Script-Based QOS Specifications for Multimedia Presentations*

Richard Staehli, Jonathan Walpole
{*staehli, walpole*}*@cse.ogi.edu*

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Rd., PO Box 91000
Portland, OR 97291-1000

ABSTRACT

Multimedia presentations can convey information not only by the sequence of events but by their timing. The correctness of such presentations thus depends on the timing of events as well as their sequence and content. This paper introduces a formal specification language for playback of real-time presentations. The main contribution of this language is a quality of service (QOS) specification that relaxes resolution and synchronization requirements for playback. Our definitions give a precise meaning to the correctness of a presentation. This specification language will form the basis for a QOS interface for reservation of operating system resources.

Keywords: Resource Reservations, Real Time Multimedia Authoring, Operating Systems, Synchronization

1 Introduction

Multimedia systems typically support both static and dynamic media types. The static types include text and graphics that the viewers peruse at their leisure. Dynamic types such as video and animations present information that changes with time, and in fact the rate of change is a part of the information that is being communicated. In order to communicate effectively, a system that supports dynamic media types must be able to preserve the meaningful temporal relationships in a multimedia presentation. Digital audio and video are known as *continuous media* because they approximate an analog signal. Such media have *natural synchronization* constraints for playback that arise from the sample recording rate. A *multimedia presentation* may also include *synthetic synchronization* constraints, specified by the presentation's author, to create a meaningful relationship between media objects. Synthetic constraints can be specified in most multimedia authoring tools [10, 5, 7, 4] both through explicit synchronization between objects and through rate control of continuous media. Playback quality depends on how close the presentation events are to the specification. In these systems, the playback quality of service (QOS) depends on scheduling mechanisms and the availability of operating system resources that are not easily understood by a user.

In our architectural model, shown in Figure 1, an author uses a presentation specification tool, which we refer to as a scripting tool, to define a presentation. A user views a presentation via a playback tool that may be separate from the scripting tools. The playback tool may execute concurrently with other unrelated applications in a general purpose computing system. While the

*This research is supported by NSF Grant IRI-9117008 and by funds from Tektronix, Inc. and the Oregon Advanced Computing Institute.

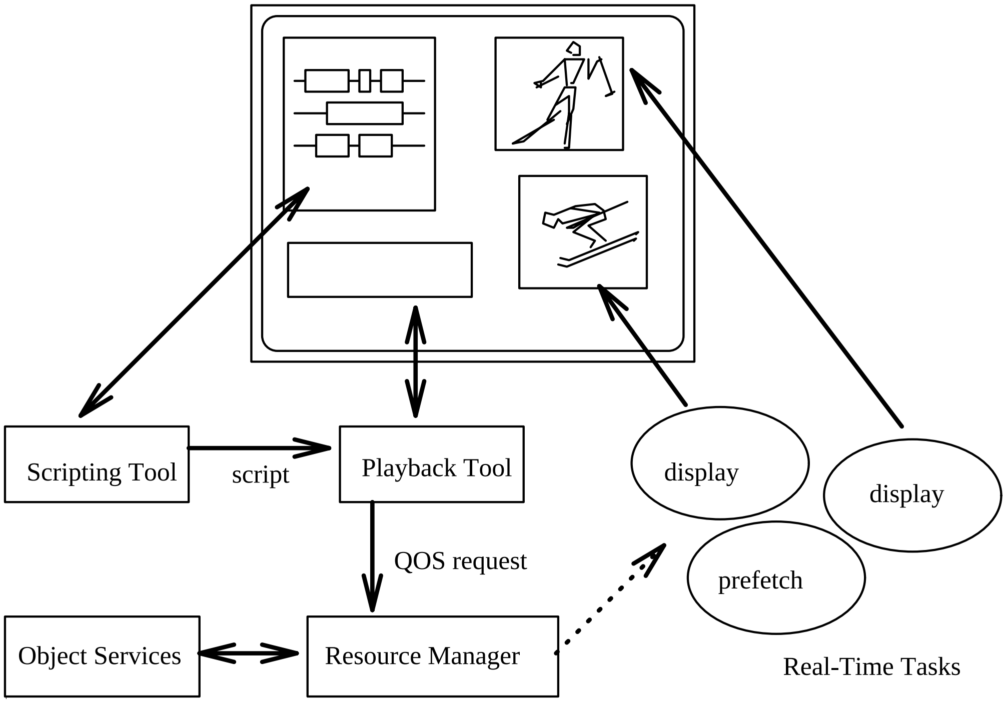

Figure 1: Specification of content and playback quality in multimedia presentations.

playback tool is responsible for the correct timing of the presentation it must rely on the underlying operating system for timely access to resources. To limit the scope of the specification problem this paper assumes that presentations are not interactive.

Presentation timing depends on the playback algorithm, the clock that provides time values, the availability of required resources and the latency of presentation operations. Assuming that the playback algorithm is correct, timing errors can arise from each of the latter three sources independently. For example, if the rate of audio playback is controlled exclusively by a clock on the output device then the audio may drift out of synch with a video stream that is controlled by a different clock. A display process can miss deadlines while waiting in a ready queue while another process executes on the CPU. A display process may have all needed resources before a deadline, but still miss the deadline because of the inherent processing time required. A playback algorithm can be designed to anticipate timing errors and reduce their impact on the remainder of the presentation. An operating system can make this task easier by providing more information about resource availability, system latencies and clock rates.

When resource loads make it impossible to satisfy all constraints in a presentation, a multimedia system can either preserve the data content at the expense of timing or it can shed some of the data processing load in order to meet more timing constraints. Some playback tools[5, 4] allow presentation events to occur late, as scheduled by a non-real-time operating system. The alternative is a real-time scheduling policy that chooses which deadlines to meet and and which tasks to defer or drop. If a system does not provide resource reservations, then overload conditions will cause service to degrade during multimedia presentations. Some playback tools incorporate algorithms that attempt to preserve QOS for audio streams while video streams that can tolerate a greater loss in bandwidth are allowed to miss frames.[8] As load increases, however, sharing a degradation of service eventually leads to effective service to none! Resource reservations can be used as an alternative to provide service guarantees for at least some users. A system that provides service guarantees must also have an acceptance test that checks for availability of resources before providing guarantees to any user.

In other words, this approach avoids overloads by occasionally denying service to new users, rather than degrading service for existing ones. To provide service guarantees, a service provider must know which resources are needed and how much to reserve of each.

Recent research into protocols for network bandwidth reservation has yielded a number of proposals for QOS specifications consisting of parameters such as max-message-size, max-msg-frequency, average-message-rate, average-error-rate and averaging-interval.[12, 2] For multimedia presentations, a reservation protocol must be able to ensure synchronized delivery of data from a number of sources. Bandwidth reservations alone are insufficient because they do not provide an upper bound on start-up latency. That is, the first unit of data on a guaranteed bandwidth connection might arrive too late relative to data already presented from other connections. For simple presentations it might be sensible to delay the start until connections have been set up for all media components, but in general it is necessary to create and release connections during a presentation in order to use resources efficiently.[14] In this paper, we define a QOS specification technique for multimedia presentations that is based on timing constraints for individual presentation events. We argue that bandwidth requirements for data channels are an implementation concern and may be derived from the output timing constraints and a playback algorithm. The combination of output timing and content constraints and a specification of acceptable timing and content errors define the QOS needed for a presentation.

In multimedia systems, acceptable QOS depends both on the presentation author's content specification and the user's preferences for playback. Although no multimedia system can achieve the synchronization goals of a presentation perfectly, most playback tools do not allow one to specify tolerances for errors. In other words, they lack a formal semantics for imperfect execution. If a multimedia system is to service a maximum number of playback requests without allocating excessive and costly resources, then a QOS interface is needed to enable the playback tools to specify their requirements more precisely.

This paper describes Timesync: a language that specifies synchronization for all multimedia presentation actions relative to a single clock. Real-time specification languages[3, 1, 13] provide a formal basis for proving correctness in real-time systems, but these languages do not dictate how one is to specify tolerances for errors. The Timesync language shows one way to accomplish this, starting with a real-time script and adding constraints on the number and type of exceptions to be allowed. Timesync has a formal interpretation for its specification of timing constraints and error tolerances. Such a specification can be used by an operating system to understand service requirements and, with adequate support for resource reservations, provide presentation service guarantees.

The next section outlines the requirements that motivate the design of the Timesync language followed by a discussion of related work in Section 3. Section 4 reviews the CSP notation that is used in section 5 to define our language. In section 6 we assess the strengths and weaknesses of the Timesync approach for specifying multimedia presentations. Our conclusions and a discussion of future work is given in section 7.

2 Requirements for a Presentation Specification Language

A specification language for non-interactive multimedia presentations must be able to identify the presentation contents as well as describe the synchronization and spatial layout of the contents on the output devices. In addition to these basic requirements, we feel that a presentation language should also attempt to meet the following goals:

- Physical data independence. A specification should have the same meaning across implementations of data sources.
- Recursive composition of presentations. Arbitrarily complex presentations can be specified by composing simpler specifications.
- Formal semantics for error tolerances. It must be possible to detect when a presentation fails to deliver its meaningful timing and contents.

- Completeness in expressibility. It should be possible to specify all meaningful synchronization and layout of media objects.
- Soundness in constraint specifications. It should not be possible to specify conflicting constraints in a presentation.
- Simplicity. Common presentation types should be easy and compact to express. Unusual presentation requirements should not be unnecessarily difficult.

3 Related Work

There are a large number of scripting tools for multimedia presentations, both in commercial and experimental systems. QuickTime[11], the well known commercial product from Apple, assigns a playback time for each display action relative to a single clock. By translating each playback time t to the clock interval (t', t'), where t' is the closest integral clock value closest to t, we can interpret QuickTime movies directly as scripts with Timesync's formal semantics. There is, however, no existing standard for specifying playback quality for QuickTime movies. Instead, the typical playback algorithms perform a best effort scheduling of playback actions. In particular, an unbounded number of video frames may be dropped when the video data path has insufficient bandwidth so that playback is frequently unacceptable on an overloaded system.

The MAEstro system[5] provides a graphical timeline editor that allows easy specification of synchronization between media objects that are represented as segments of various tracks within the timeline. Again, the timeline editor specifies synchronization goals, not playback requirements. In fact, the playback of a MAEstro composition is accomplished by best-effort dispatching of play commands to separate (possibly remote) applications that support the playback of individual media types. Naturally, on a Unix platform, a number of resource limitations may cause unpredictable delays in the playback.

Little and Ghafoor have described an interval-based approach to specifying synchronization of multimedia elements.[10] Their approach provided some of the inspiration for the recursive composition of complex presentations described in this paper. However, their work still does not address the specification of playback quality. An algorithm is given for playback that assumes ample system resources are available to meet presentation deadlines.

4 A Brief Review of CSP

Our process specifications will use the definitions and notation developed by C.A.R. Hoare for his Communicating Sequential Processes (CSP) [6]. We assume that the reader is familiar with this work and that this section need only provide a brief review of terminology.

A *process* describes the behavior pattern of an object in terms of a finite set of *events* from some alphabet. For example, we could describe the order of *push* and *pop* events in a stack using the alphabet $\{push, pop\}$:

$$\begin{aligned} STACK &= P_0 \\ P_0 &= (push \rightarrow P_1) \\ P_{n+1} &= (push \rightarrow P_{n+2} \| pop \rightarrow P_n) \end{aligned}$$

A trace of a process execution is a finite sequence of events observed by the process in the order that they occurred. Simultaneous events may be recorded in any order since there is no implication of time elapsing between events. A process description constrains the order in which events occur and $traces(P)$ denotes the set of all possible traces of a process P.

Specifications are predicates on a trace that constrain the allowable sequences of events within it. We use the following notation to talk about properties of a trace:

$\#t$ the number of events in a trace

$t[i]$ is the ith event in trace t

A specification requiring that the number of pop events never exceed the number of push events in a trace tr is written:

$$\#pop \textbf{ in } tr \leq \#push \textbf{ in } tr$$

We say that a process P satisfies a specification S if, for every possible trace tr of P, the predicate S holds. We abbreviate this relation to

$$P \textbf{ sat } S$$

5 Script-Based QOS Specification

To give our language a formal basis, we first define the low-level timing constraints based on observable presentation and clock events. Later, we will introduce high level constructs and composition operators that will make it easier to express common presentation components such as synchronized audio and video segments.

5.1 Time.

A clock produces a monotonically increasing sequence of integer time values. A real-time clock advances independently of an observer process but may communicate its value through read operations. In general, a read operation may incur some delay so that the value of the clock read may be somewhat smaller than the clock's current value. We model such a real-time clock with the following CSP process:

$$\begin{aligned} RTCLOCK &= P_0 || READ \\ P_n &= (tick \rightarrow P_{n+1} || t.n \rightarrow P_n) \\ READ &= (bef.t.n \rightarrow t.n \rightarrow aft.t.n \rightarrow READ) \end{aligned}$$

where $tick$ increments the clock value and $bef.t.n, t.n$ and $aft.t.n$ correspond to the initiation of a read, assignment of the clock value n and completion of a read respectively.

A clock interval is a pair of time values (i, j) where $i \leq j$. An interval (i, j) contains another interval (k, l) iff $i \leq k \wedge l \leq j$. For containment, we use the notation for a subset so that for two intervals, I and I', $I' \subset I$ means that I contains I'. Addition of an integer to an interval is defined so that $k + (i, j) \equiv (i + k, j + k)$.

5.2 Actions.

While a CSP event is the instantaneous recording of an observation in the trace of a process, multimedia presentation *actions* such as the transfer of a frame of video data to an output device, may have non-zero duration and significantly overlap other presentation actions. An action produces an observable state change that can be delimited in time by a pair of events. For example, a function that displays a video frame is delimited by call and return events. Since a real-time specification requires a total ordering of a constrained event with observations of the constraining time values, we require that $before$ and $after$ events in the observing process causally precede and follow respectively the initiation and completion of an action. For an action a we will let $bef.a$ and $aft.a$ denote these events.

We would like to specify when significant actions occur in a process with respect to a (possibly remote) real-time clock. As before, let $t.i$ denote the action of reading a clock with i being the value returned. An action a is said to occur during a clock interval $(i, j + 1)$ iff a sequential process reads the clock value i before the action and the value j after completion of the action. That is, the trace of the process observes the event sequence $< aft.t.i \ldots bef.a \ldots aft.a \ldots bef.t.j >$. Figure 2 illustrates this synchronization with a minimum number of communications. Note that even if the observer reads the same clock value i before and after the action, the clock interval during which the event occurred, $(i, i + 1)$, is non-zero.

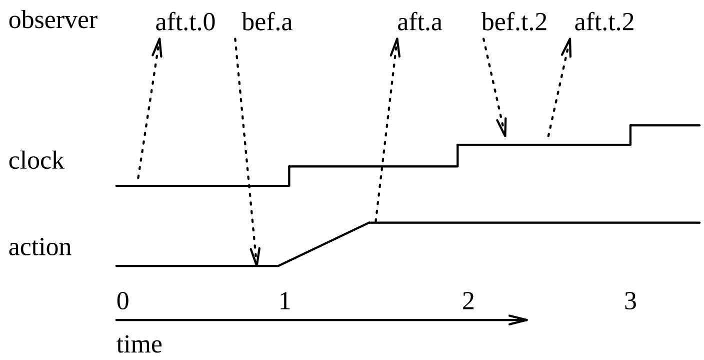

Figure 2: From the event sequence $< aft.t.0, bef.a, aft.a, bef.t.2 >$ the observer concludes that that action a 0 in the interval $(0, 3)$.

5.3 Scripts.

A script is a mapping of actions to clock intervals. We write a script as a set of pairs (a, I) where a is the description of an action and I is a clock interval. The lowest time value in a script is defined to be the start time. We define the addition of an integer to a script as:

$$i + S \equiv \{(a, i + I) | (a, I) \in S\}$$

Logically a script is a real-time process specification with a timing constraint for each action. In order to formalize this meaning for a script, we need to develop some machinery to help us relate the pattern of events in the trace of a process to the constraints in a script. The following three conditions allow us to define a unique interval for the occurrence of every action.

To interpret clock readings as times we must require that they are monotonically increasing as mentioned previously:

$$CLOCK \equiv \forall i, j, k, l : (i < j \wedge tr[i] = aft.t.k \wedge tr[j] = bef.t.l) \Rightarrow k \leq l$$

Requiring the first and last events in a trace to give us a bounding clock interval for the rest of the events ensures that there exists at least one clock interval for every action:

$$BOUNDED \equiv \exists i, j : tr[1] = aft.t.i \wedge tr[\#tr] = bef.t.j$$

Finally, if the same action a occurs more than once, that is, we observe the sequence of events $< bef.a \ldots aft.a \ldots bef.a \ldots aft.a >$, then we must require that there be a reading of the clock between the occurrences so that we can distinguish them:

$$\begin{aligned} DISTINGUISH \equiv\ & \forall i < j : (s[i] = aft.a \wedge s[j] = bef.a) \Rightarrow \\ & (\exists k, l, m : i < k < l < j \wedge s[k] = bef.t.m \wedge s[l] = aft.t.m) \end{aligned}$$

We say that a process is *2* if all of its possible traces satisfy these requirements:

$$P \textbf{ sat } CLOCK \wedge BOUNDED \wedge DISTINGUISH$$

In the rest of this paper, we will assume that all processes are *2* so that our timing specifications have their intuitive meaning.

Let $\textbf{obs}(tr)$ be the set of all traces that observe some subset of the events in tr in the same order as they occur in tr:

$$\textbf{obs}(tr) \equiv \{s | \forall i \in \{1 \ldots \#s\} \exists k : s[i] = tr[k] \wedge \forall j \in \{i + 1 \ldots \#s\} \exists l : k < l \wedge s[j] = tr[l]\}$$

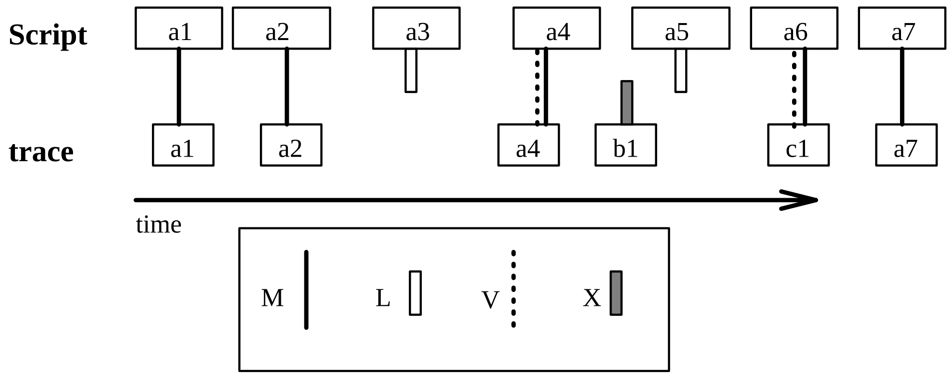

Figure 3: Example mapping of actions in a trace to the constraints in a script.

For any trace of a *real-time* process, the following definition gives us the set of most tightly constrained time intervals for each occurrence of an action in the trace:

$$\begin{aligned}\mathbf{tim}(tr) \quad \equiv \quad & \{(a,(i,j)) | \langle aft.t.i, bef.a, aft.a, bef.t.j\rangle \in \mathbf{obs}(tr) \\ & \wedge \neg \exists k,l : \langle aft.t.k, bef.a, aft.a, bef.t.l\rangle \in \mathbf{obs}(tr) \wedge (k > i \vee l < j)\}\end{aligned}$$

Let $R121$ be the set of all one-to-one relations. Formally, we interpret the meaning of a script S, abbreviated $\mathcal{M}(S)$, as a logical formula with free variable for a trace of a process tr:

$$\mathcal{M}(S) \equiv \exists M \in R121 : dom(M) = S \wedge ran(M) = \mathbf{tim}(tr) \wedge \forall ((a,I),(a',I')) \in M : a = a' \wedge I' \subset I$$

In other words, the constraints of the script should each be satisfied when mapped one-to-one onto the timings of a trace. We say that a *2* process P satisfies a script S if, for every possible trace tr of P, the constraints in $\mathcal{M}(S)$ hold. We abbreviate this relation to

$$P \ \mathbf{sat} \ \mathcal{M}(S)$$

5.4 Quality of Service Specification.

If the goal of a process is to satisfy a script, then an execution whose trace fails to satisfy the script is of lower quality than one that does. A quality of service (QOS) specification may relax the constraints on satisfying a script by telling what type of exceptions and how many of each can be tolerated. Our definition for **sat** allows us to group exceptions in three sets:

V A one-to-one relation mapping elements of the script with elements in the trace that violate either the action description or the timing constraints.

L Lost actions in the script that are not mapped to actions in the trace.

X Extra actions in the trace that are not mapped to actions in the script.

Figure 3 illustrates the definition of these sets. Set M shows those actions that satisfy the script. Note that if a script constraint is not met by a given trace, it is ambiguous whether the constrained action should be considered lost or whether it is somehow related to an action in the trace that is in violation of the constraint. This ambiguity can only be resolved through constraints on V that use knowledge of the application semantics.

We refer to the definition of the sets M, V, L and X as MAP:

$$MAP \quad \equiv \quad \exists M \in R121 : dom(M) \subset S \wedge ran(M) \subset \mathbf{tim}(tr)$$

$$\wedge L \equiv S - dom(M)$$
$$\wedge X \equiv \mathbf{tim}(tr) - ran(M)$$
$$\wedge \exists V \subset M : \forall((a, I), (a', I')) \in M - V : a = a' \wedge I' \subset I$$

A QOS specification may allow for specific actions to be lost, or it may place some constraint on the number or pattern of missing actions. Similarly, extraneous actions and constraint violations may be allowed for individually or with constraints on groups of actions. In general, the QOS specification is a conjunction of logical formulae that express constraints on these three groups of exceptions.

The "perfect" QOS specification, $\mathcal{Q}_{\mathcal{P}}(S)$, allows no exceptions, that is:

$$\mathcal{Q}_{\mathcal{P}}(S) \equiv MAP \wedge (V = X = L = \{\}) \equiv \mathcal{M}(S)$$

We would like to be able to require that no more than one percent of the actions in a script are missing in a trace. Using the definition of the set L, our QOS specification can restrict the size of this set:

$$MAP \wedge (V = X = \{\}) \wedge (|L| \leq 0.01\,|S|)$$

As another example, a QOS specification may allow for a range of offsets for the script's start time:

$$\exists t : t_1 \leq t \leq t_2 \wedge \mathcal{Q}_{\mathcal{P}}(S + t)$$

In general, a script-based QOS specification can be written as a logical formula of the form

$$\exists t : t_1 \leq t \leq t_2 \wedge MAP_{S+t//S} \wedge EXCEPTIONS$$

where $EXCEPTIONS$ contains free variables for the sets M, V, L and X defined in MAP.

5.5 Example Specification of Audio Playback.

To see how a QOS specification can be used in a practical example, consider the requirements for real-time playback of digital audio. On a Sun Sparcstation, the audio device consumes a byte stream at the rate of 8K bytes/sec. The audio hardware takes care of the digital to analog conversion and the precise control of the sample output rate. The actions that we are concerned with are the writing of data from a user process to `/dev/audio`. Let's assume that the user process writes 1K of data at a time (enough for 1/8 second) and the device can buffer up-to 8K. The user process must ensure both that the buffer does not overflow, causing a loss of data and that the buffer does not become empty, causing the device to go silent.

The clock events can be derived from the audio device, by subtracting the number of bytes in the buffer from the total number written. If starvation occurs (the buffer runs empty), we will set the clock to its maximum value rather than having it pause. Clearly, this clock will mark any actions that occur after starvation as having occurred late. Ideally, the write process will satisfy a script that maps the writing of the nth 1K block of data to the clock interval $(1000 \cdot n, 1000 \cdot n)$:

$$S = \{(write_0, (0, 0)), (write_1, (1000, 1000)), \ldots (write_n, (1000n, 1000n))\}$$

Recall that this constraint implies that the action $write_n$ begins after clock event $t.(1000n)$ and completes before $t.(1000n + 1)$. If our user process obeys this script perfectly then there can be no buffer overflows or starvation. This script is unnecessarily strict though, because it does not allow the user process to work ahead, filling the 8K buffer. Also, many audio applications can tolerate occasional noise and lost data and the user process may be more easily implemented if a perfect stream of data is not required.

The following functions allow us to quantify constraint violations for a pair of constraints $(a, (i, j))$ and $(a', (i', j'))$:

$$early(\ (a, (i, j)),\ (a', (i', j'))\ ) \equiv i - i'$$

$$late(\ (a,(i,j)),\ (a',(i',j'))\ ) \equiv j' - j$$

while these boolean functions compare the actions and the intervals respectively:

$$same(\ (a,(i,j)),\ (a',(i',j'))\ ) \equiv a = a'$$

$$before(\ (a,(i,j)),\ (a',(i',j'))\ ) \equiv j < i'$$

The timing constraints in S can be relaxed to allow workahead (up-to 7 1K blocks) with the following constraint:

$$WORKAHEAD \equiv \forall (c,c') \in V : same((c,c')) \wedge early((c,c')) \leq 7000 \wedge late((c,c')) \leq 0$$

To insure that blocks are still written in order to the output device we add another constraint:

$$INORDER \equiv \forall (c,c'),(d,d') \in V : (same(c,c') \wedge same(d,d') \wedge before(c,d)) \Rightarrow before(c',d')$$

Note that all intervals in the trace will be non-overlapping since they are performed serially.

Since failure to write a block of data would cause a loss of synchronization in the playback, we must require that a replacement block of data be written for each block that is unavailable even though this will cause noise. We can map each such replacement action to the appropriate action in the script, but these mappings will be in the set of violations since the value of the data copied does not meet the specification. Constraints on the frequency of data substitutions can be expressed easily with a new definition. Let span(S) be the distance between the lowest and highest clock values from all intervals in the set S. For example:

$$span(\{(a,(0,2)),(b,(1,2)),(c,(4,6))\}) = 6$$

Then the following constraint prohibits more than 5 blocks of substituted data in any interval of less than 50 seconds (at 8000 samples/second).

$$NOISE = \neg \exists N \subset V : \forall (c,c') \in N \neg same(c,c') \wedge span(N) < 400000 \wedge |N| > 5$$

The full QOS specification for the user process that writes the audio data is then:

$$MAP \wedge WORKAHEAD \wedge INORDER \wedge NOISE \wedge (L = X = \{\})$$

5.6 Multimedia Scripts

Multimedia scripts are created by specifying synchronization of a set of media presentation actions. A single media presentation action specifies the transfer of data from a typed data source to a logical output device. For example, an action can specify the copying of the nth frame from an MPEG compressed color video file to a window on a one-bit display. In this case, the source is the output of a pipeline of processes which respectively read from the file, decompress the data and transform the single frame (compressing in a different way) to a one-bit representation. The output device is a window that is accessed via a window system display function. The only action that the script will directly constrain in time is the transfer of data to the logical window.

The specification of sources and sinks for a presentation action do not specify implementation. In particular, while the video pipeline could write directly to the window, the constraints on the presentation action can also be met by introducing a buffer between the pipeline and a display process so that writes to the window are decoupled from delays in the pipeline.

Similarly, the specification of the output device is a logical description of the device characteristics so that user interface manager retains control of physical resources. The logical device specification includes spatial layout and color mapping for graphics displays.

5.7 Script Composition Operators

In order to synchronize two actions in real-time they must be constrained according to the same real-time clock. Since the clock events in a script refer to values of a common clock, all actions in that script are synchronized with respect to each other. We would like to compose simple scripts, synchronizing their elements, to form complex scripts. To make this easier, we define the following composition operators, beginning with the time-shifting and scaling operators:

$$S + t \equiv \{(a, I + t) | (a, I) \in S\}$$

$$S * f \equiv \{(a, I * f) | (a, I) \in S\}$$

$$synch(S_1, S_2)) \equiv S_1 \bigcup S_2$$

$$S1 : S2 \equiv synch(S_1, S_2 + maxtime(S_1))$$

where $maxtime(S)$ and $mintime(S)$ are respectively the largest and smallest time values referenced in S.

$$iterate(n, S(i)) \equiv S(1) : S(2) : ...S(n)$$

where $S(i)$ is a script generation function that takes an integer argument.

$$clip((i, j), S) \equiv \{(a, (k - i, l - i)) | (a, (k', l')) \in S \wedge k \leq l \wedge k = max(i, k') \wedge l = min(j, l')\}$$

The ability to clip suggests another operation to reverse the clipping operation:

$$source(clip(I, S)) \equiv S$$

and another to modify it:

$$trim(i, j, clip((k, l), S)) = clip(t3 + t_1, t4 + t_2, S)$$

If a script S has not been clipped from any other, then:

$$source(S) \equiv S$$

One would like to extend these composition operators to apply to the QOS specifications that may apply to sup-components of a presentation. In order to preserve the meaning of QOS specifications when scripts are synchronized, the quantification of the MAP variables must continue to apply to the same set of actions as before. For example:

$$synch(Q_1(S_1), Q_2(S_2)) \equiv Q_1(S_1) \wedge Q_2(S_2)$$

These composition operators, along with standard parameterized definitions for error tolerances, can be used to specify common presentation types such as continuous media playback. More work needs to be done to provide definitions for high-level specifications and to show how QOS specifications are affected by the remaining composition operators.

6 Discussion

In Section 2 we listed 7 desirable features that a specification language for non-interactive multimedia presentations should have. In this section, we consider how well the Timesync language meets these goals.

- Physical data independence. Timesync specifications refer only to logical data objects, allowing physical data pathways to be optimized as late as possible.

- Physical device independence. Both inputs and outputs are specified by logical attributes. A Timesync specification may be executed on any configuration of devices so long as the output resolution and other logical attributes are satisfied.
- Recursive composition of presentations. All the composition operations in Timesync can be applied recursively with Timesync specifications as operands.
- Formal semantics for error tolerances. Timesync specifies its tolerance for errors via logical formulae that are unambiguously true or false when bound to the trace of a presentation. The value of this formalism is in exploiting knowledge of system resource availability and delays to prove that a specification *can* be satisfied.
- Completeness in expressibility. 0 allows specification of a presentation trace with arbitrary (within the resolution of the clock) synchronization between presentation actions. By widening the timing constraints on individual actions it is also possible to specify a set of traces, all of which satisfy the timing constraints. While it is desirable that a specification technique be able to express the largest set set of traces that capture the meaning of a presentation, we argue that, since Timesync can specify any single trace, it is complete.
- Soundness in constraint specifications. Since all primitive scripts are sound and all composition steps preserve soundness, we conclude that all Timesync specifications are sound by induction.
- Simplicity. The primitive notion of a script is a simple way to specify synchronization of an arbitrary number of actions. Iteration and recursive composition allow us to compactly specify large sets of actions with complex timing relationships.

Although we argue that Timesync is complete in its ability to express synchronization, it is worth discussing weather it shouldn't also be complete in its ability to specify allowable variations in synchronization. For example, it seems natural to specify constraints on the rate of continuous media presentations while making no restrictions on the amount of long-term drift from a static schedule. Such rate constraints are expressed as synchronization relations between presentation actions as opposed to the Timesync approach where actions are synchronized with the clock and only indirectly with each other. In Timesync it is difficult to specify that event b should occur t seconds after event a if a is constrained only to occur between $t1$ and $t2$ with $t2 - t1 >> t$. Because a Timesync script specifies all synchronization relative to a clock rather than between events, all constraints between events must involve the exception specification.

Since we have already noted how a rate-based specification can be interpreted as constraints relative to a real-time clock the question that remains is why would one want to allow drift? If the concern is that we be able to use a physical clock that is imperfect, then the drift is transparent to the playback tool since it does not see any other time source. On the other hand, if the concern is for error handling, then we have a real debate. When a presentation action is delayed, a rate-based approach might propagate the delay to subsequent actions to avoid skipping. The static scheduling approach considers each late action as a constraint violation but expects subsequent actions to maintain the original schedule. While the rate-based approach minimizes the loss of information in a single stream, it makes it more difficult to maintain synchronization between multiple streams. The static approach requires each stream to synchronize only with a single global clock.

7 Conclusions

This paper shows how to produce a formal process specification from a real-time script. Our definition of a script is simple and intuitive as all synchronization is expressed relative to a single real-time clock. Allowances for QOS degradation can be added to the formal specification through constraints on the actions in a trace that do not strictly satisfy the script. The result is that Timesync specifications can be used in a request for guaranteed service from an operating system. The operating system's acceptance test must then analyze the Timesync specification in order to identify resource requirements and to make reservations.

We intend to use Timesync specifications to request real-time services from a prototype of a digital television editing workstation. In this application, each press of the play button initiates a request for real-time service that the operating system may accept or reject depending on resource availability. Such a prototype will require a method for generating Timesync specifications from the playback tool, algorithms for planning real-time tasks to meet the timing constraints, analysis and reservation of the resource requirements.

The idea that each play request is independently subjected to an acceptance test is admittedly naive in that the user may demand predictable response during an entire editing session. The specification of resource requirements for interactive editing sessions requires further research.

References

[1] M. Abadi, L. Lamport: An Old-Fashioned Recipe for Real Time. Tech. Rept. 91, DEC Systems Research Center, October 1992.

[2] David P. Anderson: Metascheduling for Continuous Media. ACM Transactions on Computer Systems, Vol. 11, No. 3, August 1993, pp. 226-252.

[3] G. Berry, G. Gonthier: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Tech. Rept. Res. Rept. No. 842, INRIA, 1988.

[4] Director 3.1, Studio Manual. Macromedia Inc., March 1993.

[5] G.D. Drapeau, H. Greenfield: MAEstro - A Distributed Multimedia Authoring Environment. Proceedings of the Summer 1991 USENIX Conference, USENIX Association, pp. 315-328.

[6] C.A.R. Hoare: Communicating Sequential Processes. Prentice-Hall International, London, UK, 1985.

[7] M.E. Hodges, R.M. Sasnett, M.S. Ackerman: A construction set for multimedia applications, IEEE Software, January 1989, pp. 37-43.

[8] K. Jeffay, D.L. Stone, T. Talley, F.D. Smith: Adaptive, Best-Effort Delivery of Digital Audio and Video Across Packet-Switched Networks. Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video, November 1992, pp. 1-12.

[9] A. Lazar, G. Pacifici: Control of Resources in Broadband Networks with QOS Guarantees. IEEE Communications Magazine, October 1991.

[10] T.D.C. Little, A. Ghafoor: Interval-Based Conceptual Models for Time-Dependent Multimedia Data. IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 4, August 1993, pp. 551-563.

[11] Apple Computer, Inc: Inside Macintosh: QuickTime, Addison-Wesley Publishing Co., 1993.

[12] Jean Ramaekers, Giorgio Ventre: Quality-of-Service Negotiation in a Real-Time Communication Network. Tech. Rept. TR-92-023, International Computer Science Institute, Berkeley, April 1992.

[13] G.M. Reed, A.W. Roscoe: A Timed Model for Communicating Sequential Processes, Proceedings of the 13th International Colloquium on Automata, Languages and Programming, July 1986, Lecture Notes in Computer Science, No. 226, Springer-Verlag, pp. 315-323.

[14] R. Staehli, J. Walpole: Constrained-Latency Storage Access. Computer, Vol. 26, No. 3, March 1993, pp. 44-53.