Compositional Specification of ODP Binding Objects

Arnaud Février, Elie Najm,
{fevrier,najm}@res.enst.fr
ENST Paris
Département Réseaux
46, Rue Barrault
75013 - Paris
FRANCE

Guy Leduc, Luc Léonard
{leduc,leonard}@montefiore.ulg.ac.be
Université de Liège
Systèmes et Automatique
Institut d'Électricité Montefiore, B 28,
B-4000 LIÈGE 1
BELGIUM

Abstract

A building blocks approach for the formal specification of binding objects in the ODP computational Model is presented. The formal notation that is used is based on LOTOS extended with two features - real time and gate passing. These features are among the extensions that are currently studied in the ISO standardisation Formal Description Techniques group. We apply our building blocks approach to the specification of a multicast, multimedia binding object.

1 INTRODUCTION

The ODP reference model (ISO/ODP, Stefani 1990) provides for a multiple-viewpoint specification of distributed applications and systems. Five viewpoints have been defined within ODP and are considered to encompass the different areas of concerns that need to be covered when one develops a system or application. These five viewpoints are: enterprise, information, computation, engineering and technology. For a given system or application, the enterprise viewpoint defines its requirements at a strategic level; the information viewpoint describes the information needed to represent it; the computational viewpoint provides an abstract implementation of it; the engineering viewpoint describes how the computational description is supported in terms of generic system components and communication protocols; and the technological viewpoint maps the generic engineering components onto existing pieces of hardware and software.

In the present paper, we will concentrate on the computational viewpoint which is of particular relevance to application programmers. The computational viewpoint is also interesting for systems designers as they are concerned with mapping computational descriptions onto generic execution components in the engineering model.

The Computational Model is the (abstract) language used to describe applications in the computational viewpoint: in the Computational Model, an application is represented as a dynamic configuration of interacting objects. Objects are the key concept in the Computational Model. A Computational object has a state that may be accessed externally

only through interactions at its interfaces (we will hereafter, when there is no confusion, refer to computational objects simply as objects). An Object may possess (possibly many) interfaces which may be dynamically created and deleted. Interfaces are names of locations of interactions between objects. Objects may change dynamically their communicating partners by exchanging interface names. Objects may also create and delete other objects.

There are two kinds of objects in the Computational Model, namely, basic objects and binding objects. Binding objects are used to convey interactions between interfaces (of basic objects or other binding objects). In fact, in the Computational Model, the programmer may choose one of two ways for describing the interactions between interfaces: (i) either explicitly through a binding object, or (ii) implicitly without exhibiting a binding object. When specifying an explicit binding object, the programmer may incorporate the QoS requirements (order, timeliness, throughput, ...) on the transport of the interactions supported by that binding object. In contrast, in an implicit binding between two interfaces, no specific requirements are made on the transport of interactions: interfaces interact by message passing with no explicit ordering or delay required on the transport of these messages.

There are three kinds of interfaces in computational objects: signal, operational and stream. Signal interfaces are the most primitive: operational and stream interfaces can be modeled as special types of signal interfaces. A signal is an operation name and a vector of values (references to interfaces). A signal interface is an interface that emits and receives signals. An operational interface is an interface that can receive invocations and possibly react with result messages. Invocations and result messages are signals. An operational interface has a type which is, roughly, defined to be the type of the operations it can handle (where the type of an operation includes the types of its return messages). A subtyping system allows for the safe substitution of an interface of a given type by another interface having a subtype of this type. A stream interface is an abstraction of a signal interface: the type of a stream interface is simply a name and a role (sender or receiver).

The development of ODP is a new challenge for formal techniques (Stefani 1990, Vissers 90). Since July 94, the Formal Description Techniques (FDTs) group within ISO has become part of the Open Distributed Processing (ODP) standardization committee (SC21/WG7). Thus, supporting the formal design of open distributed systems is a new objective for this group. Indeed, the ISO-FDT team of experts is now working on the standardization of an extension of LOTOS - temporarily called E-LOTOS (WD95), which is targeted, among other things, at providing support to the design of ODP systems. This group has established a list of desirable features together with a list of requirements that E-LOTOS should aim to fulfill. Aspects related to real-time, constructive data representations and modularity are being actively studied. Dynamic reconfiguration of communication structures is also being tackled.

The present paper is an exercise in the specification of a binding object using a formal description technique. Binding objects are important for both application and system designers and developers and they can be used in many different ways. For instance, application designers may specify their transport requirements and let system designers develop new networks and protocols that match these requirements. On the other hand, application programmers may use the abstraction provided by existing binding objects to develop and analyze their applications. A specification of a binding object should cover the

functional and QoS requirements. Functional requirements include: connection establishment, dynamic reconfiguration, orderly transport of information, etc. QoS requirements involve: connection establishment delay, jitter, throughput, error rate, inter and intra flow synchronization, etc. Thus, the specification language should be expressive and able to address real-time constraints and to capture dynamic reconfiguration of communicating components.

We use for our specification exercise the LOTOS language (ISO88, Bolognesi 1989) extended with two features that are currently under study in the ISO/FDT group: real time and gate (i.e. reference) passing à la p-calculus (Milner 1992). We call this language MT-LOTOS. We show how the MT-LOTOS indeed allows for a modular construction of our binding object. We introduce first a collection of generic building blocks specified in MT-LOTOS. Each of these blocks has a self contained meaning and can be composed with other building blocks. The genericity and reusability of these blocks is illustrated in the construction of a multimedia, multicast binding object.

The remainder of the document is structured as follows. Section two is a short presentation of MT-LOTOS. In section three we introduce the collection of building blocks specified in MT-LOTOS. Section four is devoted to the specification of the binding object example, which is first presented informally. In section five we conclude.

2 A BRIEF PRESENTATION OF MT-LOTOS

As said earlier, MT-LOTOS is a combination of two extensions to LOTOS. These extensions are formally specified and discussed in (Léonard 1994, Léonard 1995) and (Najm 1995a, Najm 1995b). In this paper we give a short informal presentation, leaving aside the data typing aspects. In order to make our presentation clear and self contained, we present MT-LOTOS as a language on its own, without discussing its differences with LOTOS. It is however important to note that MT-LOTOS is an upward compatible extension of LOTOS.

The primitive concepts of MT-LOTOS are: actions, processes and agents. These can be composed as follows: (i) actions can be composed to form processes, (ii) processes can be composed to form processes and/or agents; (iii) agents can be composed to form agents. We briefly introduce each of these concepts.

2.1 Actions

Actions can be internal, represented by the symbol `i`, or external. An external action is a gate and an ordered list of offers: `goff1...offn`; where an offer is one of four possible forms: (i) presenting a value: `!E` (resulting from the evaluation of expression E), (ii) accepting a value of some type `?x:t` (and storing it in variable x), (iii) presenting a gate name: `!g`, (iv) accepting a gate name: `?h:gid` (which is stored in h. Note the special type for gates, *gid*). A typical feature of MT-LOTOS is that actions may be conditioned by a predicate and/or a time constraint, and may have a side effect of creating new agents. The syntax of actions is captured by the following table:

a ::= i{time-const} new C | g off1...offn {time-const}[pred] new C
off ::= !E | ?x:t | !g | ?h:gid
time-const ::= t in t1..t2 | t | t1..t2

The general form of time-const is `t in t1..t2` where `t` is a declared variable that records the time elapsed between action offering and action occurrence. Two other forms are allowed, respectively when there is just a recording variable `t` without actual constraint, or a constraint without time recording. `Pred` is a Boolean expression, possibly referring to variables used in the offers `offi` of the action or declared in time-const. Finally, `C` is an agent that is created as a side-effect of the execution of the action (we will see how agents are constructed in the sequel). An unconstrained action `a`, is equivalent to: `a {0...inf} [true]`. Note that no predicate `Pred` can be associated with an internal action i.

2.2 Processes

Processes are obtained by composition of actions and/or other processes. Let us first introduce three simple process constructs, then we turn to more sophisticated ones: (i) `stop` is the simplest process, representing the do-nothing-while-letting-time-pass behaviour, (ii) if `B` is a process then: `c; B` is the process representing the behaviour: "perform action c and then enable (the actions) of process B", (iii) `wait t; B` represents the behaviour "let t time units pass and then enable (the actions) of B". Assuming `B, B1` and `B2` represent generic processes, the general form of MT-LOTOS processes is givem in the following table (using a BNF grammar rule).

B	::=	stop
	|	c ; B
	|	wait t; B
	|	B1[]B2
	|	B1 |[g1, ..., gn]| B2
	|	exit
	|	B1 >> B2
	|	B1 [> B2
	|	hide g1, ..., gn in B
	|	P [h1, ..., hk](E1, ..., Em)

A few words on each of the newly introduced constructs.

`B1[]B2` is a disjunctive choice between the actions of `B1` and the actions of `B2`. The choice is resolved by the execution of the first action. For instance, the behaviour of `(c1; B1) [] (c2; B2)` is: "perform c1 then enable `B1` (thus disabling `c2;B2`) or perform `c2` then enable `B2` (thus disabling `c1;B1`).

`B1[>B2` is the disabling of the behaviour of `B1` by the first action of `B2`. For instance, the behaviour of `(c1; B1) [> (c2; B2)` is: "perform `c1` then enable `B1[>(c2;B2)` or perform `c2` then enable `B2` (thus disabling `c1;B1`)".

`exit` is the process which performs one special action, the successful termination action, and then stops. This termination action is used to enable a new process as explained in the following construct.

`B1>>B2` is the enabling of the behaviour of `B2` by the last action of `B1`. For instance, the behaviour of `exit >> B2` is: "perform a (hidden) termination action and then enable `B2`"; and the behaviour of `(c; B1)>>B2` is: "perform action `c` then enable `B1>>B2`".

`B1|[g1, ..., gn]|B2` is: "run `B1` and `B2` in parallel enforcing the synchronisation on actions occurring at gates `g1, ..., gn` while letting the other actions free". In order for two actions to be synchronisable, they must have the same gate, two matchable list of offers (i.e., where the i^{th} offer of one list matches the i^{th}offer of the other list), and the predicates, if any, must evaluate to true. Offers are matchable as follows: (i) two values can be matched if they are of the same type and they are equal, (ii) a value and a variable can be matched if they have the same type; the matching, in this case, results in a transfer of the value to the variable, (iii) two variables of the same type are always matchable. Note that if `B1` and `B2` synchronise on two actions, then they are said to perform jointly a synchronised action, and this action can be further synchronised with yet a third process, like e.g., in the expression: `(B1|[g]|B2)|[g]|B3`. Note that `i` is the action that can never synchronise with any action.

`hide g1,...,gn in B` hides (transforms into the internal action, `i`) the actions occurring on gates `g1,..., gn`. Thus, in an expression`(hide g1,...,gn in B)|[g1]|B'`, the actions of `B` occurring on gate `g1` are internal and thus cannot synchronize with actions from `B'`. hide has also another function: it creates new gate names. For instance in the expression, `hide g in h!g ; B`, a new gate, `g`, is created and sent on gate `h`.

The specifier may define named processes by a set of equations of the form:
`Process P[g1,...,gk](x1:t1,...,xm:tn) := B Endproc` where `P` is the name of the process, `g1,...,gk` is a list of gate name parameters and `x1:t1,...,xn:tn` a list of value parameters (typed variables). Hence, the behaviour of `P[h1,...,hk](E1,...,Em)` is defined to be the same as `B` where each `gi` has been substituted with `hi` and each `xi` has been substituted with `Ei`.

2.3 Agents

The communicating architecture of processes is static and imposed by the parallel operators. Agents have dynamic communicating structures: agents may discover new agents and interact with them, agents may also forget about previously known agents. Agents are made from processes using the embedding operator <_>: if `B` is a process, then <B> is an agent. <B> is the simplest form of an agent. The function of the embedding operator, <_>, is to put a boundary around a process, thus allowing it to interact with other agents. Agents can be put in parallel with other agents using the operator |. For instance <B1> | <B2> is the parallel composition of agents <B1> and <B2>. In contrast with processes, interaction between agents is binary and the synchronisation gates are not given explicitly in the parallel operator. In <B1> | <B2>, <B1> and <B2> can perform actions freely (without synchronisation) and can also synchronise on matching actions. In this case, the resulting joint action is hidden. The behaviour of an agent can be restricted by disallowing actions occurring at a specified list of gates. For instance, if `C` is an agent, then `restrict g1,...,gn to C` is an agent which has a behaviour similar to `C` except that it does not perform any action occurring on gates `g1,...,gn`.

One last remark concerning the creation of agents. We have seen that new agents can be spawn as a side effect of some actions. The following is an example of this feature. Take the agent < `g!h new <B1> ; B2`>. This agent performs action `g!h` (offering gate `h` on gate `g`) which enables then the agent: <`B1`> | <`B2`>, i.e., the parallel composition of <`B2`> with the spawn agent <`B1`>.

Finally, one can define named agents in a way similar to that of named processes. The syntax of agents is given by the grammar rules:

C	::=	< B >
	|	restrict g1, ..., gn to C
	|	C1 | C2
	|	A [h1, ..., hk](E1, ..., Em)

3 A COLLECTION OF BUILDING BLOCKS

One of the most important specification styles of (MT-)LOTOS is the constraint oriented style. Thanks to this style, one can obtain specifications composed from generic modules where each module represents a constraint that acts upon a designated part of the system. The constraints can be of different forms, such as the order of actions on a given gate, the timeliness of actions, the structure of the data conveyed in the actions, etc.

We have identified a collection of generic components that are suitable for the specification of functional and QoS requirements of multimedia and multicast binding object. We present them below, together with their MT-LOTOS specification. In these specifications, `dt` represents a packet of a certain media (audio or video).

Medium: This component describes a point to point transmission medium between two points. Packets are received on gate `ist` (input stream), and are delivered on gate `ost`. Medium is very general. The only constraint it expresses is that no packet is lost. On the other hand, the transmission delay of each packet is totally unconstrained and the ordering of the packets is not preserved. After the reception of a packet on `ist`, `i{0..inf}` introduces a nondeterministic delay before the delivery on ost (output stream). In parallel a new occurrence of Medium handles the following packets.

```
PROCESS Medium [ist, ost]: NOEXIT :=
    ist ? dt:data; ( i{0..inf}; ost ! dt; STOP ||| Medium [ist, ost])
ENDPROC (* Medium *)
```

FIFO_Const: This component also considers gates `ist` where packets are received, and `ost` where these packets are delivered. It enforces that the packets be delivered in the same order as they are received. In process FIFO_Const, the ordering is handled with an appropriate data structure: `q`, that describes a FIFO queue. We will not enter here into the details of the datatypes definition. At any time, FIFO_Const can accept`(ist ?dt:data)` a new packet that is added to `q`, or deliver `(ost !first(q))` the first packet in `q`.

```
PROCESS FIFO_Const [ist, ost](q:fifo):NOEXIT :=
        ist ?dt:data; FIFO_Const [ist,ost](append(dt,q))
    []  [not(IsEmpty(q))] -> ost !first(q); FIFO_Const [ist,ost](rest(q))
ENDPROC (* FIFO_Const *)
```

Delay_Const: Here again, two gates are considered. Delay_Const enforces that at least a minimal delay `delmin` elapses between the receiving of a packet on `ist` and its delivery on `ost`.

```
PROCESS Delay_Const [ist, ost](delmin):NOEXIT :=
    ist ? dt:data ; (Wait delmin ; ost!dt; STOP  ||| Delay_Const [ist, ost](delmin) )
ENDPROC (* Delay_Const *)
```

Delay_Obs: Delay_Obs expresses a requirement on the service provided by a transmission medium. It verifies that the delay between the reception and the delivery never exceeds a maximal value. If the packet is not delivered before this maximal delay, an error message is sent on the management gate `m`. After the reception of a packet, Delay_Const proposes `ost!dt` during a time `delmax`. On the other hand, `m!error_delay!ost` is delayed by `delmax+epsilon`. In other words, `m!error_delay!ost` is enabled when the delivery cannot occur anymore.

```
PROCESS Delay_Obs [ist, ost, m](delmax):NOEXIT :=
    ist ? dt:data ; (  ( ost ! dt {0..delmax}; STOP
                         []
                         wait(delmax+epsilon); m!error_delay!ost ; STOP )
                       ||| Delay_Const [ist, ost, m](delmax) )
ENDPROC (* Delay_Obs *)
```

Jitter_Const: Jitter_Const has an effect similar to Delay_Const, but on just one gate. It enforces that at least a minimal delay `jmin` elapses between any two successive deliveries of packets at gate `ost`.

```
PROCESS Jitter-Const [ost](jmin):NOEXIT :=
    ost ? dt:data; Jitter-Const2 [ost](jmin)
where
    PROCESS Jitter-Const2 [ost](jmin):NOEXIT :=
        wait jmin ; ost ? dt:data; Jitter-Const2 [ost](jmin)
    ENDPROC (* Jitter-Const2 *)
ENDPROC (* Jitter-Const *)
```

Jitter_Obs: Jitter_Obs has an effect similar to Delay_Obs, but on just one gate. It verifies that the delays between successive deliveries of packets on gate `ost` do not exceed `jmax`. Like Delay_Const, it signals an error if this happens.

```
PROCESS Jitter-Obs [ost, m](jmax):NOEXIT :=
    ost ? dt:data; Jitter-Obs2 [ost, m](jmax)
where
    PROCESS Jitter-Obs2 [ost, m](jmax):NOEXIT :=
            ost ? dt:data {0..jmax}; Jitter-Obs [ost, m](jmax)
        []  wait (jmax+epsilon); m!error_jitter!ost ; STOP
    ENDPROC (* Jitter-Obs2 *)
ENDPROC (* Jitter-Obs *)
```

One_Ind_Flow: One_Ind_Flow gives a first example of the modularity allowed by

MT-LOTOS. It describes a flow that combines the effects of the previous components. So, this flow loses no packet and preserves their order; the transmission delay of each packet is undetermined, but it is at least of `delmin`, and it cannot exceed `delmax`, otherwise an error message is sent and the transmission is stopped; the delay between successive deliveries of packets (the jitter) is at least of `jmin` and at most of `jmax`, if this maximal value is exceeded, an error message is also sent and the transmission is stopped.

One_Ind_Flow is simply obtained by putting in parallel the various constraints (or processes) and by enforcing their synchronisation on the gates `ist` and `ost`. In this case, One_Ind_Flow integrates all the constraints, but any other combination of them would have been possible too (For example with no lower bound on the transmission delay or with no preservation of the order) resulting in a less constraining flow. Furthermore, One_Ind_Flow allows the handling of a disconnection through the management gate `m`: the occurrence of `m !Dreq!ost` interrupts the flow and the whole process turns into `stop`.

```
PROCESS One_Ind_Flow [ist, ost, m]
    (q:fifo, delmin, delmax, jmin, jmax):NOEXIT :=
    (    (    Medium [ist, ost]
              |[ist, ost]| FIFO_Const [ist, ost] (q)
              |[ist, ost]| Delay_Const [ist, ost] (delmin)
              |[ist, ost]| Delay_Obs [ist, ost, m] (delmax)
         )    |[ost]| ( Jitter_Const [ost](jmin)
                        |[ost]| Jitter_Obs [ost, m](jmax) )
    ) [> m !Dreq!ost ; STOP
ENDPROC (* One_Ind_Flow *)
```

Inter_Sync_Const: Until now, we have only presented constraints handling one flow. Inter_Sync_Const controls the synchronisation between the packets delivered by two flows. The complete synchronisation mechanism requires two brother instances of process Inter_Sync_Const: one per flow. Inter_Sync_Const controls the packets delivered on `ost` by its local flow and exchanges on gate `s` synchronisation information with the Inter_Sync_Const responsible for its brother flow. The way Inter_Sync_Const is combined with a flow is illustrated by the next component: One_Sync_Flow. The effect of Inter_Sync_Const is to ensure that the packets on its local flow are not delivered too late or too early with respect to the packets on the brother flow. The local flow (resp. the brother flow) may be ahead of the brother flow (resp. the local flow) of at most `my` (resp. `ym`) time units. If these constraints cannot be met, an error message is sent on gate `m` and the flow is interrupted. We will not enter here into more details about the synchronisation mechanism. The actual values for these parameters are given in the following section. The meaning of the parameters used in this process is the following:

- `ml` is the ideal time for my last packet
- `yl` is the ideal time for your last packet
- `me` is the time elapsed since my last packet
- `ye` is the time elapsed since your last packet
- `my` is the accepted advance of my stream over your stream
- `ym` is the accepted advance of your stream over my stream
- `mm` is the interval between two successive packets of my stream
- `yy` is the interval between two successive packets of your stream

- `ms` is the name of my stream : takes one of two values `a` or `v`
- `ys` is the name of your stream : takes one of two values `v` or `a`

```
PROCESS Inter_Sync_Const [ost, s, m]
        (ml, yl, me, ye, my, ym, mm, yy:time, ms, ys:streams):NOEXIT :=
    ost ? dt:data {t} [ my > ml+mm-(yl+ye+t) > 0-ym ] ;
            Notify_Other_Stream[ost, s, m]
                (ml+mm, yl, 0, ye+t, my, ym, mm, yy, ms, ys)
    []  s!ys; Inter_Sync_Const [ost, s, m]
            (ml, yl+yy, me+t, 0, my, ym, mm, yy, ms, ys)
    []  wait (ml+mm - (yl+ye)+ym+epsilon); m!inter_sync_error!ost ;
    STOP
WHERE
    PROCESS Notify_Other_Stream [ost, s, m]
        (ml, yl, me, ye, my, ym, mm, yy:time, ms, ys:streams): NOEXIT :=
        s!ms {0..0}; Inter_Sync_Const[ost, s, m]
            (ml, yl, me, ye, my, ym, mm, yy, ms, ys)
        [] s!ys; Notify_Other_Stream[ost, s, m]
            (ml, yl+yy, me+t, 0, my, ym, mm, yy, ms, ys)
    ENDPROC (* Notify_Other_Stream *)
ENDPROC (* Inter_Sync_Const *)
```

One_Sync_Flow: This component gives a new example of the modularity allowed by MT-LOTOS. One_Ind_Flow is already the composition of several features. One_Sync_Flow enhances it with Inter_Sync_Const, a synchronisation mechanism with another flow. Again, the addition of a constraint is simply obtained by putting both processes in parallel. Furthermore, this last interflow constraint may be removed if a request is made on gate `m`. The constraint is then replaced by the neutral process Sink.

```
PROCESS One_Sync_Flow [ist, ost, m, s]
    (q:fifo,delmin, delmax, jmin, jmax:time,
     ml, yl, me, ye, my, ym, mm, yy:time, ms, ys:streams): NOEXIT :=
(   One_Ind_Flow [ist, ost, m] (q, delmin, delmax,jmin, jmax)
    |[ost]|
    ( Inter_Sync_Const [ost, s, m] (ml, yl, me, ye, my, ym, mm, yy, ms, ys)
        [> m!dis_other_stream!ost ;Sink[ost] )
)[> m!Dreq!ost ; STOP
ENDPROC (* One_Sync_Flow *)
```

Sink: This component enforces no constraint on the actions occurring on a gate. It is specified by a process with a single gate: `st`. In process sink no predicate or time constraint restricts the acceptance of packets on `st`.

```
PROCESS Sink [st] : NOEXIT := st ? dt:data ; Sink [st] ENDPROC (* Sink *)
```

Multicast: This component is independent from the previous ones. It describes a multicasting mechanism. Considering one "input" or source gate (`src`) and a collection of "output" gates (described by the generic gate variable `cast`), Multicast conveys messages, without delay, between the "input" gate and every "output" gates. The command to create a new output gate `cast` and an associated connection from `src` to the new `cast` is received on gate `m`. Remark that Multicast receives the new gate name in variable `cast` of type `gid` (which is the special type used for gate names). Figure 1 is a graphical representation of the multicast process, in two situations: one active connection (left) and

two active connections (right). Note that processes are represented by rectangular boxes wheras agents will be represented by rounded boxes.

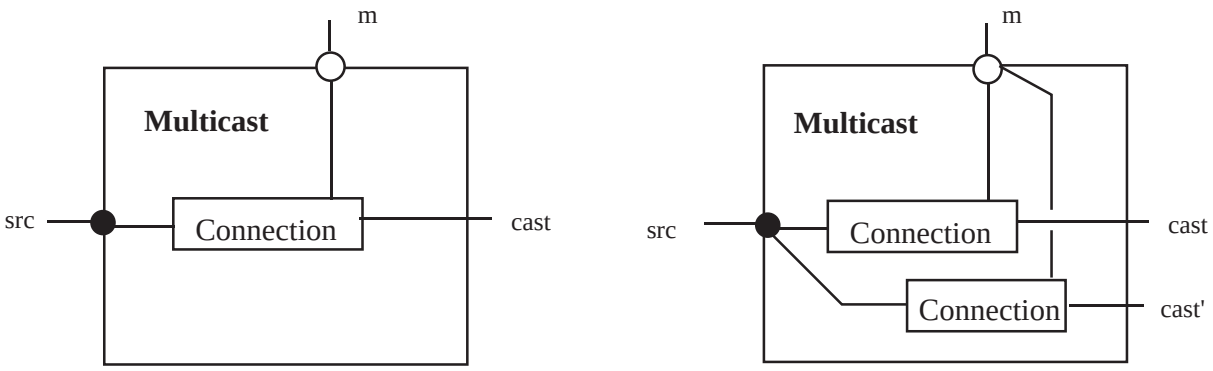

Figure 1 the muticast component, with one connection (left) and two connections (right)

```
PROCESS Multicast [src, m]:NOEXIT :=
    src? dt: data ; Multicast [src, m]
    []
    m ! Creq ? cast : gid;
        (    ( Connection [src, cast] [> m !Dreq ! cast ; Sink [src] )
             |[src]|
             Multicast [src, m]
        )
WHERE
    PROCESS Connection [src, cast]: NOEXIT :=
        src ? dt : data ; cast !dt {0}; Connection [src, cast]
    ENDPROC (* Connection *)
ENDPROC (* Multicast *)
```

4 A MULTICAST MULTIMEDIA BINDING OBJECT

We want to specify an ODP Binding Object that supports a video broadcast application. The binding object we aim at fulfils the following functions:

- it listens to a source emitting two synchronised flows, an audio and a video, and multicasts the two flows to a dynamically changing set of clients,
- at any time a client can request to join the audio or the video or both the audio and video streams by providing the reference of one (or two) receiving interface(s),
- at any time a client may request to leave the audio, or the video or both audio and video flows,
- it tries to enforce the intra and inter synchronisation of flows and notifies failures to do so.

The source flows have these characteristics, inspired from (Stefani 1992):

- there are 25 images per second, i.e., the video stream is constituted by packets delivered every 40 ms.
- the sound is sampled every 30 ms, i.e., a sound packet is delivered every 30 ms.

We suppose that the two source flows do not deviate from the above figures and that both flows are fully synchronized. The Binding object accept these flows and delivers them to any requiring customer. Since the binding object will encapsulate the behaviour of a concrete network, it will have to deal with usual networks problems (jitter, packet loss, end to end delay, ...). Nevertheless the customers expect a minimal QoS. The QoS is two fold:

- each flow must respect a QoS,
  - the sound may suffer no jitter,
  - the video allows a jitter of 5ms, i.e. consecutive images may be seperated by 35 to 45 ms.
- both must be *reasonably* synchronous. This is known as *lip synchronization*

The lip synchronization is considered correct if the sound is not too far (back or ahead) from the corresponding lip movement. The actual figures are:

- the sounds must not come more than 15 ms before the lip movement,
- the sounds must not come later than 150 ms after the lip movement.

We give the MT-LOTOS specification below. The above figures will be used in the specification as follows:

- `abv` = 15 ms (allowed advance of the audio stream on the video stream)
- `vba` = 150 ms (allowed advance of the video stream on the audio stream)
- `ar` = 30 ms (audio packets rate)
- `vr` = 40 ms (video packets rate)

Figure 2 represents the initial configuration of agents, i.e., when no clients have joined the casts. Note that MT-LOTOS agents are represented by rounded rectangles. Figure 3, at the end of the specification, represents one client connected to the (synchronised) audio and video flows.

SPECIFICATION `binding-object [srca, srcv, c]`
(* gate `c` is the controlling gate of the binding object *)
(* gates `srca` and `srcv` are the audio and video source gates respectively *)

TYPE SORTS streams
OPNS a, v -> streams
ENDTYPE
BEHAVIOUR
RESTRICT mma, mmv **TO**
< MGR[c, mma, mmv] > | < Multicast[srca, mma] > | < Multicast[srcv, mmv] >

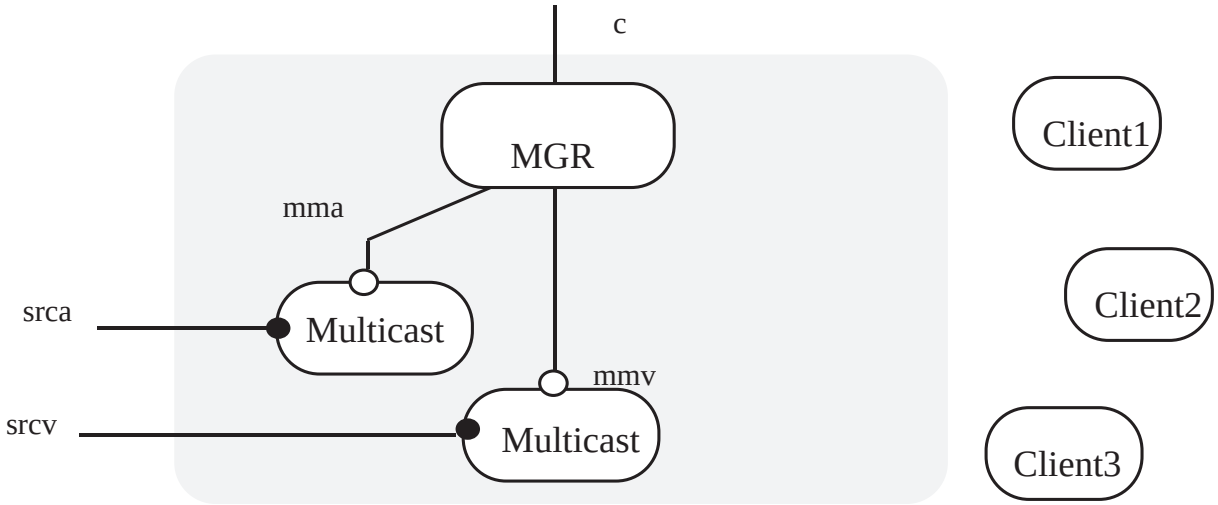

Figure 2 the binding object with no connected clients

At the initial state, the binding object configuration contains two Multicast objects, one for audio and one for video and a manager object, MGR. At this initial state, the Multicast objects only listen, each to its specific media source, and no destinations are active, i.e. no client is being serviced.

The manager of the binding object, MGR, is accessed by the clients at gate c, and manages the audio and video Multicast objects through gates mma and mmv respectively.

WHERE
PROCESS MGR [c, mma, mmv]:**NOEXIT** :=

`MGR` is a choice between three actions corresponding to three types of requests from clients: a request to join the audio Multicast, a request to join the video Multicast, or a request to join both the audio and video Multicast.

```
HIDE isa, m, mgt_client IN
    c !Creq-a ?r_client:gid ?osa:gid
        ?delmin:time ?delmax:time ?jmin:time ?jmax:time5
    NEW
    ( < One_Ind_Flow [isa, osa, m](empty, delmin, delmax,jmin, jmax) >
    | < Client_One_Flow_MGR [mgt_client, mma, m](isa, osa, r_client) >
    | < r_client !mgt_client ;STOP>    );
```

This is a request to join the audio Multicast. The request contains a return gate, `r_client`, of the requesting client and the gate that has to be bound to the audio stream, `osa`.

The MGR agent creates the gates `isa`, `m` and `mgt_client` and two objects, `One_Ind_Flow` and `Client_One_Flow_MGR`, and a return message <`r_client!mgt_client;STOP`>. Gate `isa` connects the Multicast object to `One_Ind_Flow` which conveys the audio stream to the client gate `osa`. `Client_One_Flow_MGR` manages through the access gate `mgt_client`

the requests from the client concerning this stream. `Client_One_Flow_MGR` operates on `One_Ind_Flow` through gate `m`. Message <`r_client!mgt_client; STOP`> notifies the client of the success of the binding and provides him with the interface name `mgt_client` that has been created for him to manage his connection.

```
mma !Creq !isa ; MGR [c, mma, mmv]
(* MGR conveys the request, on behalf of the client, to the
audio Multicast object, and provides the name of the input gate isa *)
[]Hide isv, m, mgt_client IN
    c    !Creq-v ?r_client:gid ?osv:gid
         ?delmin:time ?delmax:time ?jmin:time ?jmax:time
    NEW
         ( < One_Ind_Flow [isv, osv, m]
                  (empty, delmin,delmax,jmin, jmax) >
         | < Client_One_Flow_MGR [mgt_client, mmv, m]
                  (isv, osv, r_client) >
         | < r_client !mgt_client ;STOP> );
         mmv !Creq !isv ; MGR [c, mma, mmv]
```

This is the symmetric request for a video connection

```
[] Hide isa, isv, m, mgt_client IN
c !Creq-av ?r_client:gid ?osa:gid ?osv:gid
    ?delmin:time ?delmax:time ?jmin:time ?jmax:time
    ?abv:time ? vba:time ?ar:time ?vr:time
NEW
    ( RESTRICT s TO
         (    <    One_Sync_Flow [isa, osa, m, s]
                   (empty, delmin, delmax,jmin, jmax,
                   0, 0, 0, 0, abv, vba, ar, vr, a, v) >
              | < One_Sync_Flow [isv, osv, m, s]
                   (empty, delmin, delmax,jmin, jmax,
                   0, 0, 0, 0, vba, abv, vr, ar, v, a) >)
         | < Client_Two_Flows_MGR [mgt_client, mma, mmv, m]
              (isa,isv,osa,osv, r_client) >

         | < r_client !mgt_client ;STOP>
    ); ( mma !Creq !isa ; EXIT ||| mmv !Creq !isv ; EXIT )
         >> MGR [c, mma, mmv]
```

This third sub-expression of the choice represents the handling of a combined audio/video connection request. The client provides two gates to be bound, `osa` and `osv`, one for the audio and one for the video stream respectively. In this case, the objects that convey the streams are instantiated from the `One_Sync_Flow` template: the two streams have to be synchronized. In this instantiation, we have used the following constants:
`abv` : Maximum advance of the audio stream on the video stream
`vba` : Maximum advance of the video stream on the audio stream
`ar` : Interval between two audio data packets.
`vr` : Interval between two video data packets.

WHERE

```
PROCESS Client_One_Flow_MGR [mgt_client, mm, m]
    (ist, ost, r_client:gid) :noexit :=
    ( mgt_client ! Dreq ; (mm !Dreq !ist ; STOP
                          ||| m !Dreq !ost ; STOP )
        [] m ? er:error!ost ; mm !Dreq !ist NEW
            <r_client!er!ost; STOP> ; STOP )
ENDPROC (* Client_One_Flow_MGR *)
PROCESS Client_Two_Flows_MGR[mgt_client, mma, mmv, m]
    (isa,isv,osa,osv,r_client:gid) :noexit :=
    mgt_client ! Dreq !a ; m!dis_other_stream!osv ;
    (     mma !Dreq!isa; EXIT ||| m!Dreq !osa ; EXIT )
        >> Client_One_Flow_MGR [mgt_client, mmv, m](isv, osv, r_client)
    [] mgt_client ! Dreq !v ; m!dis_other_stream!osa ;
        (mmv !Dreq!isv; EXIT ||| m!Dreq !osv ; EXIT )
        >> Client_One_Flow_MGR [mgt_client, mma, m](isa, osa, r_client)
    [] m ? er:error!osa ;
        ( mma!Dreq !isa; EXIT ||| mmv!Dreq !isv; EXIT ||| m!Dreq!osv;EXIT )
    >> i NEW <r_client!er!osa ; STOP> ; STOP
    [] m ? er:error!osv ;
        ( mma!Dreq !isa; EXIT ||| mmv!Dreq !isv; EXIT ||| m!Dreq!osa; EXIT )
            >> i NEW <r_client!er!osv ; STOP> ; STOP
ENDPROC (* Client_Two_Flows_MGR *)
ENDPROC (* MGR *)
ENDSPEC
```

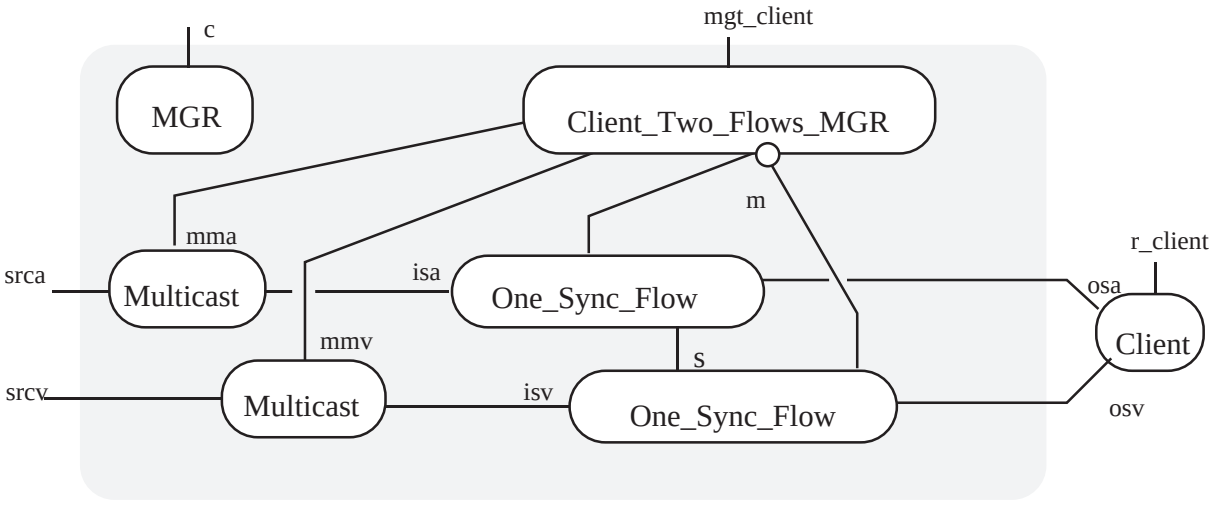

Figure 3 the binding object with one client connected to both the audio and video casts

5 CONCLUSION

Mobile-Timed-LOTOS extends the classical formal description technique LOTOS with two new features: *quantitative time* and *mobility*. These two enhancements make MT-

LOTOS very well adapted to the specification of complex dynamic systems like an ODP binding object.

Quantitative time allows the precise description of real-time aspects. With the recent evolutions in networking, e.g. multimedia, such aspects become more and more crucial and being able to specify them is mandatory.

Mobility allows the writing of specifications whose structure can be dynamically reconfigured. New processes can be created as well as new connections between them, through new gates. This is typically the kind of flexibility required by a binding object which conveys interactions between interfaces that can change in time. Our binding object for example allows new customers to be connected on demand.

We have presented here a building blocks approach. Such an approach is made possible by the modularity that MT-LOTOS permits. We have defined a series of generic components that can be put together very easily to form various combinations. Note that LOTOS already offers a constraint oriented style. However, the structure of LOTOS specifications is static. The dynamic evolution of the binding object cannot be described as naturally as with MT-LOTOS. In particular, the inability to create new gates forces to resort to specification "tricks": all the customers are connected to the same gate but each one is differentiated by a special attribute added to the gate. This results in less generic building blocks that cannot be reused easily in other contexts.

REFERENCES

T. Bolognesi, E. Brinksma (1987), "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems 14, pp25-29.

[ISO 88] International Standard 8807 (1988) - "LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behavior".

[ISO/ODP] ODP reference Model, Parts 1, 2, 3, 4: ISO/IEC 10746-1/2/3/4 or ITU-T X.901/2/3/4.

L. Léonard, G. Leduc, (1994) *An Enhanced Version of Timed LOTOS and its Application to a Case Study,* in: R. Tenney, P. Amer, Ü. Uyar, eds., Formal Description Techniques, VI (North-Holland, Amsterdam) 483-98.

L. Léonard, G. Leduc, D. de Frutos, L. Llana, C. Miguel, J. Quemada, G. Rabay, (1995) *Belgian-Spanish Proposal for a Time Extended LOTOS*, in J. Quemada, ed., Revised Draft on Enhancements to LOTOS, ISO/IEC JTC1/SC21/WG7 N1001.

R. Milner, J. Parrow, D. Walker (1992): "A Calculus of Mobile Processes: Parts I & II" — Journal of Information and Computation 100, p 1-77.

E. Najm, J-B. Stefani, A. Février 1995 *Towards a Mobile LOTOS* in: G. Bochman, R. Dsouli, O. Rafiq, eds., Formal Description Techniques, VIII, Montréal, Canada.

E. Najm, J-B. Stefani, A. Février (1994) *Introducing Mobility in LOTOS* in J. Quemada, ed., Revised Draft on Enhancements to LOTOS, ISO/IEC JTC1/SC21/WG1 N1349.

J.B. Stefani, L. Hazard, F. Horn (1992). "Computational model for multimedia applications based on a synchronous programming language." Computer Communications, 15(2): 114-128.

J.B. Stefani(1990): "Open Distributed Processing: The Next Target for the Application Of Formal Description Techniques" in the 3rd International Conference On Formal Description Techniques FORTE 90 – Madrid, Spain .

C.A. Vissers: "FDTs for Open Distributed Systems, a Prospective View" in Proceedings 10th IFIP WG6.1 Workshop on Protocol Specification, Testing and Verification, Ottawa, Canada (1990).

[WD 95] ISO/IEC JTC1/SC21/WG7 N1001 Revised Working Draft on Enhancements to LOTOS. May 95.

BIOGRAPHY

Arnaud Février graduated from the École Nationale Supérieure des Télécommunications de Bretagne (1988) and Université de Nice-Sophia-Antipolis (DEA 1989). He worked as a computer scientist in a private company. He is now a Ph.D. Student at the École Nationale Supérieure des Télécommunications. His thesis subject treats the formalization of real-time and object paradigms.

Guy Leduc received his degree in electrical engineering in 1983, and his Ph. D. degree in computer science in 1991 from the University of Liège, Belgium. He is "Maître de Recherches F.N.R.S." (Senior Research Associate of the Belgian National Fund for Scientific Research) at the University of Liège in the research unit in networking headed by Professor A. Danthine. His activities and research interests include computer networks and the theory and application of formal description techniques.

Luc Léonard received his degree in electrical engineering in 1991 from the University of Liège, Belgium. Since 1991 he has been working for the Belgian National Fund for Scientific Research (F.N.R.S.) at the University of Liège in the department headed by Professor A. Danthine. His activities and research interests focus on the formal description technique LOTOS. He works on the definition of a timed extension of LOTOS. He also applied LOTOS in the European project ESPRIT/OSI95 on the design of high performance protocols.

Elie Najm Elie Najm has the grade of "Habilitation of Director of Research" from the university of Paris VI (1993) and has graduated from Ecole Polytechnique (1975) and ENST (1977). He is currently Senior Lecturer at ENST (formal methods, networks and distributed systems) and head of the DAF research group - Formal Methods for Distributed Systems & Applications.