Specifying Aspects of Multimedia in LOTOS

Dr Richard O. Sinnott,
GMD Fokus,
Kaiserin-Augusta-Allee 31,
10589 Berlin, Germany
sinnott@fokus.gmd.de

Abstract

The formal specification language LOTOS provides a model of systems where the temporal ordering of actions is specified. LOTOS typically does not deal with the specific times at when actions can occur. Most approaches for specifying real time systems in LOTOS have either extended the language, e.g. with timing aspects, or used the language in conjunction with a temporal logic. We argue that such approaches are not always necessary and that LOTOS is much more flexible than sometimes claimed. To support our arguments we show how LOTOS can be used to specify a multitude of timing issues that can be associated with the production and consumption of multimedia flows.

1. Introduction

LOTOS [5] is a formal specification language originally developed by the international standardisation organisation to specify protocols for the interconnection of systems. The behaviour of a system is modelled by an ordering of events in LOTOS. As such, it does not attempt to deal with quantitative timing considerations, e.g. where events occur at certain specific times or within specific time intervals. Numerous approaches have been put forward for dealing with these issues, e.g. extensions to the language [3] or using LOTOS together with a temporal logic [1]. In this paper we show how LOTOS is expressive enough to specify a range of temporal aspects of systems, focusing in particular on multimedia timing issues. LOTOS is divided into two parts: a data modelling part based on the algebraic specification language Act One [2] and a process algebra part based on CSP and CCS [4,6]. Typically data is specified in Act One and used, e.g. passed around between processes, in the process algebra. We shall see that through a combination of these languages a wide range of time aspects of systems can be specified.

2. Modelling an Information Flow in Act One

We consider here a generic idea of information flow where the flow of information is represented by a sequence of *frames*. A frame may be regarded as a particular item in the flow of information. Each frame in an information flow can be considered as a unit consisting of data (this may be compressed) which we represent by *Data* and a time stamp used for modelling the time at which this particular frame was sent or received. It is also often the case in multimedia flows that particular frames are required for synchronisation, e.g. synchronisation of audio with video for example. Therefore we associate a particular *Name* with each frame. This can then be used for selecting a particular frame from the flow as required. From this, we may model a frame as:

```
type Frame is Name, NaturalNumber, Data
    sorts Frame
    opns makeFrame: Data, Nat, Name → Frame
              getData: Frame → Data
              getTime: Frame → Nat
              getName: Frame → Name
              setTime: Nat, Frame → Frame
    eqns forall d: Data, s,t: Nat, n: Name
```

```
 ofsort Data  getData(makeFrame(d,t,n)) = d;
ofsort Nat  getTime(makeFrame(d,t,n)) = t;
ofsort Name getName(makeFrame(d,t,n)) = n;
ofsortFramesetTime(s,makeFrame(d,t,n)) = makeFrame(d,s,n);
endtype (* Frame *)
```

It should be noted here that we model time as a natural number, i.e. time is modelled discretely. It might well be the case that real (dense) time could be used although we note that the real numbers are not explicitly defined in the LOTOS standard. We also introduce sequences of these frames:

```
type FrameSeq is Frame
  sorts FrameSeq
  opns makeFrameSeq: → FrameSeq
      addFrame: Frame, FrameSeq → FrameSeq
      remFrame: Frame, FrameSeq → FrameSeq
      timeDiff: Frame, Frame → Nat
  eqns forall f1, f2: Frame, fs: FrameSeq, n1,n2: Name
        ofsort FrameSeq getTime(f1) le getTime(f2) =>
              addFrame(f1,addFrame(f2,makeFrameSeq)) = addFrame(f2,makeFrameSeq);
endtype (* FrameSeq *)
```

For brevity we do not supply all equations. Frames are added to the sequence provided they have increasing timestamps. An operation is provided to get the time difference between time stamps of two frames. From this abstract information flow model we are able to consider the timing issues for the production and consumption of information flows. For simplicity we consider production and consumption of a single flow of information.

3. Realising the Flow of Information in the Process Algebra

To consider flows of information it is necessary to consider how frames are timestamped when sent from a producer. With this information the consumer can determine what course of action to take with the received frame. Therefore it is necessary to consider how time is to be represented in LOTOS. There are several choices for modelling time in LOTOS. We consider some of these options.

It might be the case that all processes have access to a global clock from which they can establish the current time. The frames sent from a producer could then be timestamped with this time value. The modelling of global clocks in LOTOS is not without its problems however. For example, one model of a clock might be:

```
process Clock[ t ](tnow: Nat) :noexit:=
        t !tnow; Clock[t](tnow) [] i; Clock[ t ](tnow+1)
endproc (* Clock *)
```

Here the clock either outputs the current time or an internal event occurs and the time is incremented. This model of time is limited when modelling flows of information. Here, the time itself is based on non-deterministic internal events. Flows of information have explicit temporal requirements that must be satisfied. As such this model of time is not sufficiently expressive. Modifying this process so that the non-determinism is removed, e.g. each time the clock is referenced the time is given and then incremented, is also an unsatisfactory model since it can adversely influence concurrent behaviours [7]. Concurrency loosely implies that they can happen at the same time. If processes are time dependent though then this is not the case, i.e. one must happen after the other if they access the global clock.

A second approach is to model time in the formal parameter list associated with a process modelling a flow. This might for example be represented by:

```
 process ProduceAction[ g,s ](toSend: FrameSeq, tnow: Nat):noexit:=
 g !setTime(tnow,head(toSend));
  (* other behaviour and recurse with frame removed from toSend and time incremented *)
 [] s ?tupdate: Nat;  ProduceAction[ g,s ](toSend,tupdate)
endproc (* ProduceAction *)
```

Here we have a local model of time. That is, the process itself keeps track of its current time. With this model of time it is possible to model processes running at different speeds say, e.g. where different *tnow* variables and their modifications exist in different processes. If this approach is taken then there should exist some means whereby the current local time can be set, e.g. so that processes can re-align their clock values (*tnow* variables). We introduce the gate *s* for this purpose. The current time (local to the process) is time stamped onto the frame being sent through the operation *setTime*. The value that the time variable *tnow* is increased by is proportionate to the rate of the flow. If the modification to the time variable is the same each time a frame is sent then we have an isochronous flow.

It is often the case that levels of control are required for modifying flows of information, e.g. send faster or slower as the case might be. In this model, the control is given by the recursive call and how the *tnow* variable is incremented. As such the flexibility of manipulating *tnow* is limited.That is, the operations given in the recursive call are static. Further, if *tnow* represents the local time, then operations to manipulate how time progresses might also seem unnatural. To overcome this, it is possible to model the rate of flow of frames to be sent as a formal parameter. This might be represented by:

```
process ProduceAction[ g, m ](toSend: FrameSeq, tnow: Nat, rate: Nat):noexit:=
 g !setTime(tnow+rate,head(toSend));
   ...(* other behaviour and recurse with frame removed from toSend and time incremented *)
 [] m ?newRate: Nat [newRate gt 0];  ...(* other behaviour and recurse with new rate set *)
 [] s ?tupdate: Nat;  ...(* other behaviour and recurse with new current time set *)
endproc (* ProduceAction *)
```

Here we note that we model the rate as a natural number. This allows the rate of flow to be sped up or slowed down depending on whether the rate is decreased or increased respectively. When instantiated, predicates should be given to ensure that the rate is greater than zero. Proposed new rates are checked to ensure that they are greater than zero. Having a zero or negative value for the rate would allow consecutive frames in the sequence to have decreasing or equal time stamp values. This could destroy the temporal integrity of the flow of information, for example where a negative rate was given that was greater than the time difference between two consecutive frames in the flow.

It is likely that the gate *m* will be internal to the object with which the interface is associated. This gate might be used for management and control purposes e.g. another interface used for controlling the flow of information might receive a message to slow the speed of flow up (or slow it down) and this information then used to establish the new rate.

A third alternative is to model time entirely through Act One. That is, when the sequence of frames is created, the time stamps are given there. For example, a constructor operation (fs_1) that returns a sequence of frames might have equations that give the time stamps explicitly, e.g. *addFrame(makeFrame(*d_1*,*t_1*,*n_1*),...,addFrame(makeFrame(*d_i*,*t_i*,*n_i*), makeFrameSeq)))*. Here the time stamps t_i might be represented by explicit natural numbers. With this model of frame sequences, frame production may be represented by:

```
process ProduceAction[ g ](toSend: FrameSeq):noexit:=
  g !head(toSend);   ...(* other behaviour and recurse with frame removed from toSend *)
endproc (* ProduceAction *)
```

This model of timed frames is the simplest to represent in the process algebra. Unfortunately, the actual sending of the frame itself is given by the occurrence of the process algebra event. As a result, the time stamping achieved in Act One is independent of the event offer occurrence in the process algebra. Another problem with this model of time stamping frames is that anything like a realistic sized sequence of frames would not be well suited to Act One specification. That is, the equations required for an information flow of several thousand irregularly time stamped, i.e. consecutive frames in the sequence may have time stamps whose differences are not constant, frames say would be far too verbose to be practicable. It might be the case that only isochronous flows are considered. In this case, the constant value of the difference between time stamps of consecutive frames could be specified directly in the equations associated with the frame sequence.

Consumption of frames typically has different requirements placed upon it. The need to continually monitor the time stamps of the incoming flow of information is of particular importance. Due to the potential spatial separation of producers and consumers of flows of information, there is often a non-negligible time difference between the sending of a frame from a producer to its arrival at the consumer. This time difference is heavily dependent upon the connection between the producer and consumer of the flow. This connection is likely to have a limit on the information that can be passed through it at any given time. The current usage of this connection will thus influence the speed at which information is passed from producer to consumer. These issues and their modelling in LOTOS are discussed in more detail in [7]. A consumer of an information flow may be represented by:

```
process ConsumeAction[ g,s ](recFrames: FrameSeq, limit,tnow: Nat):noexit:=
 g ?inFrame: Frame;
  (    [Diff(getTime(inFrame) - tnow) gt limit ] → ...(* frame too early or too late behaviour *)
   []  [Diff(getTime(inFrame) - tnow) le limit ]→
        ...(* other behaviour, e.g. display frame and recurse with time incremented *)
           (* or recurse with frame added to received frames and time incremented *))
  []    s !tnow;  ConsumeAction[ g,s ](recFrames,limit,tnow)
endproc (* ConsumeAction *)
```

Here we check that the time of the incoming frame is within some limit. We focus in more detail on this limit and other timeliness considerations shortly. If the frame arrives outside of the allowed time window then some appropriate behaviour is taken, e.g. the frame is dropped, and a recursive call is made with the clock incremented by some amount. If the frame does not violate any timing constraints, then either it is displayed or appended to those already received, or possibly a combination of these.

We also include an event offer here that allows the current time of this process to be passed as a parameter to synchronise clocks for example. With these models of flows we can show how various features of the flows can be specified and checked. We focus on jitter, throughput and delays of flows. Extensions to the work are also possible, e.g. lossy communications and communications where latency issues have to be dealt with [7].

3.1. Maximum Jitter in LOTOS

Jitter may be regarded as the upper and lower limit on the time window at which a consumer can accept a frame. For example, if a frame is expected every t seconds with an allowed variation of δt, then a frame should arrive within the range $t - \delta t$ to $t + \delta t$. There are two cases of jitter that we consider here: bounded and unbounded jitter. The distinction between the two is dependent upon whether the arrival time of the last frame influences the arrival time of the next frame. In unbounded jitter, if frames are expected every t seconds with a variation of δt then should frames consistently arrive early, but within the allowed time range, then the flows will eventually drift out of synchronisation. For example if t was 30 time units say and δt was 5 time units and frames arrived every 29 time units, then after five frames had arrived, all subsequent frames would be outside the allowed range, i.e. the next frame would be expected at 180 but would arrive at 174 time units which would exceed the maximum unbounded jitter rate of 5.

In bounded jitter if a frame arrives early but within the allowed time variation, then the arrival time of the next frame is time t after that arrival time. Hence using the above numbers, if a frame arrives at time 29 then the next one would be expected at time 59 and not 60. From this, it can be seen that bounded jitter does not allow flows to drift out of synchronisation. Unbounded jitter was represented previously. This corresponded to the variable *limit* given in the process definition representing the consume action. It should be noted that with this model of production and consumption of flows, consideration of the time taken for consuming and producing single frames is critical. If both producer and consumer produce and consume frames respectively at the same rate, then ignoring issues of latency and lost frames, there should never be any jitter. It might be the case that producers and consumers operate at different speeds however. Hence the modifications to the time variable in the process instantiations are not necessarily equal. Bounded jitter models situations where the time difference between production and consumption of frames is slightly different but within certain limits. This may be represented as:

```
process Consumer[ g ](recFrames: FrameSeq, jitter, tnow: Nat):noexit:=
 g ?inf: Frame;
 (   [ Diff(getTime(inf),tnow) gt jitter ] → ...(* error behaviour, e.g. drop frame *)
     Consumer[ g ](recFrames,jitter,(tnow+t))
  [] [ (Diff(getTime(inf),tnow) le jitter) and ((getTime(inf) - tnow) gt 0) ] →
        ...(* successful behaviour, e.g. display frame *)
```

```
    Consumer [g ](addFrame(inf,recFrames),jitter,(tnow+t+Diff(getTime(inf)-tnow)))
  [] [ (Diff(getTime(inf) - tnow) le jitter) and ((getTime(inf) - tnow) le 0) ] →
        ...(* successful behaviour, e.g. display frame *)
    Consumer [g ](addFrame(inf,recFrames),jitter,(tnow+t-Diff(getTime(inf)-tnow))))
endproc (* Consumer *)
```

Here we state that if the maximum bounded jitter is exceeded then the frame is dropped and the local time incremented by *t* time units. If the frame arrives within the timing restrictions imposed by the jitter then two conditions arise depending on whether the frame arrives slightly early or late. If the frame arrives earlier than expected then the time variable *tnow* is modified by adding on the time for consumption and subtracting the amount the frame was early by. For example, assuming frames are expected every 30 time units and the current time is 60 if the next frame is time stamped 59 then the time variable *tnow* would be set to (60+30-1= 89). As a result, the next frame is expected at time 89. Alternatively should the frame arrive later than expected but within the allowed variation, e.g. at time 61 then the time variable is modified to (60+30+1= 91). Hence the next frame is expected at time 91.

3.2. Minimum Delay between Frames in LOTOS

The minimum delay between the frames a producer produces may be specified directly in LOTOS. This corresponds to the minimum difference between time stamps associated with two frames in the sequence of frames to be sent. If production of all frames is isochronous then the minimum delay is equal to the maximum delay and is constant. Typically, information flows can have their rate increased or decreased. The minimum delay between two frames is inversely proportional to the maximum throughput of the flow. The minimum delay for a producer may be represented by:

```
 process Producer[ g ](toSend: FrameSeq, tnow, maxRate: Nat):noexit:=
    g!setTime(tnow+maxRate,head(toSend));
    Producer[ g ](tail(toSend),(tnow+maxRate),maxRate)
endproc (* Producer *)
```

Here we assume the existence of some upper limit (*maxRate*) on the flow of frames. The minimum delay between frames for a consumer flow may be represented by:

```
 process Consumer[ g ](recFrames: FrameSeq, tnow, maxRate: Nat):noexit:=
   g ?inf: Frame;  ((* reject frame if outside timing constraints *)
   [] (* accept (display) frame and recurse *)
   Consumer[ g ](addFrame(inf,recFrames),(tnow+maxRate), maxRate))
endproc (* Consumer *)
```

4. Conclusions

In this paper we have shown that the LOTOS language can be used to model numerous timing issues that can apply to multimedia flows of information. We have largely avoided dealing with how these flows are established in the first place. The issues in establishing meaningful stream connections are discussed in more detail in [7].

5. References

[1] L. Blair, The Formal Specification and Verification of Distributed Multimedia Systems, Ph.D Thesis, University of Lancaster, England, June 1994.
[2] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 1, vol 6 EATCS Monographs on Theoretical Computing Science, Springer Verlag, 1985.
[3] E-LOTOS, Extended-LOTOS, ISO/IEC/WG7 Working Draft WI1.21.20.2.3.
[4] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.
[5] LOTOS, A Formal Technique Based on the Temporal Ordering of Observational Behaviour, ISO 8807, 1989.
[6] R. Milner, A Calculus of Communicating Systems, vol 92, Lecture Notes in Computing Science, 1980.
[7] R. Sinnott, An Architecture Based Approach to Specifying Distributed Systems with LOTOS and Z, Ph.D Thesis, University of Stirling, Scotland, July 1997.