pattern, see Section 3.1.3.

In Section 4, the Strategy pattern will be used to encapsulate different algorithms for creating and deleting different graphical items.

3.4 Comparing the Three Patterns

The three patterns we have presented are based on the same principles. All of them make use of a distinguished

object that serves to pass on messages or requests to the other components of the pattern. Thus, they are variations of the event-action architectural style [10].

The Facade object receives messages from the environment that are passed on to the colleague objects. Hence, the environment is encouraged to communicate with the Facade object instead of with the colleague objects directly. The interaction *between* the colleague objects, however, is not of interest for this pattern. These principles are reflected in our formalization by the following facts:

- The `Facade` process does not read from the gates *`IN_1, ..., IN_n`* that connect it with the colleague objects but only from the gates that connect it with the environment.
- The processes modeling the colleague objects are only required to *read* the messages sent to them by the `Facade` process. Hence, the connection between the Facade and the colleague objects is one-directional.
- The processes modeling the colleague objects may communicate with one another, as is expressed by use of the parallel operator `|[...]|` in the communication pattern of Section 3.1.2.

In contrast, the Mediator object is concerned with managing the *internal* communication between the colleague objects. The communication of the various objects with the environment is not a concern of this pattern. This concept is reflected in our formalization as follows:

- The `Mediator` process must read incoming messages from the colleague objects.
- The colleague objects are not allowed to communicate with one another, as expressed by using the interleaving operator `|||` in the communication pattern of Section 3.2.2.
- Each object contained in the pattern instance may have gates that connect it with the environment.

From a technical point of view, the Strategy pattern is a combination of the Facade and Mediator patterns. As in the Facade pattern, the communication between the `Strategy` process and the processes modeling the concrete strategies is one-directional, because the purpose of the Strategy object solely is to select the appropriate algorithm. For these algorithms, there is no need to communicate. Hence, the respective processes act independently, as is the case for the colleague processes in the Mediator pattern.

4 Example: Development of Graphical User Interfaces Using the PAC Framework

The PAC (Presentation, Abstraction, Control) multi-agent model [5] is inspired on the MVC (Model-View-Controller) approach for the development of graphical user interfaces (GUI) [9]. 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 to structure recursively the architecture of an interactive system. 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-controler 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 [14] for PAC agents is defined in [2]. Two main patterns characterize this framework: Mediator and Strategy. The control class, which implements communication 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.

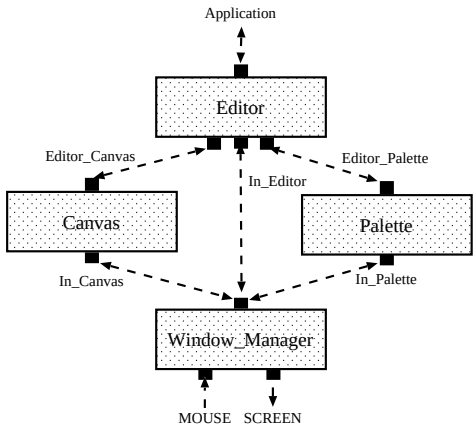

Figure 5: Application of Facade Pattern

This framework is used to specify a simple graphical editor containing three PAC agents: The *editor* component implements the interface of the GUI with the application. It uses a *canvas* (implementing the graphics),

and a *palette* (allowing the user to select different contexts, e.g., for drawing circles or rectangles). The specification is obtained simply by instantiating the framework. Fig. 5 shows the overall architecture of the graphical editor. The three components `Editor`, `Canvas` and `Palette` are three PAC agents. The `Window-Manager` acts as a facade for the other components, thus providing an interface between the PAC agents and the user.

Each PAC agent is specified by instantiating the Mediator pattern as shown in Fig. 6.

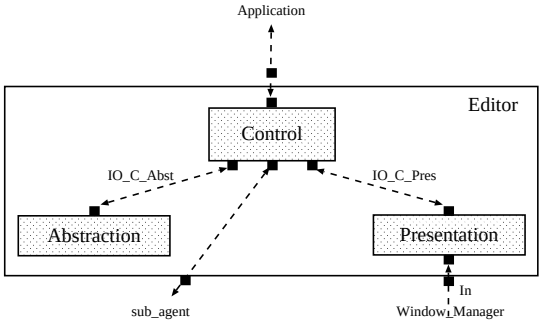

Figure 6: Application of Mediator Pattern

An example of the Strategy behavior is to relate the way the context of a presentation or view changes with respect to the user inputs, encapsulating the corresponding algorithms required for handling these inputs as shown in Fig. 7.

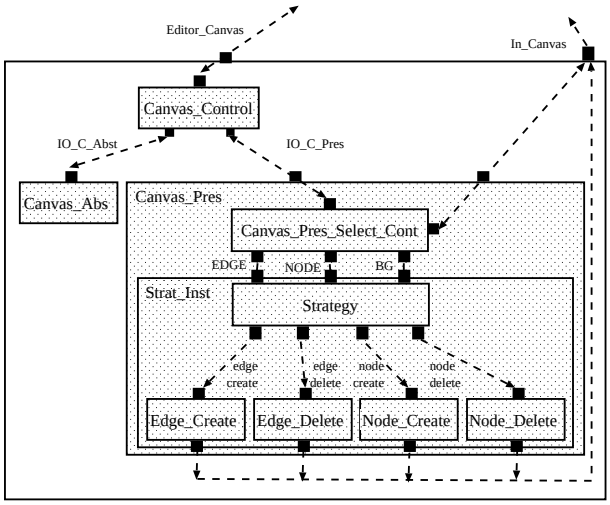

Figure 7: Application of Strategy Pattern

In this example, we consider the contexts *edge*, *node*, and *background*. In the contexts *edge* and *node*, one may create and delete items by mouse clicks. On the background, however, mouse clicks have no effect.

In the following, we present two more formalizations of communication in design patterns. These are based on different principles than the ones presented in Section 3, and correspond to different architectural styles.

5 THE CHAIN OF RESPONSIBILITY DESIGN PATTERN

The intent of the Chain of Responsibility design pattern, as shown in Fig. 8, is [8]:

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

The Chain of responsibility design allows to send requests to an object implicitly through a chain of candidate objects. Any candidate may handle the request depending on conditions which are dynamically determined. The number of candidates is open-ended. An example illustrating an application of the Chain of Responsibility is the handling of requests for an on-line help system. The user can obtain help information on any part of the interface, just by clicking on it. The help that is provided depends on the part of the interface that is clicked and on its context.

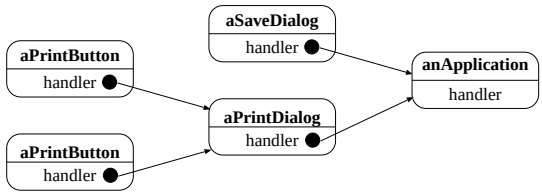

Figure 8: The Chain of Responsibility Pattern of [8]

5.1 Component Characteristics

As shown in Fig. 9, an object in the chain (called a *handler*) is modeled by a process that receives requests from incoming gates, and either handles the request, or passes it on to the next object in the chain. Communication with the environment will be necessary to deliver the results of a request that has been handled.

An object in this design pattern is therefore characterized by the following behavior:

```
in_gate ? v: TYPE;
  ( [Prop_1(v)] -> out_gate ! v ; recursive_call
  [][Prop_2(v)] -> int_beh      >> recursive_call
  )
```

where ***Prop_1*** and ***Prop_2*** are predicates deciding whether the request ***v*** is treated or passed on. ***int_beh*** is a behavioral expression describing the handling of the request and the ***recursive_call*** is just the mechanism for continuing the process. Its gates are divided into a list of incoming gates ***in_gate_list***, at

most one gate *out_gate* connecting it with the next handler in the chain, and a list of gates *env_gate_list* connecting it to the environment. A process modeling a handler does not write on gates of its *in_gate_list* and does not read from *out_gate*.

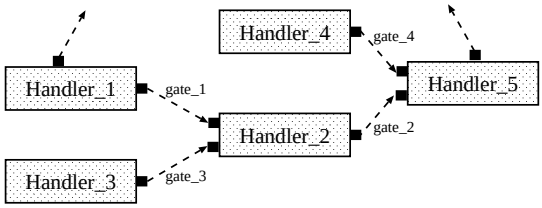

Figure 9: The LOTOS Chain of Responsibility Pattern

5.2 Communication Pattern

Two handlers communicate via their common gates. For example, in Fig. 8, the handlers `aPrintButton` and `aPrintDialog` exhibit the communication behavior

```
aPrintButton[gate_1]
   |[gate_1]|
aPrintDialog[gate_1, gate_2, gate_3]
```

where gates with the environment are not taken into account. When adding a third handler `anApplication` synchronizing with the previous system via the gate `gate_3`, the following behavior is obtained:

```
  (aPrintButton[gate_1]
   |[gate_1]|
aPrintDialog[gate_1, gate_2, gate_3] )
   |[gate_3]|
anApplication[gate_3, gate_4]
```

Note that, since we have a chain, each synchronization list has length 1. Hence, the general communication pattern of a Chain of Responsibility design has the form

```
hide gate_1, gate_2, ... gate_n-1 in
  ( ... ((Handler_1[gate_list_1]
             |[gate_1]|
          Handler_2[gate_list_2]
         )
            |[gate_2]|
          Handler_3[gate_list_3]
        ) ...
            |[gate_n-1]|
          Handler_n[gate_list_n]
  )
```

We have used "..." instead of an inductive definition for better comprehensibility of the communication pattern.

5.3 Constraints

Again, we state the constraints in terms of the top-level behavior *behav_expr* and the *local_def_list*:

- All synchronization gates (i.e. the values given to *gate_1*, ..., *gate_n-1*) occurring in *behav_expr* are different. This means that such a gate connects only two handlers.
- Each gate occurring in some synchronization list of *behav_expr* occurs exactly twice in the gates of the processes *Handler_1*, ..., *Handler_n* defined in *local_def_list*. This means that a gate connecting two handlers cannot be re-used as an external gate.
- Each of the processes that occur in *behav_expr* must conform to the characterization given above. The gates of a process representing connections with other handlers are exactly the ones that are used as a synchronization gate *gate_i*. The direction of such a gate can be determined from the process definition.
- There are no cycles allowed in the chain. This is ensured by the fact that each synchronization list has length 1.

Note that some of the formalizations given in this paper (in Sections 3 and 6) contain a distinguished component. This results in a relatively detailed characterization of the other components of the design pattern. One can state requirements concerning the communication of the other components with the distinguished one. Further constraints are not necessary. In contrast, the Chain of Responsibility pattern does not have a distinguished component. This allows only a weak characterization of the components, but leads to non-trivial constraints concerning the communication between the different components.

The Chain of Responsibility design pattern is structurally similar to the pipe/filter architectural style [10], where no cycles are allowed but the components do not react the same way.

6 THE OBSERVER DESIGN PATTERN

The intent of the *observer* design pattern is [8]:

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The Observer design defines and maintains a dependency between objects. The classical example illustrating the Observer behavior is the MVC (Model-View-Controller) approach [9] for separating the presentational aspects of the user interface (Views) from the underlying application data (Model).

6.1 Component Characteristics

There is one distinguished `Subject` object (and process), and several independent observer objects as shown in Figure 10. Each observer may change the subject. After a change, all observers must be notified of the change. However, only one observer at a time may change the subject (the subject is locked during a change). There is no interaction between observers: they behave independently and only communicate with the subject and the environment.

Three kinds of interactions between the subject and the observers are possible: an observer may read the subject, it may change the subject independently of its previous state, and it may change the subject, taking its previous state into account.

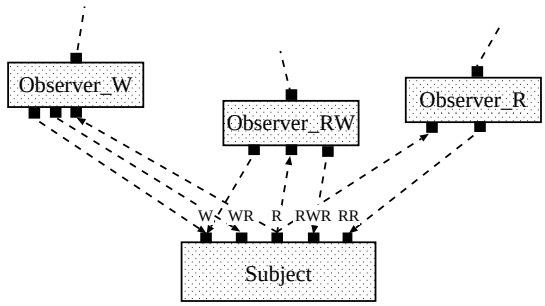

Figure 10: The LOTOS Observer Pattern

If an observer wants to change the subject, it sends the message `WR` (write request). This causes the subject to set a lock. Only then can the new value be passed, using the gate `W` (write). If an observer wants to read the subject, it sends the message `RR` (read request). If no lock is set the value is passed via the gate `R` (read). It may happen that a value to be written into the subject depends on a value that was read previously. In this case, no other write operation should be allowed between the read and the write action. For this purpose, the message `RWR` (read/write request) is used.

Each process sending a request must also send a unique identification. This prevents other processes from accessing the subject during a transaction. The process implementing the subject is defined as follows:

```
process Subject [RR, R, WR, W, RWR]
                (sub: subject, is_locked: BOOL,
                 for_whom: id): noexit :=
  [ is_locked = false ]
    -> (  RR ? who:id;
          R  ! who ! sub;
          Subject[RR, R, WR, W, RWR]
                 (sub, false, for_nobody)
       []
          WR ? who:id;
          Subject[RR, R, WR, W, RWR]
                 (sub, true, who)
       []
          RWR ? who: id;
          Subject[RR, R, WR, W, RWR]
                 (sub, true, who) )
[][ is_locked = true ]
    -> (  W ? who:id ? nsub:subject [who=for_whom];
          R ! nsub ;
          Subject[RR, R, WR, W, RWR]
                 (nsub, false, for_nobody)
       []
          R ? who:id ;
          R ! who ! sub ;
          W ? who:id ? nv:subject [who=for_whom];
          R ! nsub ;
          Subject[RR, R, WR, W, RWR]
                 (nsub, false, for_nobody))
endproc
```

The process `Subject` has the gates `RR, R, WR, W, RWR` and the parameters `sub` representing the state of the subject, `is_locked` and `for_whom`. It does not terminate, as indicated by the keyword `noexit`. If the lock is not set, either a read request can be served, or the lock can be set because of a write or read/write request. If the lock is set, either a new value for the subject read via the gate `W`, or the state of the subject is output on gate `R`, followed by reading a new value via gate `W`. These actions can only take place if the same process that sent the request participates in them, as expressed by the guard `[who=for_whom]`.

Whenever the state of the subject is changed, the new value is sent to all observer processes, using the action `R ! nsub`. The new value of the subject becomes the new parameter of the process, and the lock is reset. The constant *for_nobody* indicates that access to the subject is not reserved for a particular observer.

Each concrete observer design consists of a process `Subject` as defined above and an arbitrary number of independent observers. Each of these observers must be able to receive the notification of subject change, that is an `R` message, at any time. To guarantee this behavior, the observer must not be blocked by another communication with the environment. If an observer may be involved in such a communication, it must be embedded in a choice expression as follows:

```
     R  ? nsub : subject ; process_1
 []  process_2
```

where *process_2* expresses the communication with the environment and *process_1* what is to be done after a subject notification. This means that each observer process is ready to read a new subject value at all times.

Apart from this requirement, each observer process must contain at least one of the following behavioral patterns: A *read* behavior is defined by the pattern

```
    RR ! me ;
    R  ? who: id ? v : value [who = me]
```

where *me* is the identification of the observer process. A *write* behavior is defined by the pattern

```
WR ! me ;
W  ! me ! nsub
```

where *nsub* is the new value of the subject. A *read/write* behavior is defined by the pattern

```
RWR ! me ;
R  ? who: id ? sub : subject [who = me]
```

followed, in all the branches of the process, by writing access to the subject according to the pattern

```
W  ! me ! nsub
```

for a new subject value *nsub*. This condition can be decided syntactically.

6.2 Communication Pattern

The communication between the subject and the observers is expressed by the following pattern:

```
hide   RR, R, WR, W, RWR in
  Subject [RR, R, WR, W, RWR]
          (init of subject, false, for_nobody)
          |[ RR, R, WR, W, RWR ]|
  (Observer_1[gate_list_1]
          |[R]|  ...  |[R]|
   Observer_n [gate_list_n] )
```

All observers behave independently of each other except for the broadcasting on R. For every *Observer_i*, its *gate_list_i* must contain the gate R and some of the gates RR, WR, W, RWR. The subject and observers must synchronize on these gates, as expressed by the synchronization list |[RR, R, WR, W, RWR]|.

6.3 Constraints

For an observer design, we have the constraints that the *behav_expr* must conform to the communication pattern given above, and that each process occurring in *behav_expr* must conform to other behavioral constraints as defined in the process characteristics.

This pattern is a variant of the repository architectural style [10], the difference being the broadcasting of the new value of the subject.

7 CONCLUSION

We have presented the formalization of the communication aspects of several design patterns, thereby resolving ambiguities and defining an unambiguous semantics of the communication between the various objects of the patterns. Making the communication structure explicit encourages the definition of variants of these patterns by varying the parts of the formalization, e.g., the communication pattern, or the constraints.

Note that some of the formalizations given in this paper (in Sections 3 and 6) contain a distinguished component. This results in a relatively detailed characterization of the other components of the design pattern. One can state requirements concerning the communication of the other components with the distinguished one. Further constraints are not necessary. In contrast, the Chain of Responsibility pattern does not have a distinguished component. This allows only a weak characterization of the components, but leads to non-trivial constraints concerning the communication between the different components.

Besides contributing to a semantic foundation of design patterns, our work allows to make explicit the relation between design patterns and architectural styles. The architecture of a software system defines that system in terms of computational components and interactions among those components [15]. Some design patterns, such as those we have studied, can be regarded as defining an architecture, because they state how different objects have to interact to achieve a certain purpose. Both architectural styles and design patterns can be composed hierarchically.

We have applied our formalization to define a prototype of a graphical editor, as described in Section 4. It was remarkably simple to generate the LOTOS specification from the design of the graphical editor, expressed as a hierarchical combination of different design patterns. The generated specification was checked and animated with the CADP tool [7]. This shows that a formalization may not only contribute to precision, but also facilitate the practical usage of patterns and provide valuable validation of designs based on patterns.

In the future, we intend to provide machine support for our approach. Two development frameworks designed by two of the authors are good candidates for achieving this goal. The first is the knowledge representation mechanism called *strategies* [11]. Strategies form a generic framework in which development knowledge for various software development activities can be expressed. The second one, called PROPLANE [12], aims at modeling specification construction and providing specifiers with tools to support them during the development process.

REFERENCES

[1] C. Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.

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

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad,

and M. Stal. *Pattern-Oriented Software Architecture, a System of Patternsx*. J. Wiley and Sons, Inc, 1996.

[5] J. Coutaz. *Developing Software for the User Interface*. Addison-Wesley Co., 1991.

[6] A. Eden, J. Gil, and A. Yehudai. Automating application of design patterns. *Object-Oriented Programming*, 10(2), May 1997.

[7] J. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In L. A. Clarke, editor, *Proc. ICSE'14*. ACM, 1992.

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

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

[10] M. Heisel and N. Lévy. Using LOTOS patterns to characterize architectural styles. In *Proc. 7th TAPSOFT'97*, volume 1214 of *LNCS*, 1997.

[11] M. Heisel, T. Santen, and D. Zimmermann. Tool support for formal software development: A generic architecture. In *Proc 5-th ESEC*. LNCS 989, 1995.

[12] N. Lévy and J. Souquières. Modelling specification construction by successive approximations. In *To appear In 6th AMAST'97*. LNCS, 1997.

[13] 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.

[14] W. Pree. *Design Patters for Object-Oriented Software Development*. Addison-Wesley, 1995.

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