Proceedings of the Second
IASTED/ISMM
International Conference

IASTED

ISMM

DISTRIBUTED MULTIMEDIA SYSTEMS AND APPLICATIONS

Stanford, California USA
August 7-9, 1995

Editors: B. Furht and M.H. Hamza

A Publication of
The International Association of Science and Technology
for Development - IASTED
and
The International Society for Mini and Microcomputers - ISMM

IASTED/ISMM - ACTA PRESS
ANAHEIM ◊ CALGARY ◊ ZURICH

ISBN: 0-88986-226-5

Addressing Real Time with Temporal Logic in Multimedia Communications

A. John M. Donaldson[£] and Plamen L. Simeonov[τ]

Abstract— **We present a strategy for specifying real time aspects of multimedia transfer by means of temporal logic. The method we have chosen allows us to establish common proof criteria for transmission synchronization and flow adjustment.**

Keywords— **Temporal Logic, Real Time, Multimedia, Behaviours, Actions, Liveness.**

I. Introduction

End-to-end synchronization among different flows is an important issue in multimedia communication. It can be implemented as uni- or bidirectional transfer, simultaneously or sequentially at one or more destination sites with one or more data streams converging at these sites while being bounded according to some common feature. A typical example of synchronization is the problem of timing the outputs of video and voice to achieve lip synchronization. Tricks such as temporal and spatial flow adjustment are often necessary to achieve reasonable presentation rates for multimedia.

Related work: Numerous works of protocol and service designer have been addressing the various aspects of real-timing and synchronization in distributed systems while trying to provide reasonable engineering solutions. For example, Escobar et al. [1] have developed a flow synchronization protocol based on synchronized network clocks to provide end-to-end synchronization and adapt to flow delay changes over arbitrary data transfer topologies.

On the other hand, theoretitians are developing methods and techniques to exactly specify and prove complex systems up to their real-time requirements by using formal means. Investigating the real-time expressivness of formal techniques such as extended LOTOS, SDL and real-time temporal logic, Hogrefe and Leue, conclude that temporal logic as *requirement oriented approach* is better suited to enhance verification, validation and testing during the service design phase, [2].

This paper is a contribution to how these two streams, the engineering and the formal one, can benefit from each other by approaching the subject with some common idea.

£ John Donaldson, Department of Computing Science, University of Stirling, Stirling, FK9 4LA, Scotland. Email: ajd@compsci.stirling.ac.uk

τ Plamen Simeonov, Technical University of Berlin, Sekr MA 073, Strasse des 17 Juni 136, D-10623 Berlin, Germany. Email: plamen@prz.tu-berlin.de

Our approach: There are two basic schools in multimedia design trying to solve the problem of flow synchronization: in the upper OSI layers (application and/or orchestration), and in the lower OSI layers (transport and/or network) respectively. We approach the problem of synchronization from the viewpoint of system design and verification by using formal description means and stressing on characteristics, such as buffering, flow control and message feedback mechanisms to derive appropriate specification proof rules that can be used with other conventional techniques to improve the design of distributed multimedia, [3].

II. The Problem

Let us consider a group of remote terminals capable of multi-media exchange and let us define the main requirement to the system as to enable a real-time data transfer between the terminals.

This requirement can be expressed by means of real-time temporal logic [4]. In our approach we use the Lamport's Temporal Logic of Actions (TLA) [5], an offspring of Pnueli's Temporal Logic ([6]), as a base for specificying time affected by the environment. *Real* time is refered in [7] in absolute sense by using the variable *now* to represent *this point in time* and then by relating that to the surrounding environment. This is a very powerful means of involving critical time constraints in system specifications with actions regarded in isolation from time advancing steps.

A. *Specifying Real Time with* TLA

Communication protocols and services providing multimedia in a network under time constraints can be ranked among real-time control systems. In the following we present a TLA view on the subject.

Actions and State Transitions: TLA is able to represent a finite state system model based on temporal logic. The specification consists of *steps* defining state changes as result of actions. The distinction between old and new states is achieved with the operator $'$ for the new state and $\Box$ to mean *henceforth*.

In TLA, the notation "*Unchanged s*" is used for $new\_state = old\_state$ or $(s' = s)$. Further, $[\mathcal{A}]_v$ denotes $\mathcal{A} \vee (Unchanged\ v)$ to assert a single legal step, i.e. that either an $\mathcal{A}-step$ occurs or that the implicated system states do not change. Then $\Box[\mathcal{A}]_v$

requires that every step of a behaviour is legal.

The succeeding actions in a behaviour are described by the composite action $\mathcal{N}ext$ which represents all relevant state changes. A predicate $Init$ indicates the starting state and $Init \wedge \Box[\mathcal{N}ext]_v$ asserts a behaviour that is *safe*[1].

Real Time: Associating real time with the execution of behaviours is described in [7].

- Actual time in any specification is represented by the variable *now*, the value of which is a *never* decreasing real number:
  $RT \ \triangleq\ (now \in \mathbf{R}) \wedge \Box[now' \in (now, \infty)]_{now}$
  where (r, ∞) is $\{t \in \mathbf{R} : t > r\}$.
- Time-advancing steps are distinct from ordinary program steps. By introducing the state function v, a tuple of all variables relevant to the system, advancing time is related to other system features:
  $$\begin{aligned} RT_v \ \triangleq\ & \wedge (now \in \mathbf{R}) \\ & \wedge \Box[(now' \in (now, \infty)) \ \wedge \ (v' = v)]_{now} \end{aligned}$$
  This is *relative real time* RT_v (regarding the state function v) which is defined to be THE MOMENT OF TIME IN WHICH CERTAIN DISCRETE SYSTEM VARIABLES REMAIN INVARIANT. In this way, we can define different *real times*[2] related to different sets of system states or different system components. Since RT_v is equivalent to $RT \wedge \Box[now' = now]_v$, then:
  $$\begin{aligned} Init \wedge \Box[\mathcal{A}]_v \wedge RT_v \ = \ & \wedge Init \\ & \wedge \Box[\mathcal{A} \wedge (now' = now)]_v \\ & \wedge RT \end{aligned}$$

This is the essence of the Lamport-Abadi approach [7]: to define time as a system intrinsic function constraining state changes. We use this concept as a base for developing system requirements for timing and synchronization.

Synchronization: The problem of synchronizing several data flows in a network can be formulated as setting up a number of timing requirements to be hold on a set of system parameters (buffer/data unit size, media scaling, flow control policy, etc.). The specification depends on how we describe the system requirements, i.e. where we expect to realize the synchronization: in the network, in the workstation, or in the application[3]. Thus, we can compare (and adjust!) diferent system descriptions regarding timing constraints and relate "real times" of the system components by converting them into state-function predicates and vice versa whenever appropriate.

Real Time Constraints: Lamport makes following assumptions:

- Safety property $Init \wedge \Box[\mathcal{N}ext]_v$ does not constrain time iff $now' = now + \delta$ is a subaction of it.
- Real-time constraints on the system F may be imposed using *timers* to restrict the increase of *now* using a state function t such that $F^t_\bullet \ \Rightarrow \ \Box(t \in \mathbf{R} \cup \{\pm\infty\})$ as
  - upper-bound timer (*now* never advances past $\mathbf{t}$):
  $$Most(t) \ \ \triangleq \ \ (now \leq t) \ \wedge \ \Box[now' \ \leq t' \ ]_{now}$$
  - lower-bound timer (an $\langle\mathcal{A}\rangle_v$ step cannot occur when *now* is less than t):
  $$Least(t, \mathcal{A}, v) \ \ \triangleq \ \ \Box[\mathcal{A} \Rightarrow (t \leq now)]_v$$

Often we have to assert that an $\mathcal{A}$ step must occur within δ seconds of when the action $\mathcal{A}$ becomes enabled, for some constant δ to indicate the availabilty of a particular step and make confident statements about what really has to happen.

B. Focusing on the Problem

Let us specify a service interface related to the buffering part of the network. Here the physical network consists of a transmission channel holding a *buffer* keeping the sequence of packets delivered for sending and a boolean-valued flag *toggle*. Data is submitted in the *buffer* and the process is regulated by complementing the *toggle* indicator. Input and output are represented by the pairs $(indata, toggle\_in)$ and $(outdata, toggle\_out)$ respectively where $toggle\_in$ and $toggle\_out$ are Boolean flags for input/output control and *indata* and *outdata* are the values of the input packet and the transmitted (output) packet. Transmission is purely local to the sender with inputs being lost if they arrive faster than the buffer can handle them[4]. The sequence of output messages is a subsequence of the sequence of input messages.

C. Temporal Specification

Let us start with the untimed system specification[5]. Predicate $Init_B$ describes the initial states of the variables in the queing buffer B:

$$\begin{aligned} Init^t_B \ \ \triangleq \ \ & \wedge \ toggle\_in, toggle\_out \in \{\mathsf{true}, \mathsf{false}\} \\ & \wedge \ indata, outdata \in \mathsf{Pkts} \\ & \wedge \ toggle = toggle\_in \\ & \wedge \ buffer = \langle\,\rangle \end{aligned}$$

Action $Next_B$ describes allowed changes to the system variables as disjunction of the actions *Put*, *Store*, and *Send*.

$$Next_B \ \ \triangleq \ \ Put \ \vee \ Store \ \vee \ Send$$

[1] The issue of *liveness* to ensure that behaviours proceed with time is handled later in the text.

[2] no matter whether continuous or discrete time is applied !

[3] So we naturally have more than one solution. The formal proof of such timing constraints for the corresponding system model can be done by applying TLA formulae in a straightforward manner.

[4] Later we can add timing constraints to rule out the possibility of lost messages.

[5] There is a distinct advantage of this approach, because we are able to add in timing after we have established the description of the various actions to take place there

Functional Components: TLA allows to specify a series of actions in a module associated with the functioning of a particular system component. For example, action *Put* establishes the necessary state-changes for placing packets into the buffer. It is always *enabled*, i.e. in any state, new input may be sent:

$$PutVars \triangleq (toggle\_out, outdata, buffer, toggle)$$

$$\begin{aligned} Put \triangleq\ & \wedge\ toggle\_in' = \neg toggle\_in \\ & \wedge\ indata' \in \mathsf{Pkts} \\ & \wedge\ PutVars' = PutVars \end{aligned}$$

In a similar way are constructed the actions *Store* and *Send* for getting packets into the buffer and transmitting the tokens to the outer network.

Temporal Formula: $Spec_B$ represents the internal specification of the sending procedure and describes all sequences of states that may be instantiated [6]:

$$Spec_B \triangleq Init_B \wedge \Box[Next_B]_v$$

$Init_B$ asserts that the initial state is correct; $\Box[Next_B]_v$, asserts that every step is either a legal $Next_B$ step or leaves v unchanged.

Correctness: TLA is an untyped logic. However, to ensure the consistency of declarations, it is possible to define alternative functions to verify them. $Spec_B$ is correctly specified, if $Spec_B \Rightarrow \Box Check_B$, where $Check$ is the predicate asserting the nature of all variables declared.

D. Adding in Timing

Let us now add timing to the actions of the previous section. Data packets are added to and transmited from the buffer at most once every δ_{snd} seconds with lower-bound timers t_{Put} and t_{Send} used for these operations. An incoming packet must be added to the buffer within Δ_{str} seconds of appearing at its interface. The "upper limit" timer T_{Store} is used for this purpose. Similarly, timer T_{Send} restricts the delivery of data packets to the output buffer interface within Δ_{snd} seconds after the *unbuffering* action *Send* becomes enabled. Let us redefine the actions discussed above with timing added in. For example, *Put* becomes Put^t:

$$\begin{aligned} Put^t \triangleq\ & \wedge\ Put \\ & \wedge\ t_{Put} \leq now \\ & \wedge\ t'_{Put} = now' + \delta_{snd} \\ & \wedge\ \textbf{if } data\_in \textbf{ then } T'_{Store} = now' + \Delta_{str} \\ & \wedge\ \textbf{if } \neg data\_in \textbf{ then } T'_{Store} = \infty \\ & \wedge\ \textbf{if } ebuf \textbf{ then } (t_{Send}, T_{Send})' = (\infty, \infty) \\ & \wedge\ \textbf{if } \neg ebuf \textbf{ then } Unchanged(t_{Send}, T_{Send}) \\ & \wedge\ now' = now \end{aligned}$$

where

if $toggle' \neq toggle\_in'$ **then** $data\_in =$ "true"

if $buf = \langle\rangle$ **then** $ebuf =$ "true"

The description of the process is further developed by providing a representation for the advance of time in relation to the other specified actions. Action *TF* is defined to meet the variable *now* in establishing the exact moments of time we are referring to [7]:

$$\begin{aligned} TF \triangleq\ & \wedge\ now' \in (now, \min(T_{Send}, T_{Store})] \\ & \wedge\ vt' = vt \end{aligned}$$

where $vt \triangleq (v, t_{Put}, t_{Send}, T_{Send}, T_{Store})$.

The timed system description is contained in the module *TimedTransmission* shown below. Each constituent action $\mathcal{A}$ of formula $Spec_B$ specified in the original module has a corresponding[8] timed version $\mathcal{A}^t$ with the time advancing action *TF* added in to allow for the advance of variable *now* with time. Formula $Spec^t_B$ satisfies each maximum-delay constraint by preventing *now* from advancing before the constraint has been satisfied. Such TLA formulas describe *what* is supposed to happen, not how it is to be achieved. $Spec^t_B$ says only that an action occurs before *now* reaches a certain value, and this way of thinking is different from conventional methods where *now* is changed by the system.

module *TimedTransmission*

import *Transmission, BufferActions, RealTime*

predicates

$$vt \triangleq (v, t_{Put}, t_{Send}, T_{Send}, T_{Store})$$

$$wt \triangleq (vt, now)$$

$$\begin{aligned} Init^t_B \triangleq\ & Init_B \wedge now \in \mathbf{R} \\ & \wedge\ t_{Put} = now + \delta_{snd} \\ & \wedge\ t_{Send} = T_{Store} = T_{Send} = \infty \end{aligned}$$

actions

$$Next^t_B \triangleq Put^t \vee Store^t \vee Send^t \vee TF$$

temporal

$$Spec^t_B \triangleq Init^t_B \wedge \Box[Next^t_B]_{wt}$$

E. Liveness

Formula $Spec^t_B$ describes a *safety* property, meaning that it is satisfied by an infinite behavior which starts in a correct initial state, and in which every successive state is correct [9]. In practice, *safety* asserts what may not happen. However, in asynchronous communications we have to ensure that events will eventually happen. This is achieved through the property of **liveness** [8] in the form of **fairness** conditions. While a

[6] State Predicate v is the state function tuple of all relevant system variables.

[7] Note: $(r, s]$ denotes the set u such that $r < u \leq s$.

[8] This is perfectly "legal" as we know that the logic of the equivalence of $(\Box F) \wedge (\Box G)$ and $\Box(F \wedge G)$[5] may be used to verify the actions in $Spec_B$ when they are conjoined with appropriate temporal *timing* formulas.

[9] Note: TLA recognises the concept of *stuttering* which allows a safe state to remain so forever.

safety condition may assert e.g. that a data packet is not sent until it is put on the buffer, a liveness condition may assert that if a data packet is put on the buffer, it will be eventually sent [5]. Thus we complement $Spec_B^t$ to a satisfactory buffering/transmission specification by conjoining the *strong fairness* property $SF_{vt}(Store)$ and the *weak fairness* property $WF_{vt}(Send)$ to it:

$$Spec_B^t \triangleq \begin{array}{l} \land\ Init_B^t \\ \land\ \Box[\mathcal{N}ext_B^t]_{vt} \\ \land\ SF_{vt}(Store) \\ \land\ WF_{vt}(Send) \end{array}$$

- Property $SF_{vt}(Store)$ asserts that if action *Store* is enabled infinitely often, then infinitely many *Store* steps must occur. It implies that if infinitely many inputs are sent, then the buffer must receive infinitely many of them.
- Property $WF_{vt}(Send)$ asserts that if action *Send* is enabled forever, then infinitely many *Send* steps must occur. This property implies that every buffered packet is eventually output.

III. Multicasting and Multiple Users

So far, buffering has been considered as "local" procedure, handling data on the sender's side in such a way that time is "lost" internally for each buffer as it handles each connection, providing it with replicated data for the transfer[10]. Synchronization is one aspect of timeliness, and buffering policy is a particular aspect of synchronization. Let us now consider what happens in the network.

A. Real Time Aspects in Multimedia Transmission

An isochronous application such as remote videoconferencing requires a bounded end-to-end delay while avoiding any observable jittering effect. However, the data packets of several synchronised connectioins may have different delay variations before being integrated and synchronised at the receiver side[11]. When talking about "real time aspects" in multimedia communication, we are addressing the whole set of techniques provided to make the user believe that she is actually/virtually "now and face-to-face" involved in a talk with her remote partner.

The problem may be solved in several different ways:

- synchronization error recovery, e.g. [9] introducing a set of QoS values to determine the mismatching tollerance time;
- flow synchronization, e.g. [1] using synchronized network clocks and fixed end-to-end delay enforcement;
- synchronization transport mechanisms, e.g. [10] using groups of related multimedia connections with a common base for calculating rate control parameters.

All three methods can be modelled with TLA, and here we concentrate on the last one.

B. Synchronisation using Rate Control

Rate control is a prefered technique for enforcing policies, performance optimisation and ressource usage in multimedia applications [11]. The flow control mechanism of the XTPX protocol [12] is based on:

1. Rate Control Timer (RCT)
2. BURST – the maximum number of bytes allowed to be sent in an RCT interval (RCTI)
3. RATE – the maximum number of bytes sent per second: $RATE = BURST/RCTI$

The XTPX rate control techniques may be influenced by application layer framing and the nature of the media. The protocol uses feedbacks of the receiver (CNTL response PDUs), the network and the local application to (re)negotiate the sending rate of the connections. Thus, multimedia traffic can be shaped by synchronizing connections to different or same desinations using *bundles* to assign appropriate rate control parameters for each connection of the bundle (e.g. to independent media sources such as audio and video).

Let us consider *interval* dT_i be a tuple $\langle t_i, t_{i+1} \rangle$ describing a timer interval with starting moment t_i and ending moment $t_i + 1$ for the RCT to process the sum of the bursts of connections included in a bundle: $dT_i = t_{i+1} - t_i$. Let d_i^{iso} and d_i^{asyn} be the duration of the burst of an isochronous and a synchronous media connection within dT_i respectively, and let both connections be coupled in a bundle. The timing transfer requirement is then $d_i^{iso} + d_i^{asyn} \leq dT_i$ for each i in an RCTI sequence. Ergo, to "squeeze" the asynchronous media traffic within the bundle means that: $d_i^{asyn} \leq dT_i - d_i^{iso}$, i.e. the bursts of the asynchronous media traffic must be transmited in the intervals d_i^{asyn} fulfilling the above inequation [12].

Finding an optimal RCT interval to provide for timely delivery of the different media packets is the goal of bundled rate control synchronization. By introducing statistical values for the timer interval in the system specification related to the desired throughput and burst ranges, it is possible to check the fullfilment of timing constraints and to evaluate the scalability of the service with respect to synchronous media transfer.

Relating Real Time to its Environment: We proceed as follows. First, we tune dT_i to fix a constant $BURST$ for a specific media unit [13]. Then, large scale variations and fine tuning of dT_i are intended to sample the whole range of bursts typical for that kind of media transfer. The minimal interval dT_i^{min} allowing a

[10] The specification provides the data to only "one" alias/group address for multicast and we have assumed that the network does the rest of the job while reducing the buffers to one.

[11] This can be due to the heterogenious nature of the networks, terminal equipment and operating systems which shape the traffic as a result of buffering policies and processing power.

[12] The same condition applies if we try to couple more than two synchronous and asynchronous connections in a bundle.

[13] e.g. some image frame size of a motion JPEG video. [13]

desired constant average $BURST$ can be obtained independently for each media type, e.g. $dT_i^{min}(vid)$ and $dT_i^{min}(aud)$ for video and audio bursts respectively.

If we now return to the subject of fairness and use the strong and weak fairness arguments, we can strengthen the specification and give the very practical assertion of *hard and soft synchronisation* requirements [14].

First, we revise the specification of $Next_B^t$ to consider the whole multimedia transmission process, e.g. for *aud*io and *vid*eo data streams:

$$\begin{aligned} Next_B^t \;\triangleq\; & \vee\ (Put_{aud}^t \vee Store_{aud}^t \vee Send_{aud}^t \vee TF) \\ & \vee\ (Put_{vid}^t \vee Store_{vid}^t \vee Send_{vid}^t \vee TF) \end{aligned}$$

Then, our specification covering the advance of time is augmented to cater for synchronisation, so that $dT_i = min(dT_i^{min}(vid), dT_i^{min}(aud))$ and

$$\begin{aligned} TF \;\triangleq\; & \wedge\ now' \in (now, \min(T_{Send}, T_{Store}, dT_i)] \\ & \wedge\ vt' = vt \end{aligned}$$

Finally, we define:

- **Hard synchronization** for multimedia means an exact temporal relation between any two information units, i.e. there must be distinct time intervals between the data units transfered[14]. One way to state this is e.g.:

  $$\begin{aligned} Fair_{vid} &\triangleq (SF_{vt}(Store_{vid}) \wedge WF_{vt}(Send_{vid})) \\ Fair_{aud} &\triangleq (SF_{vt}(Store_{aud}) \wedge WF_{vt}(Send_{aud})) \end{aligned}$$

  $$\begin{aligned} HdSync_B^t \;\triangleq\; & \wedge\ Init_B^t \wedge \Box[Next_B^t]_{vt} \\ & \wedge \vee\ Fair_{vid} \\ & \quad \vee\ Fair_{aud} \end{aligned}$$

- **Soft synchronization** occurs when certain grade of tollerance (time interval) is allowed for the representation of the different media types in addition to the fixed synchronization point. This is an application and information contents dependent feature of isochronous media where the quality or grade of acceptance is empirically derived on a statistical base. A variant of this is:

  $$\begin{aligned} Fair_{Store} &\triangleq (SF_{vt}(Store_{vid}) \vee WF_{vt}(Store_{aud})) \\ Fair_{Send} &\triangleq (SF_{vt}(Send_{vid}) \vee WF_{vt}(Send_{aud})) \end{aligned}$$

  $$\begin{aligned} WkSync_B^t \;\triangleq\; & \wedge\ Init_B^t \wedge \Box[Next_B^t]_{vt} \\ & \wedge \wedge\ Fair_{Store} \\ & \quad \wedge\ Fair_{Send} \end{aligned}$$

Both kinds of synchronization can be also used in combination to express different timing configurations between the media types. Besides, by extending the temporal operators $\Box$ (*always*) and $\Diamond$ (*eventualy*) with timing constraints to hold, each fairness requirement can be additionally strengthened. This technique can be applied to specify and verify any synchronization mechanism as far as the values of the corresponding system variables are known or predictable.

[14] An example for this is the randezvous concept in CSP, [15]. In the above case of two data streams we can assert that the *Store* and *Send* processes for each particular data stream at the initiator terminal must be ready to perform at any time that the prevailing system conditions permit.

IV. Conclusions

We have presented a formal approach to specifying real time for synchronization of data flows in multimedia communication. The main contribution of this paper is the application of temporal logic to service modelling and verification with respect to real time treatment, and in particular with the consideration of statistical values for rate control in requirements for "hard" and "soft" synchronization.

This method is continuation of our previous work in multimedia service design [3] and is intended to be integrated in a common multimedia service design platform with other formal and measurement tools [16].

References

[1] J. Escobar, J. C. Partridge, and D. Deutsch. Flow synchronisation protocol. *IEEE/ACM Trans. on Networking*. 2(2):111–121, Apr. 1994.

[2] D. Hogrefe and S. Leue. Specifying real-time requirements for communication protocols. Tech. Report IAM 92-015, University of Berne, Inst. for Informatics and Applied Mathematics, Switzerland, Jul 1992.

[3] I. Miloucheva, P. L. Simeonov, K. Rebensburg, K. J. Turner, and A. J. M. Donaldson. Prototype performance evaluation of multimedia service components. In *Proc. 3d Int. Conf. on Computer Communications and Networks (ICCCN'94)*, San Francisco, USA, September 11-14 1994. IEEE/UCS-CSI, Elsevier.

[4] T. A. Henzinger. The temporal specification and verification of real–time systems. Report STAN–CS–91–1380. Stanford University, Dept. of Computer Science, Aug 1991.

[5] L. Lamport. The temporal logic of actions. Res. Report 79, DEC, Sys. Res. Center, Dec. 1991.

[6] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[7] M. Abadi and L. Lamport. An old-fashioned recipe for real time. Res. Report 91, DEC, Sys. Res. Centre, 1992.

[8] B. Alpern and B. A. Schneider. Recognizing safety and liveness. *Distributed Computing*. (2):117–126, 1987.

[9] L. Lamont and N. D. Georganas. Synchronization architecture and protocols for a multimedia news service application. In *Proc. IEEE Int. Conf. on Multimedia Computing and Systems*, May 1994.

[10] I. Miloucheva and O. Bonnesz. Xtpx transport system for flexible qos support of multimedia applications. In *Proc. IPCCC/IEEE Conf.*, Phoenix, Arizona, USA, 1995.

[11] C. A. Eldridge. Rate controls in standard transport layer protocols. In *Proc. ACM SIGCOMM'92: Comm. Architectures, Protocols and Applications*, Baltimore, USA, Jul 1992. ACM.

[12] B. Metzler and I. Miloucheva. Multimedia communication platform: Specification of the broadband transport protocol xtpx. RACE 2060 CIO Deliverable 60/TUB/CIO/DS/A/002/b3. TU Berlin, FSP-PV, 1993.

[13] P. L. Simeonov and Lutz U. A method for qos verification by measurement for the jvtos video transfer. In G. von Bochmann, J. de Meer, and A. Vogel, editors, *Proc. Workshop on Distributed Multimedia Applications and QoS Verification*, Montreal, Canada, June 1994. CRIM.

[14] Ralf Steinmetz. *Multimedia-Technologie*. Springer, 1993.

[15] C.A.R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[16] T. Wendlinger, P. L. Simeonov, and L. Ulrich. Toolset for protocol and advanced service verification in ibc environments: Experience report. RACE 2088 TOPIC Deliverable TOPIC/WA3/CLE/TEE/DS/P/020/b2, TU Berlin, FSP-PV, Dec. 1994.