Using LOTOS Patterns to Characterize Architectural Styles

Maritta Heisel[1] and Nicole Lévy[2]

[1] FG Softwaretechnik, Technische Universität Berlin, Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin, Germany, heisel@cs.tu-berlin.de
[2] CRIN-CNRS, BP. 239, F-54506 Vandœuvre-les-Nancy, France, nlevy@loria.fr

Abstract. We show how the formal description language LOTOS can be used to define software architectures and how patterns over LOTOS can serve to characterize architectural styles. We characterize styles by giving characteristics of the involved processes, a top-level communication pattern, and constraints that are sufficient conditions for a concrete architectural description to be an instance of a given style. Three style characterizations are presented and illustrated by an example.

1 Introduction

Architectural styles are a mechanism to make system design knowledge explicit and thus amenable to reuse. They characterize designs in terms of the system components and the connectors that enable communication between components [AAG93]. Problems are how to represent styles in such a way that unambiguous criteria can be stated to decide whether a given design conforms to some style and how a style representation can help to develop concrete architectures.

Informal circle-and-line drawings have shown their limitations and today, formal languages are proposed to represent software architectures. New languages for architectural descriptions have been developed, but they are still in a maturing phase, and few are provided with tools [Cle96].

In this paper, we address these problems in three ways: first, we demonstrate that LOTOS [BB87] is a suitable language to express architectural designs. Second, we contribute to a clarification of the meaning of architectural styles by characterizing such styles as LOTOS patterns. Third, we show how the patterns can support designers in the development of concrete software architectures.

LOTOS as an Architectural Description Language. Using LOTOS to express architectural designs has several advantages:

- LOTOS consists of two parts, an algebraic specification language to define data, and a process algebra to define the behavior of a system. Hence, the communication between system components in an architecture can be described using the process algebraic parts of LOTOS, and the algebraic specification language can be used to specify the data transformations that are performed by the system.
- Architectural descriptions in LOTOS are formal and hence have an unambiguous semantics. They can be subject to proofs and analyses.

– Existing tools, such as CADP (Caesar/Aldebaran Distribution Package) [FGM+92], can be employed to analyze and animate architectures defined in LOTOS.
– LOTOS is an ISO standard. The use of a standardized language relieves system designers of the burden to learn an extra architectural description language. These can be quite rich and complex, see e.g. [LKA+95].

Style Characterizations. We characterize an architectural style by (i) requirements on the processes specifying the components of a system, (ii) a communication pattern defining its top-level behavior, and (iii) constraints, which provide sufficient conditions for an architectural description to be an instance of the style. These conditions can be checked mechanically.

Design Support. The style characterizations provide designers with patterns that simply have to be instantiated to obtain a concrete architecture. An instantiation can be performed recursively such that an architecture can combine several architectural styles. Architectures can be mechanically checked for conformance with the style. Furthermore, the architectural descriptions can be analyzed and animated using existing tools. No new tools need to be developed.

In Section 2, we explain the general approach we take to express architectural designs in LOTOS and styles as LOTOS patterns. The approach is illustrated by characterizing three architectural styles: repository (Section 3), pipe/filter (Section 4) and event-action (Section 5). In Section 6, we present three different designs for a robot, following the three architectural styles. The tool CADP is used to compare the alternative designs. The concluding section discusses our approach in the context of related work.

2 Expressing Architectural Designs and Styles with LOTOS

Architectural designs and styles are usually described in terms of *components* and *connectors* between them. In our approach, system components are modeled as *processes*. These processes usually perform some data transformation. They may consist of another architectural description, representing the design of a subsystem. In this way, hierarchical composition of architectures is possible. Connectors are no separate syntactic entities but are realized by the kind of communication that takes place between the component processes.

LOTOS specifications are composed of interacting processes. They can be parameterized by abstract data types. A process can exchange typed values with another process 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. 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.

We use this kind of communication by rendez-vous to describe the communication between the components of a system. Data are described using abstract data types with conditional equations and an initial semantics. They are used for describing process parameters and values exchanged by the processes via gates.

Each architectural description must be a valid LOTOS expression, regardless of the style it belongs to. It consists of two parts. The *behavior* part describes the overall behavior of the architecture, 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 syntactical structure of an architectural description is

`behaviour` ***`behav_expr`*** `where` ***`local_def_list`***

LOTOS *patterns* are obtained from LOTOS by abstraction, i.e. by replacing concrete LOTOS expressions by metavariables. Both parts of an architectural 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`***.

A characterization of an architectural style consists of

- **component characteristics**, which describe properties of the involved component processes;
- a **communication pattern**, which characterizes the top-level behavior of the system by a LOTOS pattern;
- **constraints**, which, when fulfilled, guarantee that an architectural description conforms to the style.

Such representations make style characteristics explicit and can serve as a guideline for designers. In the following, we present characterizations of three different architectural styles.

3 Repository Style

Garlan and Shaw [GS93] describe the repository style as follows:

> " In a repository style there are two distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store."

In our modeling, we suppose that the central data structure – the *shared memory* – contains data accessible via indices selecting parts of the stored data.

Component Characteristics

We consider three kinds of components operating on the shared memory: components that only read (part of) the memory, components that only change the memory, and components that do both. There is no interaction between components: they behave independently and communicate only with the repository and the environment.

The three kinds of components are illustrated in Fig. 1. The system interface is represented by black squares. If a component wants to change the shared memory, it sends the message `WR` (write request). This causes the shared memory to set a lock. Only then can the new value be passed, using the gate `W` (write). If a component wants to read the shared memory, it sends the message `RR` (read request). If no lock is set the value is passed via the gate `R` (read). It may happen

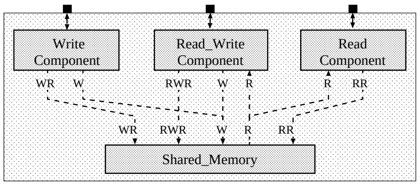

Fig. 1. General view of repository style architecture

that a value to be written into the shared memory 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 memory during a transaction. The process implementing the shared memory is defined as follows:

```
process Shared_Memory [RR, R, WR, W, RWR]
  (sm: shared_memory, is_locked: BOOL, for_whom: id): noexit :=
   [is_locked = false ]
  -> ( RR ? who: id; R ? who: id ? j : index ; R ! who ! get(sm, j);
       Shared_Memory [RR, R, WR, W, RWR] (sm, false, for_nobody)
    [] WR ? who: id;
       Shared_Memory [RR, R, WR, W, RWR] (sm, true, who)
    [] RWR ? who: id;
       Shared_Memory [RR, R, WR, W, RWR] (sm, true, who) )
[] [is_locked = true ]
  -> ( W ? who: id ? j : index ? nv: value [who=for_whom] ;
       Shared_Memory [RR, R, WR, W, RWR](store(sm,j,nv),false,for_nobody)
    [] R ? who: id ? j : index ; R ! who ! get(sm, j);
       W ? who: id ? nv: value [who=for_whom];
       Shared_Memory [RR, R, WR, W, RWR](store(sm,j,nv),false,for_nobody))
endproc
```

The process `Shared_Memory` has the gates `RR, R, WR, W, RWR` and the parameters `sm` representing the memory, `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 and an index are read via the gate `W`, or the part of the repository stored under index `j` 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]`. The new value of the shared memory becomes the new parameter of the process, and the lock is reset[1]. The constant `for_nobody` indicates that access to the shared memory is not reserved for a particular process.

The process `Shared_Memory` is the same for all instantiations of the repository architecture, except for the type of information to be stored. This type

[1] To keep our presentation concise, we do not permit parallel write or read/write actions on different parts of the shared memory, i.e. on different indices. The definition of such an optimization is straightforward.

shared_memory has to be defined algebraically. We need an initial value init, a function store changing the shared memory, and a function get reading it. The types id, index and value of the values that can be stored under an index are also defined algebraically.

Each repository architecture consists of a process Shared_Memory as defined above and an arbitrary number of independent components. Each of these is either a *read process*, a *write process* or a *read/write process*.

A read process does not use the gates WR, W, RWR and contains an arbitrary (positive) number of read behaviors but neither write nor read/write behaviors. A read behavior is defined by the pattern

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

where *me* is the identification of the process and *index* is the index to be read.

A write process does not use the gates RR, R, RWR and contains an arbitrary (positive) number of write behaviors but neither read nor read/write behaviors. A write behavior is defined by the pattern

```
WR ! me ;
W  ! me ! index ! v
```

where *v* is the new value to be stored under index *index*.

A read/write process may use three behavioral patterns. It contains at least one read/write behavior or read as well as write behaviors. A read/write behavior is defined by the pattern

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

followed by writing access to the shared memory in all subsequent branches[2] of the process according to the pattern

```
W  ! me ! index ! nv
```

for the same index *index* and a new value *nv*.

Communication Pattern

The communication between the shared memory and the independent components is expressed by the following pattern, where for better readability we use "..." instead of an inductive definition:

```
hide   RR, R, WR, W, RWR in
 Shared_Memory [RR, R, WR, W, RWR](init of shared_memory,false,for_nobody)
      |[ RR, R, WR, W, RWR ]|
 (          Component_1[gate_list_1]
      |||   ...
      |||   Component_n [gate_list_n] )
```

[2] This condition can be decided by a predicate defined inductively over the syntax of the behavior expression following the first part of the pattern.

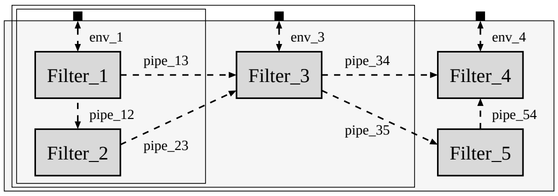

Fig. 2. A pipe/filter architecture

All components behave independently of each other (the operator `|||` involves no communication at all). For every ***Component_i***, its ***gate_list_i*** must contain the gates `RR` and `R` if it is a read process and `WR` and `W` if it is a write process. A read/write process may contain `RR, R` as well as `WR, W`, or `RWR`, `R` and `W`. The repository and the independent components must synchronize on these gates, as expressed by the synchronization list `|[RR,R,WR,W,RWR]|`. The `hide` clause hides communications via the gates `RR,R,WR,W,RWR` from the environment.

Constraints
Constraints are expressed in terms of the two parts of an architectural description, ***behav_expr*** and ***local_def_list***, see Section 2. For the repository style, we have the constraints that the ***behav_expr*** must conform to the communication pattern given above, and that each process occurring in ***behav_expr***, except `Shared_Memory`, must be a read, a write or a read/write process as defined above.

4 Pipe/Filter Style

The characteristics of pipe/filter style are the following [GS93]:

> "In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, [...] Components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence connectors are termed "pipes". [...] filters must be independent entities: in particular, they should not share state with other filters. "

Garlan et al. [GKMM96] additionally state the topological constraint that pipes are directional and that at most one pipe can be connected to a given "port" of a filter. Figure 2 shows an example of a pipe/filter architecture. A filter (in this case `Filter_3`) may have several incoming and several outgoing pipes. Cycles are also allowed, see [GS93]. In the LOTOS characterization of this style, a pipe between two filters is a synchronous communication via some gate.

Component Characteristics
A filter is modeled by a process that takes its inputs from the incoming pipes, transforms them according to its task, and delivers the results via the outgoing pipes. Communication with the environment is also possible.

Hence, a component of this style is not characterized by some specific behavior but by its gates. These are divided into the lists ***in_pipe_list***, ***out_pipe_list*** and ***env_gate_list***. A filter process does not write on gates of its ***in_pipe_list*** and does not read from gates of its ***out_pipe_list***.

Communication Pattern
Two filters communicate via their common pipes. For example, the filters `Filter_1` and `Filter_2` in the smallest box of the architecture shown in Fig. 2 exhibit the communication behavior

```
Filter_1[env_1,pipe_12,pipe_13] |[pipe_12]| Filter_2[pipe_12,pipe_23]
```

When adding the third filter `Filter_3` synchronizing with the previous system via the pipes `pipe_13` and `pipe_23`, the following behavior is obtained:

```
(                       Filter_1 [env_1, pipe_12, pipe_13]
  |[pipe_12]|           Filter_2 [pipe_12, pipe_23]  )
  |[pipe_13, pipe_23]| Filter_3 [env_3, pipe_13, pipe_23, pipe_34, pipe_35]
```

Hence, the general communication pattern of the pipe/filter has the form

```
  hide pipe_list_1, pipe_list_2, ... pipe_list_n-1 in
     (...((Filter_1[gate_list_1] |[pipe_list_1]| Filter_2[gate_list_2])
                                 |[pipe_list_2]| Filter_3[gate_list_3])
           ...
                                 |[pipe_list_n-1]| Filter_n[gate_list_n])
```

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

- All synchronization lists ***pipe_list_1***, ..., ***pipe_list_n-1*** occurring in ***behav_expr*** are disjoint, i.e., a pipe connects only two filters.
- Each gate occurring in some synchronization list of ***behav_expr*** occurs exactly twice in the gates of the processes ***Filter_1***, ..., ***Filter_n*** defined in ***local_def_list***, i.e., a pipe cannot be re-used as an external gate.
- Each of the processes ***Filter_1***, ..., ***Filter_n*** that occur in ***behav_expr*** must conform to the characterization given above. The gates of a process representing pipes are exactly the ones that occur in some synchronization list. The direction of the pipe can be determined from the process definition.

Note that, in our definition, pipes and filters have no buffers like in [AAG93], because – according to the synchronous communication of LOTOS – no data can be lost. The buffered version – which we consider to be closer to an implementation – could also be expressed in LOTOS.

5 Event-Action Style

According to Krishnamurthy and Rosenblum [KR95],

> "An event-action system is a software system in which events occurring in the environment of the system trigger actions in response to the events. The triggered actions may generate other events, which trigger actions, and so on."

Garlan and Shaw [GS93] mention that " The main invariant in this style is that announcers of events do not know which components will be affected by those events."

Component Characteristics
An event-action architecture consists of components that react to events. When an event has happened, actions are carried out and other events may be sent. An event manager is responsible for distributing all events that have occurred to all components that have to react to that event. Figure 8 shows an example of an event architecture. The event manager has the following form[3]:

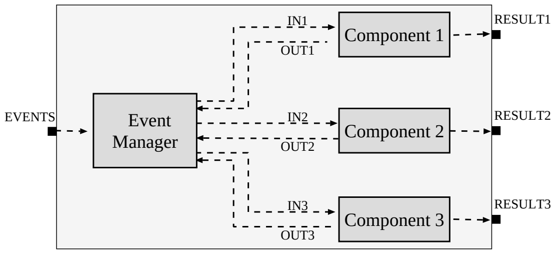

Fig. 3. An event-action architecture

```
process Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n] : func :=
      EVENTS ? e: event; exit(e)
   [] OUT_1  ? e: event; exit(e)
   [] ...
   [] OUT_n  ? e: event; exit(e)
   >> accept  e: event in
      [p_1(e)] -> IN_1,1 ! e ; ... IN_1,n1 ! e ;
                   Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
   [] ...
   [] [p_k(e)] -> IN_k,1 ! e ; ... IN_k,nk ! e ;
                   Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
endproc
```

This definition consists of two processes, separated by `>>`. The `accept` clause means that an event *e* is passed from the first process (via the `exit` clauses) to the second one. In the first process, the event manager reads incoming events, either from the environment via the gate *EVENTS* or from some other component via some gate *OUT_i*. It then decides how to distribute the events, according to the predicates *p_j*. The event manager may have functionality `exit` or `noexit`. The data type *event* must be defined algebraically. It can be structured to allow the handling of complex events.

Each event-action architecture consists of a process `Event_Manager` as described above and an arbitrary number of independent components. Each such component *Component_i* has a gate *IN_i* and contains an action

IN_i ? e: event

If the component generates events, it has a gate *OUT_i*, which is used to send events to the event manager. In this case, the process behavior contains actions of the form:

OUT_i ! e

The process does not write on *IN_i* and does not read from *OUT_i*.

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

Communication Pattern

The communication between the event manager and the independent components takes place according to the pattern

```
hide  IN_1, OUT_1, ... IN_n, OUT_n in
    Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
       |[IN_1, OUT_1, ... IN_n, OUT_n]|
  (          Component_1[IN_1, OUT_1, env_gate_list_1]
       |||   ...
       |||   Component_n [IN_n, OUT_n, env_gate_list_n] )
```

Constraints

The *behav_expr* and *local_def_list* making up the architectural description of an event-action system must satisfy the following constraints:

- *behav_expr* must conform to the communication pattern given above.
- Each of the processes that occurs in *behav_expr*, except `Event_Manager`, must conform to the description given in the component characterization.

6 Example

We illustrate our approach by specifying a robot. This robot can make the movements shown in Fig. 4: it can advance by moving its right or its left leg; it can stand still; and it can smile or not. In the following, we develop three alternative specifications, one for each style presented above. These three specifications use the same robot definition.

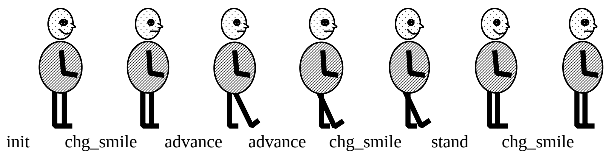

Fig. 4. The movements of the robot

The robot can be modeled as an automaton with three states: `standing`, `left_up` and `right_up` as shown in Fig. 5. To each state a boolean value is associated indicating whether the robot is smiling or not. The initial state is standing and smiling. The robot is defined by an abstract data type `robot` where the states are defined as constants and the movements as transitions from one state to another, except for smiling which is defined by a boolean value: true for smiling. For each state a predicate is defined deciding if the robot is in this state.

The movements are defined by the type `mvt` with three constants `m_stand`, `m_advance` and `m_chg_smile`. The robot will be asked to execute several movements collected in a list. This list is defined by an abstract data type `m_list` with a constant `empty`, a function `add` adding an element to the end of the list, a function `rm_first` removing the first element of a list, a function `first` selecting the first element of a list, and a predicate `is_empty`. A constant `init_list` is used to define the list of movements initially given to the robot.

We have the same interface for all architectures. The initial state of the robot and the movements to be performed are read via a gate `START`. A data

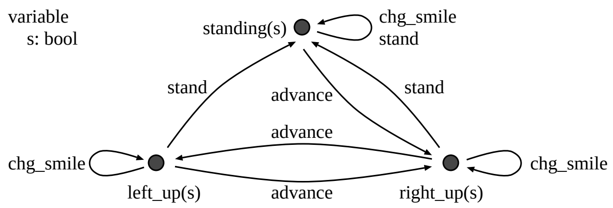

Fig. 5. The robot automaton

type `value` is defined as the Cartesian product of the types `robot` and `m_list`. Its constructor function is `make`, and its selector functions are `the_robot` and `the_list`. Via a gate `OUTPUT`, the current state of the robot is made visible to the environment. The top-level behavior

```
    START  !make(init of robot,init_list); exit  |[START]|  (behav_expr)
```

is the same for all three architectures. They are only distinguished by different definitions of *behav_expr* and the associated *local_def_list*.

6.1 The robot specification using the repository style

Our first robot design follows the repository style. The shared memory is to hold the current state of the robot and the list of movements to be executed, i.e. items of type `value`. We need only one index `index1`. The initial state and the initial list are written into the shared memory by a write process `Init_sm`.

```
  process  Init_sm [START,  W, WR] : exit :=
      START ? vv: value;
      WR ! id_Init_sm;  W  ! id_Init_sm ! index1 ! vv;  exit
  endproc
```

Furthermore, we need three components `Stand`, `Chg_Smile` and `Advance` to execute the corresponding movements, as illustrated in Fig. 6.

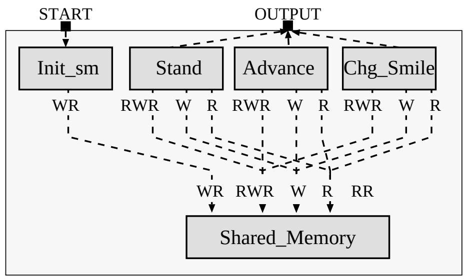

Fig. 6. The repository architecture

These components try in parallel to access the shared memory to execute the movement they are responsible for. They all are read/write processes. Each of them first reads the list of movements, denoted `ml`. If the first movement is the one it is responsible for, it is executed, the robot state changed (variable `roro`) and the rest of the movement list is written back into the shared memory. If the movement cannot be executed by the component that has been granted access, it writes back the unchanged state to unlock the shared memory.

According to our characterization, the overall behavior of the repository robot specification is

```
hide   RR, R, WR, W, RWR in
Shared_Memory [RR, R, WR, W, RWR](init of shared_memory,false,for_nobody)
          |[ RR, R, WR, W, RWR ]|
      (        Init_sm [START, W, WR]
          |||  Stand [OUTPUT, R, W, RWR]
          |||  Chg_Smile [OUTPUT, R, W, RWR]
          |||  Advance [OUTPUT, R, W, RWR]  )
```

Of the processes implementing the movements, we only present `Advance`. The others are defined analogously.

```
process  Advance [OUTPUT, R, W, RWR] : exit :=
      RWR ! id_Advance; R ! id_Advance ! index1 ;
      R ? for_whom: id ? v: value [for_whom=id_Advance];
     (let   ml: m_list = the_list(v), roro: robot = the_robot(v) in
          [is_empty(ml)= true ] -> W ! id_Advance ! v ; exit
      []  [is_empty(ml)= false] ->
       (  [first(ml) equal m_advance = true ]
               -> OUTPUT ! advance(roro) ;
                  W  ! id_advance ! make(advance(roro), rm_first(ml)) ;
                  Advance [OUTPUT, R, W, RWR]
      []  [first(ml) equal m_advance = false]
               -> W ! id_Advance ! v ;
                  Advance [OUTPUT, R, W, RWR]   ))
endproc
```

This architecture has the disadvantage that the system implementation must guarantee that each component is given the chance to access the shared memory. Otherwise, an infinite number of unsuccessful accesses is possible.

6.2 The robot specification using the pipe/filter style

In the pipe/filter modeling, we can make sure that each component is given the possibility to execute its movement if required. We have a line of filters, see Fig. 7, where each filter inspects the movement list. If it can execute the movement, it does so and hands the new robot state and the new movement list to the next filter. Otherwise, it passes on the unchanged data. Again, we need an initializing component, called here `Init_pf`.

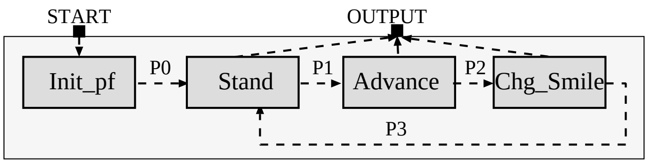

Fig. 7. The pipe/filter architecture

```
process  Init_pf [START, P0] : exit :=
      START ? vv: value; P0 ! vv ; exit
endproc
```

According to the style characterization, the overall behavior of the process is

```
  hide P0, P1, P2, P3 in
      (                     Init_pf [START, P0]
       |[ P0 ]|             Stand [P0, P1, P3, OUTPUT]
       |[ P1, P3 ]|         Advance [P1, P2, OUTPUT]
       |[P2]|               Chg_Smile [P2, P3, OUTPUT]   )
```

The `Advance` filter is defined as follows.

```
process Advance [P1, P2, OUTPUT] : exit :=
      P1 ? v: value;
      (let ml: m_list = the_list(v), roro: robot = the_robot(v)
       in [is_empty(ml)= true ] -> (exit)
       [] [is_empty(ml)= false] ->
              (  [first(ml) equal m_advance = true ]
                    ->   OUTPUT ! advance(roro) ;
                         P2 ! make(advance(roro), rm_first(ml)) ;
                         Advance [P1, P2, OUTPUT]
               [] [first(ml) equal m_advance = false]
                    ->   P2 ! v ;
                         Advance [P1, P2, OUTPUT]   ))
endproc
```

This solution is better than the repository architecture because it always terminates. It is not ideal, however, because each component must inspect the data, even if it cannot process them.

6.3 The robot specification using the event-action style

The event-action architecture, see Fig. 8, does not have the disadvantages of the previous architectures. The event manager inspects the movement list and passes on the data only to the component that can process them. Events are items of type `value`. The initial state of the robot and the movement list are given to the event manager. An initialization component is not required. The event manager is defined as follows.

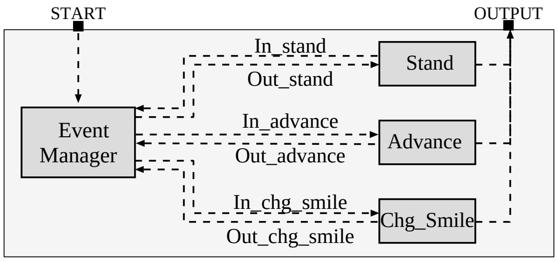

Fig. 8. The event-action architecture

```
process Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                       Out_chg_smile, In_advance, Out_advance]: exit :=
    START        ? v: value; exit(v)
 [] Out_stand     ? v: value; exit(v)
 [] Out_advance   ? v: value; exit(v)
 [] Out_chg_smile ? v: value; exit(v)
 >> accept v: value in
```

```
   (let ml: m_list = the_list(v), roro: robot = the_robot(v) in
       [is_empty(ml)= true ] -> (exit)
    []([is_empty(ml)= false] ->
       (  [first(ml) = m_stand]
             -> In_stand ! v ;
                Event_Manager [START, In_stand, Out_stand,
                   In_chg_smile, Out_chg_smile, In_advance, Out_advance]
        [] [first(ml) = m_advance]
             -> In_advance ! v ;
                Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                   Out_chg_smile, In_advance, Out_advance]
        [] [first(ml) = m_chg_smile]
             -> In_chg_smile ! v ;
                Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                   Out_chg_smile, In_advance, Out_advance])))
endproc
```

In accordance with the event-action style, we have the following overall behavior:

```
hide In_stand, Out_stand, In_chg_smile,
     Out_chg_smile, In_advance, Out_advance in
  Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                      Out_chg_smile, In_advance, Out_advance]
    |[In_stand, Out_stand, In_chg_smile,
      Out_chg_smile, In_advance, Out_advance]|
   (       Stand [OUTPUT, In_stand, Out_stand]
       ||| Advance [OUTPUT, In_advance, Out_advance]
       ||| Chg_Smile [OUTPUT, In_chg_smile, Out_chg_smile] )
```

Note that the components executing the movements are much simpler now.

```
process  Advance [OUTPUT, In_advance, Out_advance] : noexit :=
      In_advance ? v: value;
    ( let ml: m_list = the_list(v), roro: robot = the_robot(v)
      in OUTPUT ! advance(roro) ;
         Out_advance ! make(advance(roro), rm_first(ml));
         Advance [OUTPUT, In_advance, Out_advance] )
endproc
```

6.4 Comparing the three specifications with Aldebaran

Under the assumption of fairness for the repository solution, all the above specifications exhibit the same behavior to the environment. The tool CADP (Caesar/Aldebaran Distribution Package) [FGM+92] generates the same automaton minimized with respect to safety equivalence [Fer89] (i.e. internal transitions are not considered) for all the three architectures, where we use the movement list shown in Fig. 4. Stepwise execution of the three alternative architectures is also possible. This shows that existing LOTOS tools can help to animate and compare architectural descriptions, thus providing valuable support for their validation.

7 Discussion

Two of the style characterizations given in this paper, repository and event-action, contain a distinguished component (`Shared_Memory` and `Event_Manager`, respectively). This results in a relatively detailed characterization of the other components of the architecture because one can state requirements concerning the communication of the other components with the distinguished one. Further constraints are not necessary. In contrast, the pipe/filter style 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.

Formal descriptions of architectural styles and concrete architectural designs are important because only architectural descriptions with a formal semantics make it possible to precisely answer the questions stated by Clements [Cle96]: What are the components? How do they behave? What do the connections mean?

Our work shows that LOTOS is a language suitable to express individual architectures and that LOTOS patterns in combination with constraints are suitable to characterize architectural styles. Our style characterizations do not only provide a semantical foundation of architectural styles. Their schematic nature also makes it possible to use them as templates for the development of concrete architectures. The formal nature of the architectural descriptions and the availability of tools makes it possible to formally analyze and to animate them. In addition, our approach allows for hierarchical composition of architectural descriptions and definition of substyles by adding further constraints or adding further detail to the patterns.

We are not the first to formally characterize architectural styles or to use a process algebra to specify the behavioral aspects of software architectures. Abowd, Allen and Garlan [AAG93] use the specification language Z to formally define architectural styles. Concrete designs, however, are described in a different language. Thus, there is no direct way from a style definition to an instance of the style.

Allan and Garlan [AG94] use CSP to formalize architectural connection. In their approach, *connectors* are defined as processes. In contrast to our work where *components* are modeled as processes, this yields several de-centralized behaviors in one architectural description instead of one central behavioral description characterizing the whole system, as proposed in this work. Moriconi and Qian [MQ94] use CSP to show that an architectural description is a correct refinement of another. Both of these approaches are not concerned with architectural styles but with architectural descriptions in general.

The work presented here forms the basis for future work in several directions. First, a notion of architecture refinement will be defined, based on the notion of behavioral equivalence in LOTOS. Second, concepts for the machine-supported development of architectures as instances of styles will be developed. This can be done in such a way that (i) the developed architectures can be guaranteed to conform to the chosen style and (ii) dead-ends are avoided as far as possible.

Two development frameworks, designed by the authors, are good candidates for accommodating architecture development. The first is a knowledge representation mechanism called *strategies* [HSZ95]. They form a generic framework in which development knowledge for various software development activities can be expressed. This framework can be instantiated to support the development of LOTOS specifications representing architectural designs. The resulting design can be guaranteed to conform with the chosen style because strategies guarantee semantic properties of the developed product.

The second framework to model developments [SL93,Lév95] aims at providing specifiers with active tools to support them during the development process. It is language-independent and therefore can be used with existing specification languages. The resulting specifications can be verified and refined using existing tools. In this framework, developments are formalized as a stepwise application of development operators.

Experimenting with different models for machine support will help to find appropriate ways to support architectural design processes.

Acknowledgment. Thanks to Thomas Santen, Martin Simons and Jeanine Souquières for their comments on this work.

References

[AAG93] G. Abowd, R. Allan, and D. Garlan. Using style to understand descriptions of software architecture. *Proc. ACM SIGSOFT'93*, Dec. 1993.

[AG94] R. Allan and D. Garlan. Formalizing architectural connection. In *Proc. 16th Int. Conf. on Software Engineering.* ACM Press, 1994.

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

[Cle96] P. Clements. A survey of architecture description languages. In *Proc. of the 8th IWSSD*, pages 16–25, March 1996. IEEE.

[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. Rodriguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proc. of the 14th ICSE*, May 1992. ACM.

[GKMM96] D. Garlan, A. Kompanek, R. Melton, and R. Monroe. Architectural Style: An Object-Oriented Approach. In *Submitted for publication*, February 1996.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company*, 1, 1993.

[HSZ95] M. Heisel, T. Santen, and D. Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer, P. Botella, eds, *Proc. 5-th ESEC*, LNCS 989, pages 272–293, 1995.

[KR95] B. Krishnamurthy and D. Rosenblum. Yeast: a general purpose event-action system. *IEEE Trans. Software Eng.*, 21(10):845–857, Oct. 1995.

[Lév95] N. Lévy. Improving PROPLANE: a specifications development framework. In *Proc. Second IFAC Int. Workshop on Safety and Reliability in Emerging Control Technologies*, pages 229–240, Nov. 1995.

[LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans. Software Eng.*, 21(4):336–355, April 1995.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In David Wile, editor, *Proc. of the second ACM SIGSOFT Symp.*, pages 164–174. ACM Press, 1994.

[SL93] J. Souquières and N. Lévy. Description of Specification Developments. In *Proc. IEEE Int. Symp. on Requirements Engineering*, Jan. 1993.