1
Multimedia in the E-LOTOS Process Algebra

Guy Leduc
Research Unit in Networking, University of Liège, Belgium

1.1 Introduction

The description and analysis of multimedia distributed systems is a challenge for formal techniques. Traditionally these techniques have focussed on the description of functional aspects of distributed systems and their mathematical analysis. However, until recently, almost all formal techniques were still unable to tackle real-size problems, which require more expressivity or flexibility to describe complex data types, to define generic components, to support their easy combination and/or reuse in several contexts, and to describe sophisticated architectures where many processes are involved.

In addition to these more classical functional aspects, multimedia systems are basically real-time systems, whose main behaviour is so intrinsically related to timing requirements, that their specifications with traditional techniques would lack the essence of the behaviour.

Therefore, in recent years, lots of research effort has been dedicated to that problem. This is the case in particular for process algebras that we will consider in the present chapter.

The first process algebras that were able to express time quantitatively are called synchronous process algebras. Examples are SCCS [Mil83], Meije [AB84] or CIRCAL [Mil85]. These languages are called synchronous because actions can only occur at precise regular times, like in synchronous circuits for which they where mainly targeted.

A more flexible, also called asynchronous, approach was needed to describe software systems which are not so closely driven by a clock. In fact, a better model of real-time systems is a model in which actions (i.e. processing) and time passing (i.e. idling) alternate more freely. At a certain level of abstraction, one can even consider that actions are atomic and instantaneous. This leads to an elegant simple model where systems alternate between two
phases: an execution phase where actions are executed instantaneously, and an idle phase where the system ages [NS92]. For actions whose duration is such that they cannot be abstracted as instantaneous, it suffices to split them into a beginning and an ending action.

This principle has lead to many extensions of well-known process algebras, such as Timed CSP [RR88, DS94], Timed CCS [MT90, Wan91, Han91], Timed ACP [BB91, BB96, Gro90] and Timed LOTOS [MFV94, CdO94, BLT94b, BLT94a, LL97, LL98, QMdFL94].

New timed process algebras have also been proposed, such as TPL [HR95] and ATP [NS94].

Even though these languages propose interesting facilities to specify real-time behaviours, very few of them were designed as complete languages suitable for tackling real-size systems. Basically, they are not very well equipped for software engineering in general:

- most of them, except some LOTOS extensions, are basic algebras with little support to define more complex data structures;
- they have no module system, or a limited one;
- they do not support exception handling, subtyping, ...

For these reasons, ISO has decided to improve LOTOS more substantially than by merely adding real-time. This initiative has lead to E-LOTOS [ISO98], which has been presented in chapter 5.

In the present chapter, we will illustrate how E-LOTOS can be used to describe an ODP multicast multimedia binding object of some complexity. The timing features of the language will be briefly recalled, and will play a major role in the description of the example, but many other features of E-LOTOS are also very useful to obtain a more modular and more readable specification. Users familiar with LOTOS will also learn how some E-LOTOS features can advantageously replace the traditional LOTOS ones.

Our approach is based on a single language. In the next chapter, the same binding object will be described using a multi-paradigm approach.

1.2 Timing facilities in E-LOTOS

The expression of real-time behaviours in E-LOTOS is based on three simple language constructions:

- A type time with associated operations. Usually specifiers simply define this type as a renaming of either the natural numbers to get a discrete time domain, or the positive rational numbers to get a dense time domain, e.g.

  type time renames nat endtype
• A wait operator, which introduces a delay. For example:

input; wait(1); output

More interestingly, we can also specify nondeterministic delays. For example, the next specification describes a process that will choose internally some delay (within some bounds) and introduce it between input and output:

input;
?t := any time [(min <= t) and (t <= max)]; (* The question mark always indicates a variable binding
It is used consistently both for classical assignments and for variable binding resulting from synchronisations *)
wait (t);
output

The timed value can also be received from the environment of the process, such as in

gt ?t; (* we suppose that this gate is typed
in such a way that t should be a time value *)
input ?data; wait (t); output !data

• An extended communication operator, which is sensitive to delay:

input ?data @?t; (* ?data is the classical variable binding,
@?t is the time capture, see below *)
wait (10-t);
output !data

In the former example, t is bound to the time at which the communication along input happens (measured from when the communication was enabled). When the variable binding is replaced by a value expression, the same language construction can be used as a compact way to specify a unique possible occurrence time. For example, in input ?data @!3, the input action is only enabled at time 3, which is the only timed value allowed by the value expression. This behaviour is in fact equivalent to input ?data @ ?t [t = 3].

Besides these three constructions, E-LOTOS is based on the following basic principles:

• The internal action \( i \) cannot be delayed. It should occur immediately, or another competing action has to occur immediately instead.
• Observable actions are delayed until synchronization is made possible by the environment. They cannot be forced to occur before that.
• Exceptions cannot be delayed. The exception handler is started instantaneously.
• When a process successfully terminates, the following process starts immediately (no delay). Note however, that when a process is composed of several processes in parallel, termination can only occur when all subprocesses have finished.

• Two successive actions can occur at the same time. There is no minimum delay between them. Infinitely many actions can thus potentially occur in a finite time.

• A function is merely an immediately exiting process: it does not communicate and executes instantaneously.

1.3 Specification of the common example

1.3.1 A multicast multimedia binding object

In this section we first present informally the ODP Binding Object and then its formal description in E-LOTOS. This object was first used in [FLL96].

Part 1 of this book discusses ODP concepts in detail. Therefore, we will only recall a few of them here. In the ODP Computational Model the binding objects are used to convey interactions between interfaces of other objects. In fact, in the Computational Model, the programmer can 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 a binding object, the programmer may incorporate the Quality of Service (QoS) requirements (order, timeliness, throughput, etc.) on the transport of 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 modelled as special types of signal interfaces. A signal is an operation name and a vector of values. 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. 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).

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 analyse 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.

The binding object we consider in this chapter executes the following operations:

- It listens to a source emitting two synchronized 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 the audio and video flows.
- It tries to enforce the intra and inter synchronization of flows and notifies failures to do so.

The source flows are 25 images per second, i.e. the video stream is composed of packets delivered every 40 ms, and the sound is sampled every 30 ms, i.e. a sound packet is delivered every 30 ms.

We suppose that the two sources do not deviate from the above figures and that both flows are fully synchronized. The binding object accepts these flows and delivers them to any requesting customer. Since the binding object will encapsulate the behaviour of a concrete network, it will have to deal with usual networking problems, such as jitter, packet loss, end to end delay, ... Nevertheless, the customers expect a minimal QoS, which is twofold:

- Each flow must respect a QoS: sound should have no jitter, and video can have a jitter of 5 ms, i.e. consecutive images may be separated by 35 to 45 ms,
- Both must be reasonably synchronous, as defined below. This is known as lip synchronization.

Lip synchronization is considered correct if the sound is not too late from, or ahead of, the corresponding lip movement. The actual figures are:

- The sound must not come more than 15 ms ahead of the lip movement.
• The sound must not come later than 150 ms after of the lip movement.

1.4 TE-LOTOS specification of the multimedia multicast ODP binding object

The binding object will be defined as a generic process handling generic data. It will be composed of several other processes and will use some basic data types. Therefore, the best way to specify the system is by way of a generic module as follows:

interface data is
  type data
endint

generic binding-object (D: data) imports NaturalNumbers is

type stream is a | v endtype (* audio and video streams *)
type client renames Nat endtype (* client ids are just numbers *)
type time renames Nat endtype (* time is discrete *)
type clients is set of client endtype
type fifo is list of client endtype
type req_code is Creq_a | Creq_v | Creq_av | Dreq endtype
  (* enumeration of possible request primitives *)
type er_code is e_delay | e_jitter | e_sync | released endtype
  (* enumeration of possible error codes *)
type id_data is (client,data) endtype
  (* a record composed of a client id and some data *)
type ctrl is (req_code,client) endtype
  (* a record composed of a request code and a client id *)
type er_tuple is (er_code,client) endtype
  (* a record composed of an error code and a client id *)
type release is (req_code,stream,client) endtype
  (* a record composed of a request code, a stream type
   and a client id *)

function empty := {} endfunc
value epsilon: time is 1 endval
value mindelay: time is 0 endval (* Minimum transit delay *)
value maxdelay: time is 100 endval (* Maximum transit delay *)
value arate: time is 30 endval
(∗ Interval between two audio data packets ∗)
value vrate:time is 40 endval

(∗ Interval between two video data packets ∗)
value ajitter:time is 0 endval

(∗ Maximum jitter on audio packets ∗)
value vjitter:time is 5 endval

(∗ Maximum jitter on video packets ∗)
value abv:time is 15 endval

(∗ Maximum lead of the audio stream over the video stream ∗)
value vba:time is 150 endval

(∗ Maximum lead of the video stream over the audio stream ∗)

(∗ And we would insert here all the processes defined below ∗)
endgen

We first identify a collection of generic components that are suitable for the specification of functional and QoS requirements of a multimedia binding object. We present them below, together with their E-LOTOS specification. In these specifications, dt represents a generic data packet, which can contain audio or video data.

The first component is called Medium. This component describes a point-to-point transmission medium. Packets are received on gate ist (input stream), and are delivered on gate ost (output stream). 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, an unbounded nondeterministic delay is introduced before the delivery on ost. In parallel, a new occurrence of Medium handles the subsequent packets.

Process Medium [ist:data,ost:id_data] (id: client) is
  ist ?dt;
  (?	:= any time; wait (t); ost(!id,!dt); stop
   |||
   Medium [ist,ost] (id))
endproc (∗ Medium ∗)

Next we add the FIFO_Const component. It ensures that the packets are delivered in the same order as they are received. This component also considers gates ist where packets are received, and ost where these packets are delivered. In process FIFO_Const, the ordering is handled with an
appropriate data structure: \( q \), that describes a FIFO queue. At any time, 
FIFO\_Const can accept (ist\?dt) a new packet that is appended at the end 
of \( q \), or deliver ((ost!id!head(q))) the first packet in \( q \).

process FIFO\_Const [ist:data,ost:id_data](id:client,q:fifo) is 
\[
\text{ist ?dt; FIFO\_Const [ist,ost] (id,tcons(dt,q))}
\]
\[
\text{(* tcons = tail cons *)}
\]
\[
\text{if not(IsEmpty(q)) then}
\]
\[
\text{ost (!id,!head(q));}
\]
\[
\text{FIFO\_Const [ist,ost] (id,tail(q))}
\]
\[
\text{endif}
\]
endproc (* FIFO\_Const *)

The third component Delay\_Const ensures that at least a minimal delay 
delmin elapses between the reception of a packet on ist and its delivery on 
ost. Here again, the same two gates are considered.

process Delay\_Const [ist:data,ost:id_data] 
\[
\text{(id:client,delmin:time) is}
\]
\[
\text{ist ?dt ;}
\]
\[
\text{(Wait (delmin); ost (!id,!dt); stop}
\]
\[
\text{)||}
\]
\[
\text{Delay\_Const [ist, ost] (id, delmin))}
\]
endproc (* Delay\_Const *)

The fourth component 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 exception is raised. After the 
reception of a packet, Delay\_Obs proposes ost (!id,!dt) during a time 
delmax. On the other hand, the exception error (e\_delay) is delayed by 
delmax+epsilon. In other words, it is raised when the delivery cannot occur 
anymore.

process Delay\_Obs [ist:data,ost:id_data] 
\[
\text{(id:client,delmax:time)}
\]
\[
\text{raises [error:er\_code] is}
\]
\[
\text{ist ?dt ;}
\]
\[
\text{(ost (!id,!dt)@?t [t <= delmax]; stop}
\]
\[
\text{[])
\]
\[
\text{wait(delmax+epsilon); raise error (e\_delay))}
\]
\begin{verbatim}

Delay_Obs [ist,ost](id,delmax)
endproc (* Delay_Obs *)

The fifth component 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:id_client] (id:client,jmin:time) is
  loop
    ost (!id,?dt);
    wait (jmin)
  endloop
endproc (* Jitter_Const *)

The sixth component 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 raises an exception if this happens.

process Jitter_Obs [ost:id_data] (id:client,jmax:time) raises [error:er_code] is
  ost (!id,?dt);
  loop
    ost (!id,?dt)@?t [t <= jmax]
    []
    wait (jmax+epsilon); raise error (e_jitter)
  endloop
endproc (* Jitter_Obs *)

The next process, called One_Ind_Flow, gives a first example of the modularity allowed by E-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 exception is raised 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 exception is also raised 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 (e.g. with no lower bound on the transmission delay or with no preservation of the order) resulting in a less constrained flow.

process One_Ind_Flow [ist:data,ost:id_data]
   (id:client-id,q:fifo,delmin,delmax,jmin,jmax:time)
   raises [error:er_code] is
       (Medium [ist,ost] (id)
       ||
       FIFO_Const [ist,ost] (id,q)
       ||
       Delay_Const [ist,ost] (id,delmin)
       ||
       Delay_Obs [ist,ost] (id,delmax)
   )
[::-ost]
       Jitter_Const [ost](id,jmin)
[::-ost]
       Jitter_Obs [ost](id,jmax)
endproc (* One_Ind_Flow *)

We continue with process Two_Sync_Flows. This component gives a new example of the modularity allowed by E-LOTOS. One_Ind_Flow was already the composition of several features. Two_Sync_Flows combines two flows and enhances the result with Inter_Sync_Const, a synchronisation mechanism between the flows. Again, the addition of a constraint is simply obtained by putting a new process in parallel. Furthermore, each flow can be stopped independently by way of a disrupt message on gates ma or mv, and in that case the interflow constraint will be removed too, i.e. the constraint is replaced by the neutral process Sink.

   raises [error:er_code] is
       (* gates isa and isv are the audio and video source gates respectively
          gates osa and osv are the audio and video output gates respectively
          *)
       ((One_Ind_Flow [isa,osa] (id,empty,mindelay,maxdelay, 
                                    arate - a_jitter, arate + a_jitter)
> ma (!Dreq,!id); Sink[isa]

|||
One_Ind_Flow [isv,osv] (id,empty,mindelay,maxdelay,
vrate - vjitter,vrate + vjitter)
> mv (!Dreq,!id); Sink[isv])

||[osa,osv,ma,mv]|
((Inter_Sync_Const [osa,osv] (id,0,0)
> (ma (!Dreq,!id) □ mv (!Dreq,!id))
eendproc (* Two_Sync_Flows *)

The Sink process enforces no constraint on the actions occurring on a gate. It is specified as a process with a single gate st, and no predicate or time constraint restricts the acceptance of packets on st.

process Sink [st: data] is
  st ?dt; Sink [st]
eendproc (* Sink *)

The Inter_Sync_Const process controls the synchronisation between the packets delivered by two flows. The way Inter_Sync_Const is combined with the flows is illustrated in the previous component: Two_Sync_Flows. The effect of Inter_Sync_Const is to ensure that the packets on one flow are not delivered too late or too early with respect to the packets on the other flow. If these constraints cannot be met, an exception is raised and the two flows are interrupted. The meanings of the parameters used in this process are as follows:

- last_a is the time elapsed since the last audio packet delivery
- last_v is the time elapsed since the last video packet delivery

process Inter_Sync_Const [osa,osv:id_data]
  (id:client, last_a,last_v:time)
  raises [error:er_code] is
osa (!id,?dt)@?t [t >= vrate - last_v - abv];
  Inter_Sync_Const [osa,osv] (id,0,last_v + t)
□

osv (!id,?dt)@?t [t >= arate - last_a - vba];
  Inter_Sync_Const [osa,osv] (id,last_a + t, 0)
□

wait (vrate - last_v + vba + epsilon); raise error (e_sync)
□
wait (arate - last_a + abv + epsilon); raise error (e_sync)
endproc (* Inter_Sync_Const *)

The next process MGR is the main one. It realizes the multicasting by creating as many channels as necessary. A new flow manager One_client_MGR is created on receipt of a c!Creq... request. When a client requests a channel, it has to provide its id. This id will allow the MGR to connect the channel to gates r!id, mgt!id and osa!id (and/or osv!id). The MGR also ensures that a client can request at most one channel.

process MGR [c:crt1, isa, isv: data, osa, osv: id_data, 
    mgt: release, r: er_tuple] (Ids: Clients) is
(* gate c is the controlling gate of the binding object 
gate r is used to report errors to clients 
gate mgt allows clients to manage their flow(s) *)

c (?cmd, ?id) [(id notin Ids) and 
  (cmd = Creq_a or cmd = Creq_v or cmd = Creq_av)];
  
  (One_client_MGR [c, isa, isv, osa, osv, mgt, r] (id, cmd) 
   | [isa, isv] 
   MGR [c, isa, isv, osa, osv, mgt, r] (insert(id, Ids))
  )

[]
isa ?dt; MGR [c, isa, isv, osa, osv, mgt, r] (Ids)
[]

isv ?dt; MGR [c, isa, isv, osa, osv, mgt, r] (Ids)
endproc (* MGR *)

The next process One_Client_MGR describes the behaviour of a client’s manager. There are three distinct cases according to the nature of the client’s request, which can be for:

- an audio stream: One_Ind_Flow [isa, osa]
- a video stream: One_Ind_Flow [isv, osv]
- an audio stream and a video stream synchronized together:
  Two_Sync_Flows [isa, isv, osa, osv, ma, mv]

process One_Client_MGR [c:crt1, isa, isv: data, osa, osv: id_data, 
    mgt: release, r: er_tuple] 
  (Id: Client, cmd: req_code) is

  trap exception error: (?er: er_code)
is r (!er,!id);       (* notify user *)
c (?cmd,!id) [(cmd = Creq_a or cmd = Creq_v
   or cmd = Creq_av)];
   (* ready to recreate the flow(s) *)
One_Client_MGR [c,isa,isy,osa,osv,mgt,r] (id,cmd)
endexc
in
case cmd is
  Creq_a ->
    (One_Ind_Flow [isa,osa] (id,empty,mindelay,maxdelay,
                             arate,ajitter)
    |||
    One_Flow_MGR [mgt] (id,a))
| Creq_v ->
    (One_Ind_Flow [isy,osv] (id,empty,mindelay,maxdelay,
                             vrate,vjitter)
    |||
    One_Flow_MGR [mgt] (id,v))
| Creq_av ->
  hide ma,mv:ctrl in
    (Two_Sync_Flows [isa,isy,osa,osv,ma,mv] (id)
     |[ma,mv]|)
    Two_Flows_MGR [mgt,ma,mv] (id)
endhide
case
endtrap
dendproc  (* One_client_MGR *)

The final components are the processes that control the flows. We first
define the process managing a single flow, and then another one managing
two synchronized flows.

process One_Flow_MGR [mgt:release] (id:client,s:stream)
  raises [error:er-code] is
  mgt (!Dreq!,s,!id); raise error (released)
endproc  (* One_Flow_MGR *)

process Two_Flows_MGR [mgt:release,ma,mv:ctrl] (id:client)
  raises [error:er-code] is
  mgt (!Dreq!,a,!id);
  ma (!Dreq!,id);
1.5 Analysis of E-LOTOS specifications

We have illustrated that E-LOTOS allows us to produce elegant formal
descriptions of complicated objects. However, the main interest of using
the language lies in the automatic computations we could perform on such
specifications. This requires of course the development of appropriate tools.

Prototype tools exist for LOTOS NT [Sig99], which is a slight variant of
E-LOTOS. The TRAIAIN tool performs the following operations:

- lexical and syntactic analysis,
- static semantic analysis, including modules,
- flattening of non generic modules,
- C code generation for the data types

A translation from E-LOTOS to timed automata would also open very
interesting perspectives. The idea is of course to define a mapping between
E-LOTOS and a timed automaton, allowing the former to benefit from the
model-checking theory and tools developed for the latter. Then, the KRONOS tool [DOTY96] can be used. It takes a timed automaton and a TCTL
formula as input and checks whether the formula is verified on the automaton. In [DOY95] a similar method is proposed for a subset of ET-LOTOS
[LL97, LL98], which is roughly the timed subset of E-LOTOS.

More recently, this work has been extended to cope with a larger subset of
ET-LOTOS. In [Her98] an hybrid automaton model, called ETL-automaton,
is proposed, which is especially designed to support ET-LOTOS. Informally,
it can be seen as a timed automaton extended with memory cells and ASAP
(as soon as possible) transitions. The values of the memory cells and clocks
can be used in guards to constrain the occurrence of transitions. Each state
has an invariant condition, which must be verified for the automaton to
stay in it. A transition can reset clocks and change the memory cells. In
particular, it is possible to capture the clock values in memory cells, which
makes it possible to capture the occurrence time of a transition in a variable,
i.e. to model the @?t operator. The transitions are labelled either with a
LOTOS action (i.e. an i or an observable gate with a list of attributes) or

Client_One_Flow_MGR [mgt] (id,v)

mgt (!Dreq,!v,!id); mv (!Dreq,!id);
Client_One_Flow_MGR [mgt] (id,a)
endproc (* Two_Flows_MGR *)
with a third special action marking the expiry of a delay. Each transition is
also associated with a predicate on the clocks and the variables, which con-
strains its occurrence and determines the possible values of the attributes
when the label is an observable action. This ETL-automaton model covers
nearly all the features of ET-LOTOS. The restrictions merely ensure the
finiteness of the resulting automaton (e.g., no recursion through the parallel
and the left part of the enabling and disabling operators) or ease the transla-
tion process (e.g., no recursion through the hiding operator, nor unguarded
recursions through the delay or the guard operators). A simulator of ETL-
automata has been developed, which thus supports a very large subset of
the language. In [Her98] the author also studies how ETL-automata can be
mapped onto underlying models of existing model-checkers such as HyTech
[HHWT95], KRONOS [DOTY96] and UPPAAL [LPW97]. Although the hy-
brid automata accepted by HyTech are the most general ones among these
three, they are less expressive than ETL-automata, so that further restric-
tions should be considered. Basically, the ET-LOTOS expressions used in
delays, life-reducers, selection predicates, offers, etc. must be linear, and
the hide operator can only be used on non time-restricted actions. This still
covers a large subset of the language, and the semi-decidable algorithm of
HyTech can be used for reachability analysis. As regards KRONOS, its more
restricted timed automaton model, motivated by decidability purposes, re-
quires further restrictions. Basically, at first glance it seemed difficult to
model the time capture operator of ET-LOTOS because there are no mem-
ory cells. However, [Her97] shows that an extension of timed automata with
semi-timers remains decidable and can support this operator. A timer is a
clock that can be stopped and restarted, and a semi-timer is always reset
before being restarted. The interesting feature is that a semi-timer can be
modeled by two auxiliary ordinary clocks, in fact by their difference, so
that, at the price of a larger number of clocks, the time-capture operator
of ET-LOTOS can be supported by the KRONOS timed automata. Any-
way, other restrictions exist: e.g. expressions in delays should be constants,
and expressions in selection predicates and guards are restricted. The same
conclusion applies to UPPAAL, but the supported subset of ET-LOTOS is
slightly larger, especially as regards the expressions in selection predicates
and guards.

1.6 Conclusion

We have illustrated how E-LOTOS can be used to describe complex objects
such as an ODP multicast multimedia binding object. The structuring capa-
bilities of E-LOTOS, together with its real-time features, were particularly relevant to achieve concise and readable specifications.

In contrast to our approach which is based on a single specification language, the next chapter will describe the same binding object using a multi-paradigm approach.

Bibliography


