Coordinating Heterogeneous Time-Based Media Between Independent Applications

Scott Flinn
flinn@cs.ubc.ca

Department of Computer Science
The University of British Columbia

Abstract

This report discusses the requirements and design of an *event scheduler* that facilitates the synchronization of independent, heterogeneous media streams. The work is motivated by the synchronization requirements of multiple, periodic, logically independent auditory streams, but extends naturally to include time-based media of arbitrary type. The scheduler design creates a framework within which existing synchronization techniques are composed to coordinate the presentation activities of cooperating or independent application programs. The scheduler is especially effective for the presentation of repetitive sequences, and guarantees long term synchronization with a hardware clock, even when scheduler capacity is temporarily exceeded on platforms lacking real time system support. The implementations of the scheduler and of several application programs, class libraries and other tools designed to use or support it are described in detail.

Keywords

Media integration and synchronization; real-time scheduling; system architecture; auditory display; distributed systems; operating system support.

Contents

List of Figures

1 Introduction

There are many open problems in the control of time-based media, ranging from the physical transport of video streams, through synchronization of sound and image, to real-time interactive control of a presentation. This report deals with one particular aspect of time-based media control; namely, the coordination of presentation between independent applications.

Work in this area was motivated by a number of investigations into the creation of continuous or periodic auditory displays. Our long term objective is to create a subtle and continuously present ambient acoustic environment capable of conveying information concerning the state of a workstation or network, and the progress and behaviour of ongoing computations. The effort began with the implementation of a number of auditory effects that are well known to psychologists but whose effectiveness in uncontrolled, noisy environments is poorly understood [5].

Three observations became apparent during the early implementation work. First, simultaneous auditory streams must be coordinated. When independent streams are not sufficiently separated along some salient dimension such as pitch or timbre, they tend to fuse perceptually into a single stream [2]. Coordination is therefore required to ensure either that logically independent streams are sufficiently separated, or that they merge in a predictable way.

Second, the temporal aspects of musical and other acoustical presentation require a relatively high degree of flexibility. For example, the timing requirements of digital video, while extremely demanding in practice, are conceptually very simple. It is only when many video and audio segments are composed to form a larger presentation that timing requirements become involved. In contrast, a single synthesized auditory stream can exhibit rhythmic intricacies that cannot be predicted in advance and to which humans are extremely sensitive. Precise control over the relative timing of its elements is therefore required.

Third, to our knowledge there are no existing solutions to the problem of coordinating the activities of independent applications. Our search for a suitable implementation platform revealed many systems for coordinating a diverse range of elements in a unified multimedia presentation, but none that address the specific problem of coordinating independent activities.

This report describes the design and implementation of a system that addresses these requirements. Its principal purpose is to facilitate the creation of a rich acoustic environment, but the flexibility it provides for the creation of sophisticated acoustic patterns can be easily applied to the coordination of a much wider range of activities. The next section reviews work in a number of related areas that bears on the current problem. Section 3 expands on the motivation and requirements outlined above. The system design is described in Section 4, and Section 5 follows with a discussion of the implementation of the system itself as well as of several application programs that exploit its abilities. We conclude in Section 6 with a summary of the system requirements and design, and a discussion of some promising directions for future work.

2 Related Work

Although we are unaware of other work that addresses the specific problem of coordinating the presentation of heterogeneous media streams between independent applications, there are several closely related problems for which adequate (and ever improving) solutions exist. The design proposed in this report relies in some way on many of these solutions.

2.1 Presentation scheduling

The task of coordinating the presentation of multiple, conceptually distinct media streams is at the heart of many multimedia products and research efforts. Apple's *QuickTime* system, for example, directly supports the composition and display of multimedia documents (movies) containing an arbitrary number of tracks of time-based media [3]. Internal synchronization of media streams and a flexible framework for including modular drivers for new devices and media types are among the strengths of the design. The system coordinates the activities of multiple applications to the extent that it mediates contention for hardware resources, and applications can share movie files. There is no facility however for synchronizing a stream with another that is already in presentation. In addition, the expressive power of the synchronization specification does not extend beyond specifying the start and end points of multiple media tracks relative to the start of the presentation.

An example of a more expressive presentation specification language can be found in *HyTime* [10], which defines a framework for the design of specific document formats that include spatial, temporal and logical relationships between document components. Unlike QuickTime, HyTime is a specification only; HyTime compliant software applications are required to create, query or render documents.

Musical performance software provides another example of presentation scheduling. The Musical Instrument Digital Interface (MIDI) standard has facilitated a diverse range of multi-track sequencing and editing software. The Music Kit developed for NeXTStep workstations is a very complete example of this genre. It is an object oriented software kit that is designed around the metaphor of an orchestra. A *conductor* coordinates the activities of one or more *performers* which together constitute an *orchestra*. Performers have *instruments* on which they play *parts* that are assembled into *scores*. Each part is composed of a time stamped collection of *notes*. A note may initiate, terminate or modify a synthesized voice, in which case it will contain all of the parameters relevant to the particular synthesis technique, or it may contain an arbitrary MIDI packet. Scores may be specified in a *score file* language which can be interpreted at performance time or generated dynamically. The score file notation supports musical devices such as repeats and codas. A *performance* in this system can have any number of conductors directing any number of players to perform possibly independent scores on their instruments. Instrument classes can be subclassed to produce new instruments, and there are no restrictions on the way in which an instrument renders a note. Finally, a conductor can be used as a general purpose notifier to dispatch messages between arbitrary objects in a timely fashion. The core of this design is used as the basis for our solution.

2.2 Audio and video servers

The *AudioFile* system [8] is an audio server that operates in a manner analogous to the X Window System [14]. Multiple local and remote clients communicate with a local server according to a specific protocol, requesting that certain rendering be done on the local workstation. Clients make requests through a device independent interface; the server mediates access to the sound generation hardware and maps requests for services not supported by the local hardware into suitable approximations. AudioFile provides the infrastructure necessary for coordinating auditory streams from independent applications, but does not actually address the problem of synchronizing independent streams.

Arons describes a framework and set of tools for constructing asynchronous servers for speech and audio applications [1]. The audio server described is used only for playback of digital samples, which avoids many of the problems that arise when individual synthesized tones are assembled into a continuous presentation. However, the design is strongly influenced by the need to provide feedback to an application regarding the processing of its requests.

Many of the issues addressed in this report do not yet arise in the context of video servers. It is sufficiently difficult to simply transport the enormous amounts of data involved (Gemmell and Christodoulakis effectively demonstrate the magnitude of the challenge [6]) that most systems are not yet in a position to exploit more sophisticated composition of multiple streams.

2.3 Media synchronization

In recent years there has been a great deal of interest in the general problem of multimedia synchronization. This report does not seek to extend this work directly, but rather proposes a higher level of synchronization that depends on existing solutions. The proposed framework affords natural opportunities to utilize formal approaches such as the timed Petri net approach of Little and Ghafoor [9], the synchronous language model of Horn and Stefani [7], or the event stamps and restricted blocking of Steinmetz [16]. The framework also has the flexibility to exploit more detailed software designs such as the *Ttoolkit* of Guimaraes et al. [11] and the feedback techniques for media continuity and synchronization developed by Ramanathan and Rangan [12, 13].

3 System Requirements

The work described in this report was motivated by the need to coordinate the auditory presentation activities of multiple independent application programs. Recall that our long term goal is to create a rich acoustic environment in which the behaviour or state of devices, services and ongoing computations is represented through auditory displays.

3.1 Synchronization

Consider the situation in which two independent applications choose to represent some aspect of their behaviour by generating a rhythmic acoustic pattern. It is easy to demonstrate that two rhythmic patterns will fuse to form a single more complex percept unless the two components are well separated along a salient dimension such as pitch or timbre [5]. In the absence of more sophisticated system resources, independent applications will not have the means to ensure this separation. Even if this was possible, the need to shift pitch or timbre to ensure separation may conflict with the coding of the information being conveyed. A suitable compromise is to synchronize the rhythmic phase of auditory streams to ensure that any two or more streams will always generate the same complex pattern when presented simultaneously. This will dramatically reduce the space of possible rhythms that can be generated by a fixed set of programs, allowing the possibility of becoming familiar with many common combinations. A mechanism for achieving phase synchronization of independent auditory streams would therefore be useful.

3.2 Modular media handlers

There are three ways to address the phase synchronization problem. First, applications can perform their own phase calculations based on a global clock and a convention that establishes when the first cycle began. This has the disadvantage of requiring every application to provide its own scheduling infrastructure, although the use of a well designed shared library for this purpose would minimize the disadvantage. Second, a separate process could be used to coordinate scheduling, sending signal messages to applications to trigger presentation. If the time required for each message is roughly constant then the extra delay introduced by the messaging will be equivalent to a small global phase shift, which is inconsequential since the phase origin will ordinarily be chosen arbitrarily.

Both of these approaches, however, do not address the issue of resource contention. Mediation of visual display resources, for example, is commonly accomplished through a window system in which a separate process coordinates which windows will be displayed and which will be partially hidden. To prevent applications from drawing on parts of the screen they do not own, drawing is normally accomplished by sending suitable instructions to the window server process which then decides which parts are actually visible. Consider how the synchronization approaches outlined in the previous paragraph must be integrated into such a system. When the time for the next presentation element is signalled (either by a global clock event or a message from a scheduler process), the presentation must be accomplished by sending a request to the window or audio server, including a potentially large amount of data as would be required for a large image or a high quality audio sample.

A third possibility is to have the media dispatched by the scheduler itself. With this arrangement, display actions are scheduled in advance with the display server (where the display may be visual, auditory or tactile) and all necessary data is delivered as part of the scheduling process. When all actions are scheduled for immediate execution, the scheduler behaves exactly like a conventional window or audio server. Actions scheduled for a later time benefit from more timely presentation since the overhead of acquiring the data has already been incurred. Even more precise presentation scheduling is then possible since the scheduler may be provided with the initial start up characteristics of the display device and can cue the action sufficiently far in advance. The benefit of accounting for start up time may be negligible for a small image, say, but will be far more significant in the case of a video clip presentation that involves starting a VCR.

The value of providing the scheduler with presentation data in advance increases dramatically when repetitious patterns, acoustic rhythms for example, must be generated. In this case, only one cycle of data need be sent to the scheduler which can retain it for periodic dispatch.

According to this view, all of the hardware resource mediation software is gathered together into a single scheduler process. A monolithic implementation of such a process would preclude the possibility of adding new display devices, or new media types that use existing devices, without significant redesign. It is therefore essential that the system architecture support a modular design for media handlers that can be integrated into an existing scheduler according to a standard protocol.

3.3 Module interaction

The preceding discussion of resource contention considered only the problem of mediating access of application programs to a display resource. Given the requirement for modular media handlers, it is also necessary to mediate access of modules that rely on the same display device. For example, most popular sound cards use the same basic hardware to achieve precise play back of digitized samples as to create synthesized tones. However the characteristics of these two media are very different and would best be controlled by different modules. A similar comment can be made concerning the relation between still image and video display. A mechanism for implementing a policy of priority and preemption is therefore required.

The requirements listed in this section are concerned only with the highest level of synchronization: the relative start times of presentation elements. The processing of a video clip, for example, is considered complete when the instruction to display the first frame is dispatched. Clearly such operations as the timely presentation of video or the coordination of a video stream with a sound track require synchronization at a finer granularity. This will also require interaction between modules. However, details of interaction at this level are not considered here. There are many existing and sophisticated solutions for this level of synchronization (QuickTime for example), and we are principally concerned with the composition of these mechanisms at the inter-application level. Section 5, which discusses implementation, indicates how the facilities of a system like QuickTime can be exploited directly in the coordination of presentations between applications.

3.4 Feedback and performance

Even though application programs need not perform their own presentation scheduling, they should retain the ability to synchronize other actions with the presentation. A simple acknowledgement message sent from the scheduler at the time of dispatch, identifying the display action performed, facilitates this without incurring the overhead of transferring a substantial amount of data at the moment the action is taken.

Finally, performance and overhead are important concerns for every aspect of the system design. Most importantly, a suitable policy must be adopted when the display demands of application programs exceed the capacity of the system. The following section presents the design of a system that addresses the requirements outlined here.

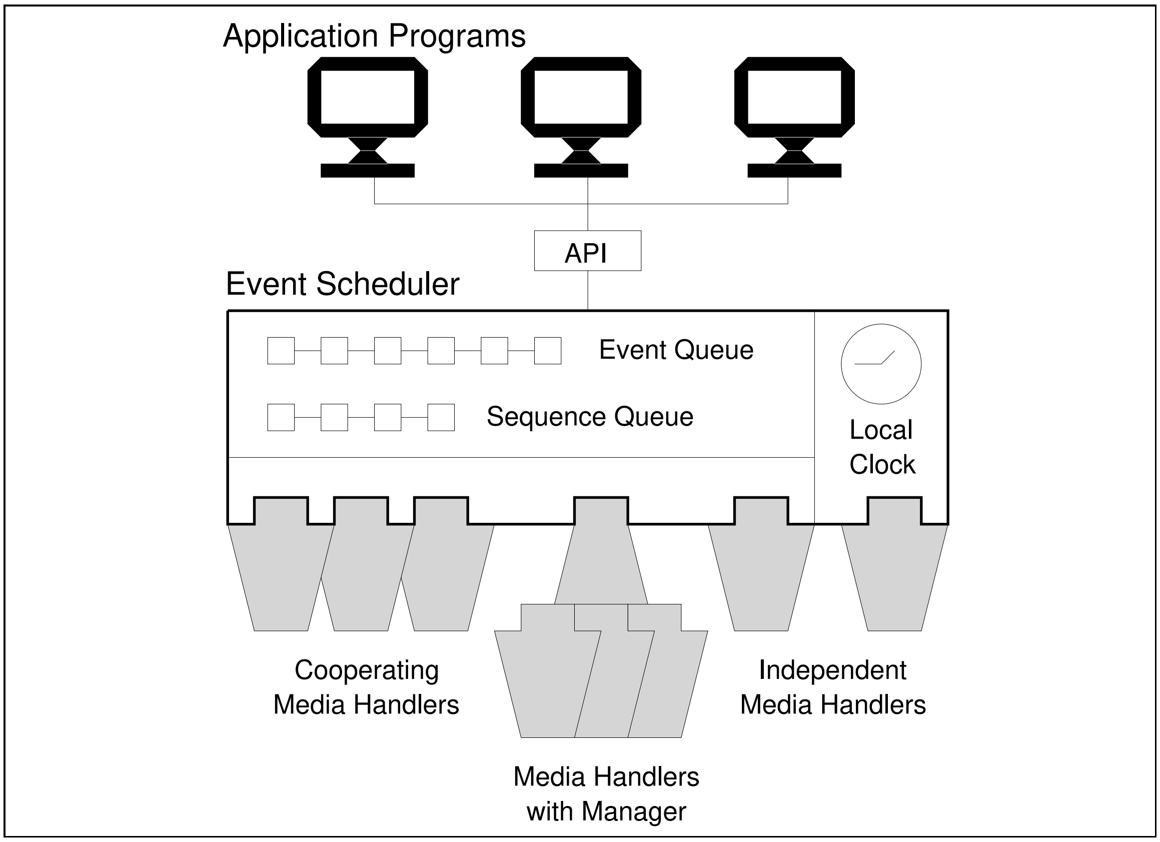

Figure 1: Event Scheduler Framework

4 Event Scheduling

The *event scheduler* is the heart of the media coordination system and establishes the framework within which a collection of modules cooperate. Scheduling functions in the current version are limited to sorting events by time stamp and dispatching them at the proper times, however even this simple arrangement offers considerable flexibility. As we shall see, there are also natural places within the existing framework to integrate more sophisticated scheduling algorithms. The scheduling framework, illustrated in Figure 1, is described in this section.

4.1 Events and sequences

The *event* is the common unit of currency in the system. An event has a type, a time stamp and a pointer to data of unspecified format. When the scheduler's clock reaches or surpasses the time stamp of an event, the event is *dispatched* to the appropriate handler as determined by the type identifier. The module then takes some action based on the associated data, usually resulting in the presentation of visual or auditory material.

A *sequence* is a collection of events ordered by time stamp. In this context, the time

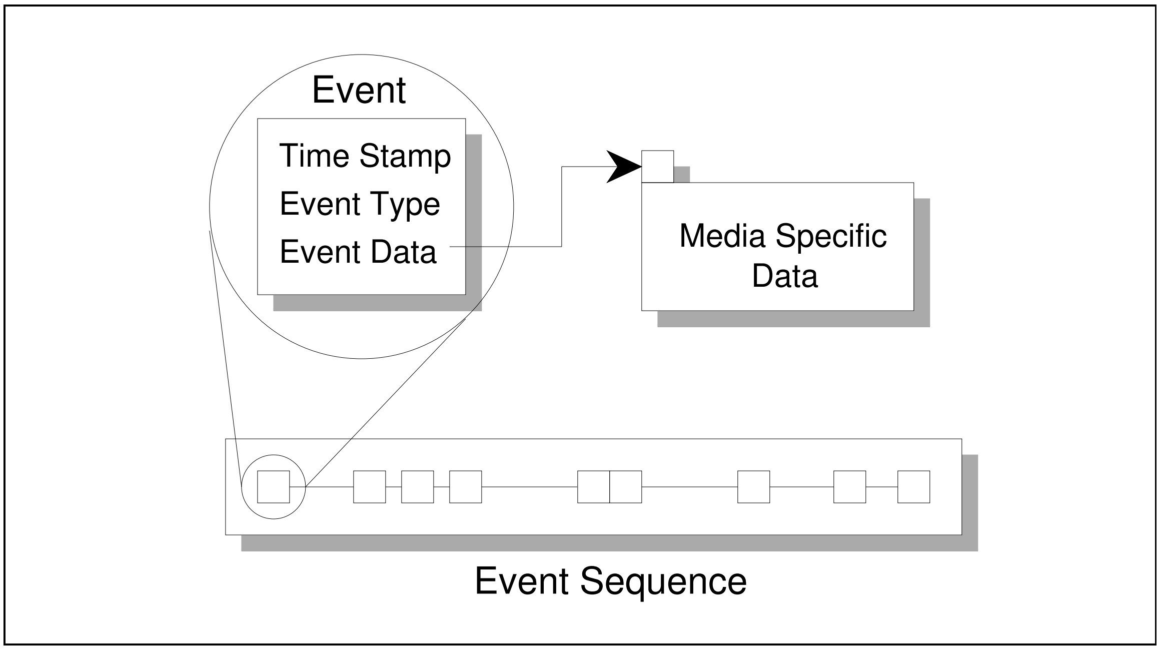

Figure 2: Event and Sequence Composition

stamp of an event is used slightly differently: it specifies the dispatch time of the event relative to the start time of the sequence, rather than in absolute terms. A sequence is created with a period, a repetition count and an absolute time indicating when it should first be *scheduled*. It is rescheduled once each period until the repetition count, which is decremented by one each time, reaches zero. Figure 2 illustrates how events and sequences are composed.

4.2 Event and sequence queues

The interaction between event dispatching and sequence scheduling is coordinated by means of two priority queues. The *event queue* is simply a list of events ordered by absolute time stamp. The scheduler sets an alarm for the time of the first event's time stamp and then sleeps. When the alarm is signalled, the scheduler dispatches the event to the appropriate module and the process is repeated. When an application program schedules an event directly, the absolute time stamp is determined from the scheduler's own notion of the time and a delay specified by the application, and the event is inserted into the proper position in the queue. If its proper position is at the head of the queue, then the scheduler's next alarm is reset.

The processing of sequences is only slightly more involved. The *sequence queue* is a list of sequences to be scheduled. Each sequence in the list has an absolute time stamp indicating when it is to be scheduled, and the list is ordered by time stamp. Just as for event processing, the scheduler sets an alarm for the time of the first sequence's time stamp and then sleeps. When the alarm is signalled, the sequence is *scheduled* and its repetition

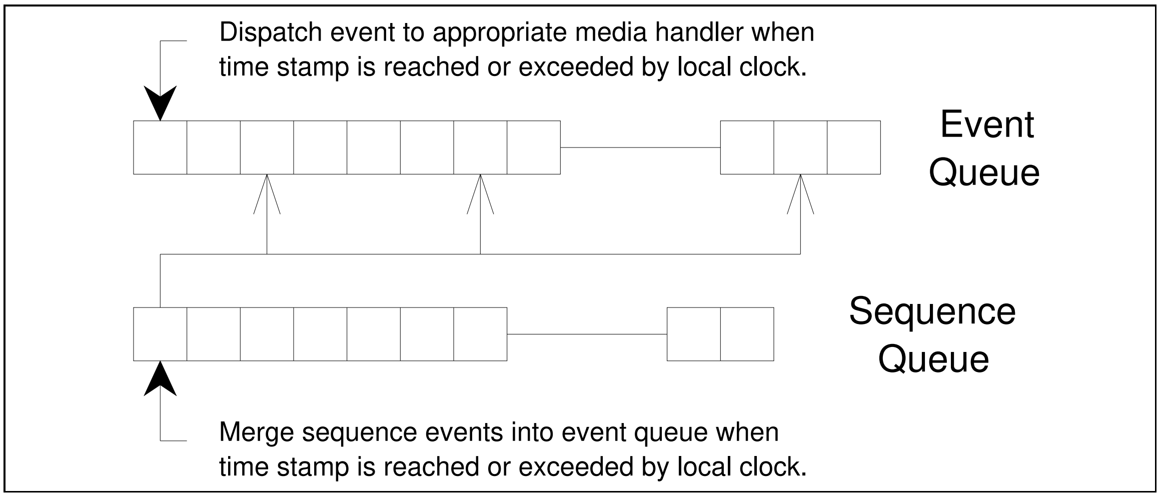

Figure 3: Event and Sequence Queues

count is decremented by one. If the count remains positive, the sequence period is added to the absolute time stamp and the sequence is inserted into the queue according to the new value.

A sequence is scheduled simply by adding its absolute time stamp to the relative time stamps of each of its events and inserting the events into the event queue according to the new values. Figure 3 illustrates the interaction between the event and sequence queues. While the simplicity of this mechanism permits an efficient implementation, it offers a great deal of flexibility. Events of different media types can be mixed to form a precisely synchronized presentation. Rhythmic patterns are generated simply by giving a repetition count greater than one (a negative value is used to indicate unbounded repetition). The scheduler supports a number of simple operations on sequences that can be used to synchronize them and to modify the relative timing of the events they contain. The application of media specific modifications to sequence events is also supported.

This combination of absolute and relative time stamps produces an important characteristic of the scheduler in its treatment of synchronized sequences. When a synchronization operation is applied to a set of sequences (as described in Section 4.4), their absolute sequence queue time stamps are adjusted to produce the type of synchronization requested. Since the time stamps of sequence events are relative, this adjustment is reflected in the computation of absolute event queue time stamps for each event when the sequence is scheduled. Each relative time stamp is added to the absolute start time of the sequence to determine an absolute dispatch time for the event. Similarly, when subsequent iterations of a sequence are re-inserted into the sequence queue, the new absolute sequence time stamp is computed by adding the period to the old absolute time stamp. The local clock is not involved in the computation of these absolute time stamps; it is only compared against them to determine if an event is due for dispatch or a sequence for scheduling. In other words, precise calculation of absolute time stamps does not depend on meeting real time constraints. This results in the following behaviour: if the computational limits of the machine are temporarily exceeded and a number of queue processing deadlines are missed, calculation of subsequent time stamps will be unaffected by the delay. Once computational load decreases sufficiently, processing will continue relative to the local clock as if there had been no imprecision. This allows sequence synchronization to be maintained even on platforms that cannot guarantee hard real time performance.

Finally, the initial scheduling time for a sequence can be specified in one of two ways. As for events, the time can be given as an offset from the time at which the scheduler receives the request. The fact that it is not possible to request an action at an absolute time implies that the relative timing of events that are scheduled individually cannot be precisely controlled. However, sequences exist for precisely this purpose. Similarly, operations are provided to coordinate the timing of individual sequences. These policies allow events and sequences to be synchronized across a network without concern for the synchronization of local clocks.

The scheduler can also be instructed to begin scheduling a sequence at the next integral period boundary. It records a global start time, t_0, during its initialization. When it receives a request at time t_r to schedule a sequence of period p in this way, it computes the first schedule time t_s as

$$t_s = \left\lceil \frac{t_r - t_0}{p} \right\rceil p + t_0. \tag{1}$$

All sequences scheduled by this mechanism have their phases synchronized at time t_0. If two sequences are synchronized in this way, and their periods p_i and p_j have a greatest common divisor gcd, they will combine to form a more complex pattern with period $p_i p_j / gcd$. This provides a simple solution to the phase synchronization problem.

4.3 Signals

It is often useful for an application to know when elements of its presentation have been dispatched. In interactive applications, for example, it is sometimes convenient to postpone decisions about presentation content as long as possible. In this situation an application might be signalled near the end of a segment that it is time to schedule the next one.

To accomplish dispatch notification, two additional pieces of information are recorded for each event: a *signal port* (Internet address and port number) and an integer *signal value*. The scheduler signals an application by sending the signal value (in standard network byte order) to the address specified by the signal port immediately prior to dispatching the associated event to a media handler. A negative signal value indicates that no signal is to be sent. The semantics of the signal values are left completely to the application, which can interpret signal values however it likes.

This mechanism was designed primarily to provide an application with timely notification that events it previously scheduled have been dispatched. However, a number of additional functions are possible. For example, a set of applications may all conform to a convention that ascribes specific semantics to specific signal values. Each application may then choose to have signals sent not only to itself but to one or more of the others in the set, facilitating more sophisticated coordination of application behaviour. Applications on different machines can coordinate their activities through a common scheduler accessed remotely, although the precision of the coordination will depend on the magnitude of network latencies.

4.4 Sequence operations

When the scheduler receives a request to schedule a sequence, it first assigns the sequence a unique identifier and returns it to the requesting application. The application can subsequently use this identifier to request that operations be applied to the sequence.

Sequence operations are divided into three categories: synchronization, time scaling, and media specific operations. Only one synchronization operation is provided: a client may supply a list of sequence identifiers to be synchronized. The scheduler simply adjusts the sequence queue time stamp of each sequence to the earliest time in the list and resorts the queue. Recall that a sequence may also be synchronized with a common time of origin when it is first registered with the scheduler.

Three time scaling operations are available. The first allows a sequence period to be scaled by a constant factor. The second scales the relative time stamps of the sequence events by the same constant factor. Since these two operations are independent, it is possible to create a sequence whose scheduling period is shorter than its actual duration, in which case successive iterations of the sequence simply overlap. The third operation specifies a list of scale factors. The ith element of the list is used to scale the time stamp of the ith sequence event. If the list is too short, it is extended with ones; if it is too long, its tail is ignored. This operation provides the means to adjust the relative placements of events within a sequence without interrupting its periodic execution.

All three time scaling operations may be applied to a list of sequence identifiers. The implementation actually stores the scale factors separately from the time stamps for each sequence or event, performing the multiplications at the time of scheduling. This allows absolute (that is, relative to the original time value) as well as relative (that is, relative to the current time value) scaling.

Media specific operations consist of a list of tag-data pairs, where the data is of arbitrary format and size. These operations are applied by forwarding them to the appropriate media handler for each event. Recall that events have an associated data block that is media dependent; these operations have the effect of modifying this data. Media specific operations can be applied to a list of identifiers.

This set of operations is relatively small, but has been sufficient to meet the needs of the applications described in Section 5. Additional operations can be easily added as the need arises. All operations are applied in place in the sequence queue, resulting in uninterrupted periodic scheduling.

4.5 Graceful degradation

Section 4.2 summarized the way in which the scheduler re-establishes synchronization with the local clock after periods of activity that exceed the capacity of the system to keep pace.

Here we consider the behaviour of the system *during* those periods.

The best way to achieve graceful degradation of a media stream usually depends on the type and purpose of the stream. The scheduler is not provided with this information regarding the streams it coordinates, so it must adopt a uniform strategy for all streams. The approach we have chosen gives a measure of control over the degradation process to the applications scheduling the streams. Each event can be given a *delay threshold* by the originating application. The threshold value is examined by the scheduler only when dispatching events from the event queue. If the current time is greater than the sum of the event time stamp and its delay threshold, then the event is simply discarded. A negative value indicates that an event should never be discarded. Typically an application will assign small threshold values to events initiating actions of short duration, larger values to events initiating longer duration actions, and a negative value to events signalling the termination of actions. It would also be possible to let the media handler for an event decide whether to discard it based on the given delay threshold. This has the advantage of making the discard decisions more media dependent, but at the expense of additional processing that will make the event even later before the decision is made.

Delay thresholds can also be applied to the sequence queue. When a sequence has surpassed its delay threshold its events are not inserted into the event queue, but it is still re-inserted into the sequence queue if its repetition count remains greater than zero.

4.6 Media handlers

The initial design includes a number of specific media handler modules that are necessary to support target applications. The *synth handler* is by far the most complex, and is used to control auditory synthesis using a digital signal processor (DSP). The handler is designed to use the lower levels of the NeXTStep Music Kit (described earlier), which allow the composition of simple synthesis elements to form sophisticated *synth patches*, each of which can synthesize a single complex voice. The synth handler manages a pool of synth patches that it uses to dispatch voice synthesis events. It also supports multiple DSPs, balancing the load as necessary between them. A *virtual DSP* is used when all others are at capacity to ensure that updates to active synth patches are applied correctly. The handler can dynamically adjust the number of DSPs it uses, migrating synth patches to other DSPs (including the virtual DSP) as necessary. A more complete description of the synth handler is given in Section 5.3.1.

The *sample handler* is used to render digital audio samples on the DSP. Although a single DSP can synthesize twenty or more 44.1 KHz stereo voices simultaneously, the software does not support digital sample playback concurrently with synthesis. If no DSP is free, the sound handler therefore notifies the synth handler that it must reduce its DSP usage by one. The sound handler uses that DSP to play the sound and then releases it back to the synth handler. Currently this sort of module interaction is done in an ad hoc fashion. A more sophisticated module subsystem, similar to the Component Manager of Apple's QuickTime system, would be required for a more extensible implementation.

The current implementation also includes *DPS*, *video* and *IPC* handlers. The DPS handler forwards Display PostScript code to the NeXTStep window server in the context of a window owned by the scheduler. The video handler sends simple commands (such as play, pause and stop) to a video disk player connected through a serial port, and the IPC handler forwards simple messages to external processes. These have been included as a proof of concept and have not been used extensively by application programs.

5 Implementation

The design outlined in the previous section has been implemented on a NeXT workstation and has undergone a number of major revisions, progressing each time toward greater generality and flexibility, in response to the requirements of new applications. This section describes the implementation environment and major system software components, including libraries and application programs that have been designed to both support and exploit the scheduling system.

5.1 Implementation environment

The scheduler has been implemented under the NeXTStep operating system. The choice of software platform was influenced by two criteria: the need to rapidly prototype flexible interactive applications, and the existence of a sophisticated sound synthesis package that is easily integrated with other development tools. The NeXTStep development application tools, such as InterfaceBuilder, ProjectBuilder, Mig (the Mach interface generator) and MallocDebug, and the services and classes of the standard Application Kit, allow prototype applications to be assembled very quickly. The Music Kit was described briefly in Section 2.1. It was originally part of NeXTStep and is now a public domain offering maintained by CCRMA at Stanford University. It allows flexible low level control of the DSP synthesis hardware, but is most commonly used by applications through a set of Objective-C classes that includes Orchestra, Conductor, Performer, Instrument, Score, Part, Note and many other more specialized classes. It has been tuned to the NeXTStep platform, making use of features of the underlying Mach kernel, such as multi-threading and fixed priority scheduling, to achieve precise performance. These characteristics make it a perfect choice for our needs.

Although NeXTStep emulates 4.3 BSD Unix very closely, we have chosen to make several parts of the scheduler implementation dependent on specific features of Mach. For example, Mach threads are exploited for a number of purposes, and the Mach interface generator is used to build RPC stubs for the server and client library. However, no facilities are used that do not have an effective counterpart in standard Unix. It would therefore be straight forward to port the scheduler to a Unix platform. Since most of the media handlers are platform dependent, the dependence of the remainder of the scheduler on NeXTStep was not seen as a significant limitation of the prototype system.

5.2 Scheduler implementation

The implementation of the scheduler follows directly from the design presented in Section 4. Figure 4 illustrates how the elements of the design are composed. The main thread, shown as a rectangle in the figure, controls most of the scheduler and media handler functions. Two separate threads are used as notifiers for the event and sequence queues. When the event queue is non-empty, an *event notifier* thread is created that simply sleeps until shortly before the event time stamp is reached at which time it sends a message to the scheduler indicating

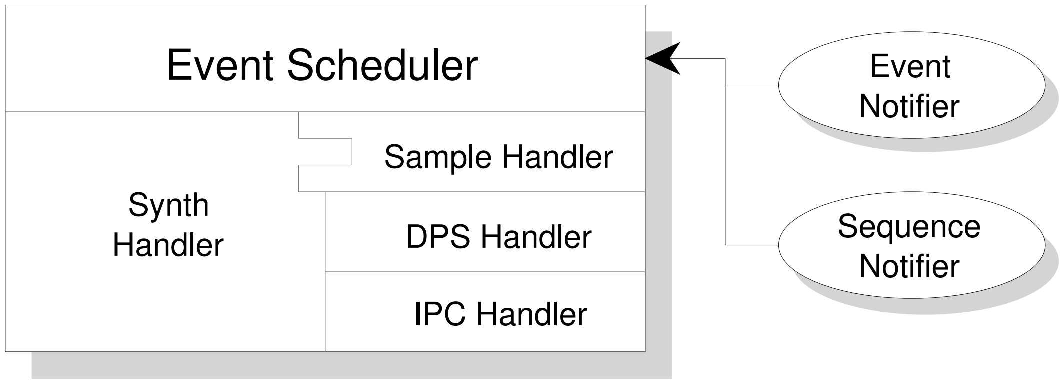

Figure 4: Scheduler Implementation Components

that the next event should be dispatched. The *sequence notifier* functions in the same way. These notifiers could be easily replaced using the interval timer and signal mechanisms of Unix.

The event scheduler portion of the main thread contains the event and sequence queues and operates on their contents in response to client requests and notifier messages. It is also responsible for initializing the set of media handlers and for dispatching events to the appropriate handler. Media handlers are hooked into the scheduler at two points. First, a table of event type identifiers is maintained. For each identifier, the table contains pointers to an initialization and a dispatch function. During initialization, the scheduler steps through this table, calling each initialization function in the order in which it finds them. When an event is dispatched, the event type is used as an index to locate the appropriate dispatch routine. Events are implemented as a header structure that identifies the type and contains a pointer to a separate block of type dependent data. The scheduler can therefore treat events uniformly, leaving all type dependent processing to the dispatch routines.

In addition to the obligatory initialization and dispatch functions, handlers have the option of extending the API with type specific requests. For example, the synth handler adds a routine to shift the pitch of a sequence. This involves adding a declaration to the Mig API specification and providing the corresponding function that will be called when the new operation is invoked by a client. Since they are invoked from within the main thread of the scheduler, these functions can operate on internal scheduler structures in whatever manner they choose. Most commonly, however, they are designed to process selected events from a sequence. The routine that shifts the pitch of a sequence, for example, modifies the frequency parameter of synth events and ignores all other event types.

Initialization, dispatch and API extension functions must be linked with the scheduler and are invoked from its thread. A more flexible design would allow new handlers to be loaded and terminated dynamically. However, although all of the handlers in the existing implementation share the main execution thread with the scheduler, the design admits other possibilities. For example, the initialization of a handler may involve creating a separate thread or process, establishing contact with an external server, and so on. The dispatch function would then act as a communications stub, forwarding the request to be handled either asynchronously or externally.

While there is considerable flexibility in utilizing external services, one should exploit it with caution. Recall that one of the design goals of the scheduler is to centralize all media presentation activity to facilitate more precise timing. The concept of modular media handlers was introduced to do away with the unpredicatable overhead of communicating data between processes. The use of threads, which execute asynchronously but share a common data space, represents a useful compromise. If a handler is added whose events initiate prolonged computation, it is be best to have the dispatch routine create a separate thread, allowing the scheduler to return to client requests while the new thread continues to process the event.

The event scheduler API is described in Section 5.4.

5.3 Media handlers

The previous section (5.2) outlined the way in which media handlers are integrated with the event scheduler. Here we provide details of each module in the existing implementation.

5.3.1 Synth handler

Synthesis algorithms are implemented in the Music Kit by composing various kinds of simple, low lever oscillators and combinators called *unit generators*. The network of unit generators that implements a particular algorithm is encapsulated in a *synth patch*. In terms of the Music Kit Objective-C classes, a tone is synthesized by giving a *Note* to a *SynthInstrument* that in turn realizes the note using a *SynthPatch*. Synth patches respond to three types of notes. A *note-on* initiates a continuous tone that is actively maintained by the synth patch, a *note-update* allows the characteristics of an active tone to be modified without interrupting it, and a *note-off* simply terminates the tone. Each tone is given a unique *note tag* that allows *note-update* and *note-off* events to be associated with the correct tone. *SynthInstrument* is a subclass of *Instrument*, whose other subclasses allow notes to be sent to MIDI devices, written to score files, and so on.

The synth handler manages synth patches directly, bypassing the more abstract Music Kit classes such as *Conductor*, *Instrument*, *Part*, *Score* and *Performer* that are intended mainly for musical performance. The *Orchestra* class deals with physical DSP resources, so it is retained.

The performance objectives of the synth handler design are to maximize DSP utilization and to minimize the delay in initiating, modifying or terminating a tone. The first objective is achieved by making the guarantee that a request will always be executed if sufficient DSP resources are available or can be made so without interrupting other active tones. Delays are reduced by maintaining pools of free resources rather than creating and terminating them on demand.

One additional criterion influences the design. Suppose that the DSP is currently operating near its capacity and that two new tone generation requests are received. One is for a

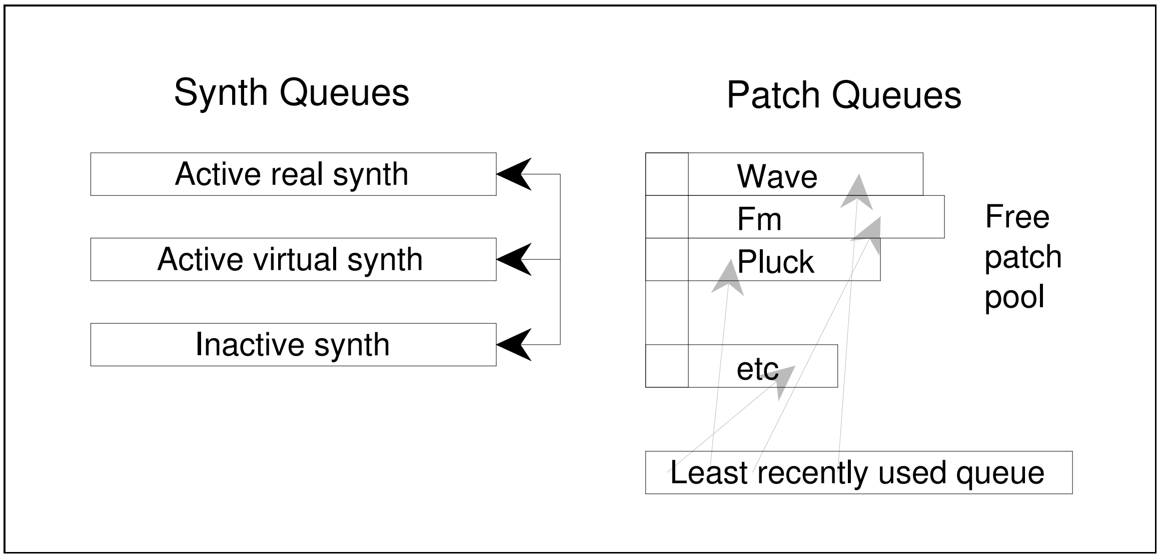

Figure 5: Internal Design of Synth Handler

tone that is to be terminated shortly after it begins and the other is for a tone of long and potentially unbounded duration. If only one request can be serviced, it might make sense to discard the short tone and initiate the longer one since this results in correct steady state behaviour. However, the duration of a tone is not specified (and may not even be known) at the time of its initiation, so it is not possible to implement this policy. To deal more accurately with this situation, the synth handler introduces the notion of a *virtual DSP* and provides the means to transfer active tones between it and the real one. This allows requests that cannot be immediately serviced by the real DSP to be simulated on the virtual one, properly applying update and termination requests as they are received. When DSP resources become available again, tones can be moved from the virtual DSP with their current set of parameters. Although it has not yet been tested, the same facility allows tones to be balanced between multiple DSPs when more than one is available.

Internal design

When synth patches are created they are allocated real DSP memory. Their existence is therefore bound to the availability of the DSP hardware. Implementation of the virtual DSP requires that synthesis elements of some sort remain intact when the DSP is relinquished and synth patches are deallocated. Tones are therefore represented internally by *synth nodes* that encapsulate their data. A synth node may optionally control a synth *patch* for use in rendering the note on the DSP. The virtual DSP is implemented simply by allowing synth nodes to function without active synth patches. Since synth nodes and patches are created for different reasons and at different times, they must be managed separately.

Figure 5 illustrates the internal structures of the synth handler. Three *synth queues* are used to manage synth nodes. A node is said to be active if the tone with which it was most recently associated has been initiated and not terminated. The tone will be audible if the node controls a synth patch and silent otherwise. The first queue contains the set of active nodes that control synth patches (i.e., those that are producing audible tones), while the second contains active nodes for which there are currently insufficient DSP resources. They can be regarded as the queues for the real and virtual DSPs respectively. Even though both queues contain active synth nodes, their contents are divided in this way to facilitate the movement of nodes from the virtual to the real queue. When a synthesis request is received for which there are insufficient resources, the new synth node is placed in the virtual DSP queue. When a node in the active queue becomes inactive, its synth patch can be given immediately to one of the virtual queue nodes and that node moved to the active real queue.

The inactive synth queue is used to maintain a pool of unused synth nodes. When a synthesis request is received, a synth node is taken from this pool, updated according to the details of the request, given a synth patch if one is available, and inserted into one of the active queues. Only when this pool is empty must a new node be created. Similarly, when a tone is terminated, the corresponding synth node can be returned to this pool without needing to free the resources it controls. Each synth node contains three Note objects (on, update and off) and an amplitude envelope object, so both initialization and termination are relatively expensive. The pool is initially empty, so creation of new nodes is at first relatively frequent. A steady state is soon reached, however, in which most new synthesis requests can be serviced by nodes from the pool.

Synth patch management

Management of synth patches is made somewhat more complicated by the fact that they come in several varieties. Recall that the network of unit generators encapsulated by a synth patch determines the synthesis algorithm that it implements. The synth handler manages three kinds of synth patches: *Wave*, a simple sinusoidal tone; *Fm*, a frequency modulation algorithm; and *Pluck*, a complex algorithm whose tones emulate plucked stringed instruments. Since each of these synth patch classes has different DSP memory requirements, the type cannot be changed without deallocating the original patch. It is therefore necessary to maintain a separate pool for each type of synth patch.

The portion of Figure 5 labelled "Free patch pool" shows the queues used for this purpose (there are only three in the existing implementation, but more can be easily added). Initially no synth patches have been created and these queues are empty. As synthesis requests are received, patches are created to service them. When a tone is terminated, it no longer requires DSP resources and its synth patch can then be returned to the free patch queue corresponding to its type. When subsequent synthesis requests arrive, patches of the required type can be taken from this pool.

If there are no free patches of the correct type to service a synthesis request, a new one is normally created. If there are no free patches of any type and insufficient DSP resources remain to create a new one, then the request cannot be allocated a patch and the corresponding synth node is placed in the virtual synth queue. However, if the right type of patch is not available and a new one cannot be created, but the pool contains unused patches of other types, then the option is available of freeing an unused patch, releasing DSP resources to be used in the creation of a patch of the required type. A decision must then be made as to what kind of patch to release. An inappropriate choice (as an arbitrary one is likely to be) may cause considerable inefficiency. Suppose that the DSP is capable of supporting n separate patches, and that $n-1$ patchs of type *Wave* have been allocated but are currently unused. Suppose further that two additional auditory streams, one using the *Pluck* class and the other using *Fm* are interleaved in such a way that requests for the two classes strictly alternate. Suppose finally that we arbitrarily decide to deallocate *Pluck* patches first, because they consume the most resources, followed by *Fm* and then *Wave* patches. When the first of the alternating requests is received, a new *Pluck* patch will be allocated, exhausting the remaining DSP resources. When the next request is received, the *Pluck* patch will be deallocated and replaced by an *Fm* patch. The next request will deallocate the *Fm* patch replacing it with a *Pluck* patch, and so on. Had we simply deallocated a *Wave* patch (or two) in response to the second request, patches of both types could have been retained and the overhead of patch creation and termination avoided.

To improve this behaviour, a *least recently used* patch queue is maintained. Each patch node contains two sets of queue pointers: one for the free patch queue of the corresponding type, and one for the *least recently used* queue. When a synth node becomes inactive, its patch node is returned to the appropriate free pool and also linked to the end of the *least recently used* queue. Then, when a decision must be made as to which unused patch to deallocate, the least recently used patch will be chosen from the head of the queue and unlinked from an arbitrary position in the free patch queue for its class.

Synth handler performance

The design decisions discussed here combine to give the synth handler an extremely high capacity. A single DSP is capable of sustaining roughly fifteen *Fm* tones, twenty *Wave* tones, or some intermediate combination. However, when tones are of relatively short duration, many more logical streams can be interleaved. Occasionally the streams will overlap in such a way as to produce a burst of activity that the DSP cannot handle, but the absence of a few voices among fifteen or twenty is rarely noticeable.

The performance of the synth handler has not been systematically evaluated, but informal experimentation suggests that forty or fifty auditory streams of the sort required by the application programs described below can be sustained on a single DSP without noticeable degradation. It appears that the limits of the scheduler will be reached before those of the synth handler. It is likely that rigorous performance evaluation would lead to many improvements in the internal design, but the existing implementation is already capable of producing auditory displays of far greater complexity than we currently require.

5.3.2 Other handlers

The original prototype of the event scheduler was designed specifically for sound synthesis, resulting in synthesis facilities that were not separated cleanly from other scheduling functions. The newer design (described in this report) provides a framework that allows different media types to share the same scheduling mechanism. However, most of our work still relies primarily on tone synthesis. The remaining media handlers are very simple and were implemented more to test the new framework than for serious use at this time.

Sample handler

The sample handler is used to play digital audio samples and works with *Sound* objects that are defined and implemented in the NeXTStep Sound Kit. They can either be transferred directly from the client as a block of type dependent data, or can be initialized from a file whose name is provided by the client. The former approach is somewhat platform dependent since the data is communicated in its raw form, while the latter assumes that the server has access to the desired file.

A sound event is dispatched simply by instructing the Sound object to play itself. The details of forwarding the digital sample data to the DSP hardware are handled by the Sound Kit. However, the sound cannot be played if the DSP has been claimed by the Music Kit for synthesis purposes. As part of the dispatch process, the sample handler must therefore preempt the DSP if required (the Music and Sound Kits do not automatically cooperate in this way). It does this through a pair of API extensions provided by the synth handler. If necessary, the synth handler is requested to relinquish the DSP and move its active synth patches to its virtual DSP queue. When the sound is finished the DSP is returned to the synth handler.

Although this approach can result in the frequent and nearly complete interruption of a complex auditory display, it still provides more seamless interaction between the Music and Sound Kits than was previously possible.

DPS handler

The DPS (Display PostScript) handler forwards arbitrary Display PostScript code to the NeXTStep window server in the context of a window owned by the scheduler. It is possible to give the scheduler access to the windows owned by client applications, but the technical details are too cumbersome to warrant implementation as part of this simple proof of concept. Existing applications that synchronize graphical and auditory display do so by utilizing the signal mechanism described in Section 4.3 and doing their own drawing upon receipt of each signal.

To achieve the goals discussed in the introduction, the scheduler would need to be more tightly integrated with media presentation facilities such as the window server. Ideally, the window server, Music Kit, Sound Kit, and so on would all be implemented as statically linked modules of the event scheduler. This would introduce an additional delay in the presentation of unsynchronized material, but the simplicity of the scheduling facilities would make the delay minimal. The benefits of having a centralized synchronization mechanism would in most cases outweigh the disadvantages imposed by this delay.

Video handler

The video handler is used to control an external laser disc player connected to a serial port. The type dependent event data contains the specific laser disc command and any relevant parameters. It can therefore be used to control any feature offered by the player. Although laser disc technology supports random frame access, a significant delay is sometimes incurred if the read head must seek past many tracks. This kind of device specific information could be incorporated in the video handler to ensure that segments are cued sufficiently far in advance. The existing implementation, however, does not yet do this.

IPC handler

The IPC (interprocess communication) handler allows arbitrary messages to be forwarded to external processes. The type specific data contains an internet address, port number and byte stream. The handler attempts to open a TCP/IP connection with that address and, if successful, sends the byte stream across the connection. It does not currently attempt to relay a reply back to the original client. This facility is separate from the signal that can be associated with any event, although it is closely related. The main difference is that signal values are restricted to integers while the IPC handler can forward arbitrary messages.

5.4 Client libraries

Two client libraries have been produced for use in making requests of the event scheduler. The first is implemented in standard ANSI C, providing access to the scheduler through a function call interface. Much of the code in this library is actually generated from an interface specification file using the Mach interface generator. The second library builds on the first, using Objective-C to construct a class hierarchy based on the more important scheduler abstractions. The function call library is both versatile and efficient, but it is awkward to use directly. For example, type dependent event data of arbitrary size are handled by separating them from the standard event header and giving their size explicitly. To send the type dependent data of an event sequence, which may contain an arbitrary number of events each with an arbitrary amount of additional data, the type dependent portions are concatenated and passed to the API function as a single block. The Objective-C library, on the other hand, allows application programs to work with more natural abstractions such as events, sequences, scheduling and dispatch. The class implementations handle the details of contacting the scheduler, martialling data for communication, providing additional buffer space and so on. Section 5.4.2 outlines the services provided by each Objective-C class.

5.4.1 Standard C library

The standard C client library implements the basic event scheduler API. There are four categories of server request: internal notification, sequence scheduling, sequence schedule modification, and miscellaneous operations. Standard scheduler services and API extensions provided by specific media handlers are both included. The first argument of every routine identifies the server port to be contacted, and each function returns a value of type `kern_return_t` indicating success or failure of the call. Other return values are handled by pass-by-reference arguments.

The following sections introduce API requests in each category, giving the complete calling sequence and a brief description of the operation. This should not however be regarded as a complete or authoritative reference manual since minor changes in the implementation may be introduced as the need arises.

Internal notification

The internal notifier threads signal the server using two reserved API functions.

processEvents `( port_t server )`

This request is reserved for use by the event notifier thread. It signals the server that the next event in the event queue should be dispatched, as described in Section 4.2.

processSchedule `( port_t server )`

This request is reserved for use by the sequence notifier thread. It signals the server that the next sequence in the sequence queue should be scheduled, as described in Section 4.2.

Sequence scheduling

Two calls are provided for dispatching sequences immediately or scheduling them for later or repeated dispatch. There are no corresponding requests for the dispatch or scheduling of individual events; a single event is dispatched or scheduled by encapsulating it in a sequence of length one and passing it to one of these functions.

dispatchSequence `( port_t server, ES_Sequence sequence,`
`unsigned int sequenceCnt, ES_Data data, unsigned int dataCnt,`
`port_t signalPort )`

Schedule the given sequence immediately, as described in Section 4.2. The sequence events are inserted into the event queue immediately, bypassing the sequence queue. The sequence is specified through the *sequence* structure and and its size in bytes given by *sequenceCnt*. Type dependent event data are concatenated into a single block and specified by the *data* and *dataCnt* values. Each event in a sequence can independently specify a signal value and a port to which it should be sent when the event is dispatched. It is sometimes convenient to be able to override the signal destination, however. If a positive value is given for the *signalPort* argument, that value is copied to the signal port entry of each event before the event is queued.

scheduleSequence `( port_t server, ES_Sequence sequence,`
`unsigned int sequenceCnt, ES_Data data, unsigned int dataCnt,`

`int delay, int reps, int period, int sync,`
`port_t signalPort, ScheduleHandle *item )`

Insert the given sequence into the sequence queue for subsequent scheduling. The sequence is specified by the *sequence* and *sequenceCnt* values, and the type specific event data for each event in the sequence is concatenated into a single block and identified by *data* and *dataCnt*. The initial delay in milliseconds is given by *delay*, *reps* indicates the number of repetitions to be scheduled (a value less than zero indicates unbounded repetition), and *period* is the scheduling period of the sequence. If *sync* is non-zero, the initial delay is adjusted as indicated by equation 1 (page 10) to match the phase of the schedule with a standard chosen arbitrarily by the server. If *signalPort* is a positive value, that value is copied to the signal port entry of each event before the sequence is queued. A unique identifier is generated for the newly scheduled sequence and is returned through the *item* argument.

Modifying scheduled sequences

Once a sequence has been queued, its parameters and those of its events can be modified without removing it from the sequence queue or otherwise disturbing its periodic scheduling. Most media handler specific API extensions are added in this category.

scaleSequenceTiming `( port_t server, ScheduleHandle sequence,`
`double eventFactor, double periodFactor, int mode )`

Scale the period and event time stamps of a sequence. The period of the given *sequence* is scaled by *periodFactor* and the time stamp of each of its events is scaled by *eventFactor*. If *mode* has the value `ES_RELATIVE`, then the current values are multiplied by the given factors. If it has the value `ES_ABSOLUTE`, then the new values are computed as the product of the original unscaled values and the given factors. This mechanism could be implemented by storing a copy of the original period or time stamp. In the existing implementation, however, a separate scaling factor is maintained and the original time stamp is multiplied by that scaling factor to determine the new time stamp. With this arrangement, the relative mode multiplies the new factor into the existing one while the absolute mode simply replaces the old factor by the new.

syncSequences `( port_t server, ScheduleHandle sequence1,`
`ScheduleHandle sequence2 )`

Synchronize the given pair of sequences by replacing the larger sequence queue time stamp with the smaller one and reinserting the modified sequence in the correct position. A planned change will provide greater flexibility by adjusting the second sequence time stamp to match the first. Multiple sequences are synchronized by repeated calls, a process that is automated by the EventSequence class described below.

shiftSequence `( port_t server, ScheduleHandle sequence, int amount )`

Change the phase of the given *sequence* by adjusting its sequence queue time stamp by *amount* milliseconds and reinserting it in the correct position.

unSchedule `( port_t server, ScheduleHandle item )`
: Terminate a scheduled *sequence* by removing it from the sequence queue and freeing any associated scheduler resources.

scaleSequenceAmp `( port_t server, ScheduleHandle sequence, double factor, int mode )`
: This is a synth handler extension to the API. It scales the amplitudes of every synth event in the given *sequence* by *factor*. The value of *mode* determines whether the scaling is relative or absolute, as described for the scaleSequenceTiming entry above.

scaleSequenceFreq `( port_t server, ScheduleHandle sequence, double factor, int mode )`
: This is a synth handler extension to the API. It scales the frequencies of every synth event in the given *sequence* by *factor*. The value of *mode* determines whether the scaling is relative or absolute, as described for the scaleSequenceTiming entry above.

Miscellaneous operations

Requests that control the over all behaviour of the scheduler or that do not affect specific events or sequences are grouped together in this category.

newEventTag `( port_t server, int *tag )`
: Acquire a unique event tag from the server.

setDebugLevel `( port_t server, int level )`
: Set the current debugging level to the given value. A value of zero generates no output, a value of one produces a trace of every scheduler request (both name and arguments), and values greater than one generate additional information as appropriate for specific functions. Note that this debugging mechanism is independent of the *showState* request, described next.

showState `( port_t server, int level )`
: Display the internal state of the server, according to the given debugging level. Higher levels generate more information. Currently, basic information including the server port, its start time, current debug level and current event tag are always shown. Levels greater than zero will show the state of the synth handler at a corresponding level (a more complete implementation would require that every media handler register a state display function – currently only the synth handler is included). Levels greater than one will display the contents of the sequence queue and levels greater than two will include the contents of the event queue.

resetServer `( port_t server )`
: Reset the event scheduler, emptying its queues and resetting its timers.

terminateActive (`port_t server`)
: This is a synth handler extension to the API. It simply terminates the active synth patches (a useful operation since note-off events sometimes get lost during development).

terminateSchedules (`port_t server`)
: Terminate all active sequences by emptying the sequence queue and freeing the associated resources.

addDSP (`port_t server`)
: Claim the DSP and move active synth nodes from the virtual DSP queue to the real queue as long as DSP resources remain (as described in Section 5.3.1).

removeDSP (`port_t server`)
: Move all active synth nodes to the virtual DSP queue and relinquish the real DSP (as described in Section 5.3.1).

5.4.2 Objective-C class library

The classes in this library do not extend the basic functionality of the event scheduler, but they make it much simpler to communicate with it and automate a number of useful procedures. They permit an application program to concern itself with the familiar abstractions of events, sequences and scheduling rather than with the details of compiling type dependent data blocks and encapsulating individual events within sequences. For example, the statement

```
[[[Tone alloc] init] dispatchSelf];
```

will cause a 440.0 Hz tone (A) to be generated for a period of one second, provided that the event scheduler is running, without any additional initialization or processing. Additional *Tone* methods allow its parameters to be adjusted as required.

Each event class is discussed briefly below, providing a good overview of the capabilities of the class library. However, this section is not meant to be a complete reference manual for the library.

The library can certainly be used on its own, overriding defaults as necessary to produce the desired events. It is most useful, however, when used in conjunction with the event modeller (Section 5.5).

Event

The *Event* class provides the basic unit of currency. An *Event* object can be used directly to register a signal with the scheduler, since a signal can be associated with every kind of event, but it cannot be used to generate additional presentation. A subclass of *Event* is provided for each media handler to coordinate the manipulation of type dependent data and procedures. *Event* itself provides the scheduling and dispatch routines that are common to all event types. Every event object can be given a unique name for use with event files.

EventSequence

An *EventSequence* is used to assemble a collection of *Events* and handle the details of martialling type dependent event data for scheduling or immediate dispatch. They can be created and modified programmatically, but it is more common to build a sequence using the event modeller (Section 5.5), saving it to an *EventFile*. This file can then be used to initialize an *EventSequence* object that can be dispatched simply by invoking its `dispatchSelf` method, or can be scheduled for delayed or repeated execution with the appropriate variant of the `scheduleWithDelay` method. Every sequence object can be given a unique name for use with event files.

SynthEvent

A *SynthEvent* is a subclass of *Event* that manages the additional parameters associated with tone synthesis. It offers methods for accessing and modifying all of the parameters associated with the three supported synthesis methods (Wave, Fm and Pluck, Section 5.3.1), but its scheduling and dispatch capabilities are inherited from *Event* and remain unmodified.

Tone

Synthesized tones can be of unbounded duration, allowing *note-on* events to be dispatched without a corresponding *note-off*. When the duration of a tone is known in advance, however, the *Tone* class provides a more convenient way to generate it. A *Tone* object combines two *SynthEvents*, one a *note-on* event and the other a *note-off*, in an *EventSequence* in which the relative time stamp of the first event is zero and the time stamp of the second is the duration of the tone. The tone is dispatched or scheduled by requesting the corresponding service of the *EventSequence* it contains.

SampleEvent

Digital audio samples are controlled by the *SampleEvent* subclass of *Event*. It can be provided with sound data in one of three ways: a *Sound* object (defined by the Sound Kit) can be passed directly, initialized from a given file, or the file name itself can be forwarded to the event scheduler which will create and initialize its own object. The class is very simple, providing only the means to query or set the source of sound sample data. All other scheduling functions are inherited from *Event*.

DPSEvent

The *DPSEvent* subclass of *Event* is a minimal implementation that allows the DSP handler of the event scheduler (Section 5.3.2) to be tested. Its type dependent data consists of a single text string that is intended to contain valid PostScript code. The scheduling methods inherited from *Event* handle the details of forwarding the type dependent data to the scheduler; *DPSEvent* implements methods only for setting and querying the text string.

IPCEvent

Like *DPSEvent*, *IPCEvent* is a minimal implementation that allows the IPC handler of the event scheduler (Section 5.3.2) to be tested. It allows an address and a byte stream to be set; the methods inherited from *Event* forward this data to the scheduler as part of an IPC operation request.

EventFile

The *EventFile* class coordinates the archiving of the classes described above. Each class is responsible for implementing `readFromFile` and `writeToFile` methods that invoke the corresponding method in the parent class then read or write the subclass dependent data in a human readable form. A collection of events and sequences can be combined in an *EventFile* object, then archived in one operation by sending the object a `saveTo:file` request. Similarly, a new *EventFile* object can be initialized with the statement

`[[EventFile alloc] initFromEventFile:`*filename*`];`

Once created or initialized, the component events and sequences can be retrieved by name, modified, scheduled, and so on.

The modeller application, described below in Section 5.5, is intended primarily for the interactive construction and editing of event files. A complex auditory display can be easily incorporated into an application by having it initialize an *EventFile* object from a file created by the modeller, then sending its components requests to schedule or dispatch themselves with the appropriate timing parameters.

SignalView

The *View* class of the NeXTStep Application Kit is used generate custom drawing classes. It manages the details of contacting the window server, establishing a local coordinate system, providing default responses to mouse and keyboard events, negotiating drag and drop protocols with other views and applications, and so on. The Application Kit allows an application to register ports and timers with its event handling service. When the timer expires or a message is detected at the port, the Kit generates a corresponding event in the event queue for the application which can then respond in whatever fashion it chooses.

The *SignalView* class is a subclass of *View* that contacts the event scheduler during its initialization, and registers the scheduler port with the Application Kit. This port can then be specified when a sequence is scheduled or dispatched. The result is that each time an event from the sequence is dispatched, the *SignalView* object detects the resulting signal and invokes a default response (which is to do nothing). Like *View*, *SignalView* is meant to be subclassed; a subclass would then override the signal response method to provide its own response. The response is usually to move to the next step in a display sequence or to use the signal value to calculate new geometry or colour. The ability to specify a signal port when scheduling a sequence, rather than using the default port recorded in each event, was introduced to allow greater flexibility in directing signal streams to different *SignalViews*.

EventScheduler

The *EventScheduler* class has not yet been implemented, although it is required to make the event class library complete. The port used for communication with the event scheduler must currently be created by an application that wishes to use it. It is then shared among event class objects by means of a global variable. Furthermore, the miscellaneous scheduler operations introduced on page 24 are not encapsulated in any existing event class.

The *EventScheduler* class is needed to address these oversights. Any application requiring the services of the event scheduler would first initialize an *EventScheduler* object. This would initiate contact with event scheduler, launching it if it is not currently running. The applications that use the scheduler do not currently take this last step, choosing instead to terminate if the scheduler cannot be located, since the code to properly locate and launch the executable is somewhat involved and would have to be replicated for each application. Providing this ability in a library class would be a significant improvement.

The initialization sequence of each event class should then be modified to fetch the scheduler port handle from the *EventScheduler* object (which can name itself in a private application name space using a facility provided by the Application Kit), doing away with the need for a global variable. Additional methods would export the miscellaneous scheduler operations such as DSP and debugging control. Finally, the class should provide the means of terminating and re-establishing contact with the scheduler, allowing applications to continue if the scheduler crashes and must be restarted (this would of course be especially useful during development).

5.5 Event modeller

The basic function of the *event modeller* has been introduced in previous sections. In summary, the modeller provides a graphical interface through which one can construct and test events and sequences of arbitrary complexity. The results can then be saved to an event file that applications can access using the *EventFile* class of the Objective-C client library (Section 5.4.2). Existing files can be loaded and modified, and multiple files can be edited simultaneously. This latter features enables a limited template mechanism. Useful sequence templates can be constructed and saved in template files. These can later be loaded and sequences or individual events can be copied from the template file and pasted into another file for a customized application.

Figure 6 shows the panel used to construct and display event files. The implementation lacks such conveniences as palettes of events and sequence types, but as a prototype project it serves its purpose adequately. Objects are currently divided into three categories: events, tones and sequences. Selecting a category displays the items it contains. When a sequence or tone is selected, its components are shown in the third column of the browser. New objects can be added or deleted with the buttons provided, and renamed using the text field. The type of object added depends on the item currently selected. For example, if the sequence category is selected, but no particular sequence has yet been chosen, an add request will produce a new sequence with a default name. If a particular sequence is then chosen and

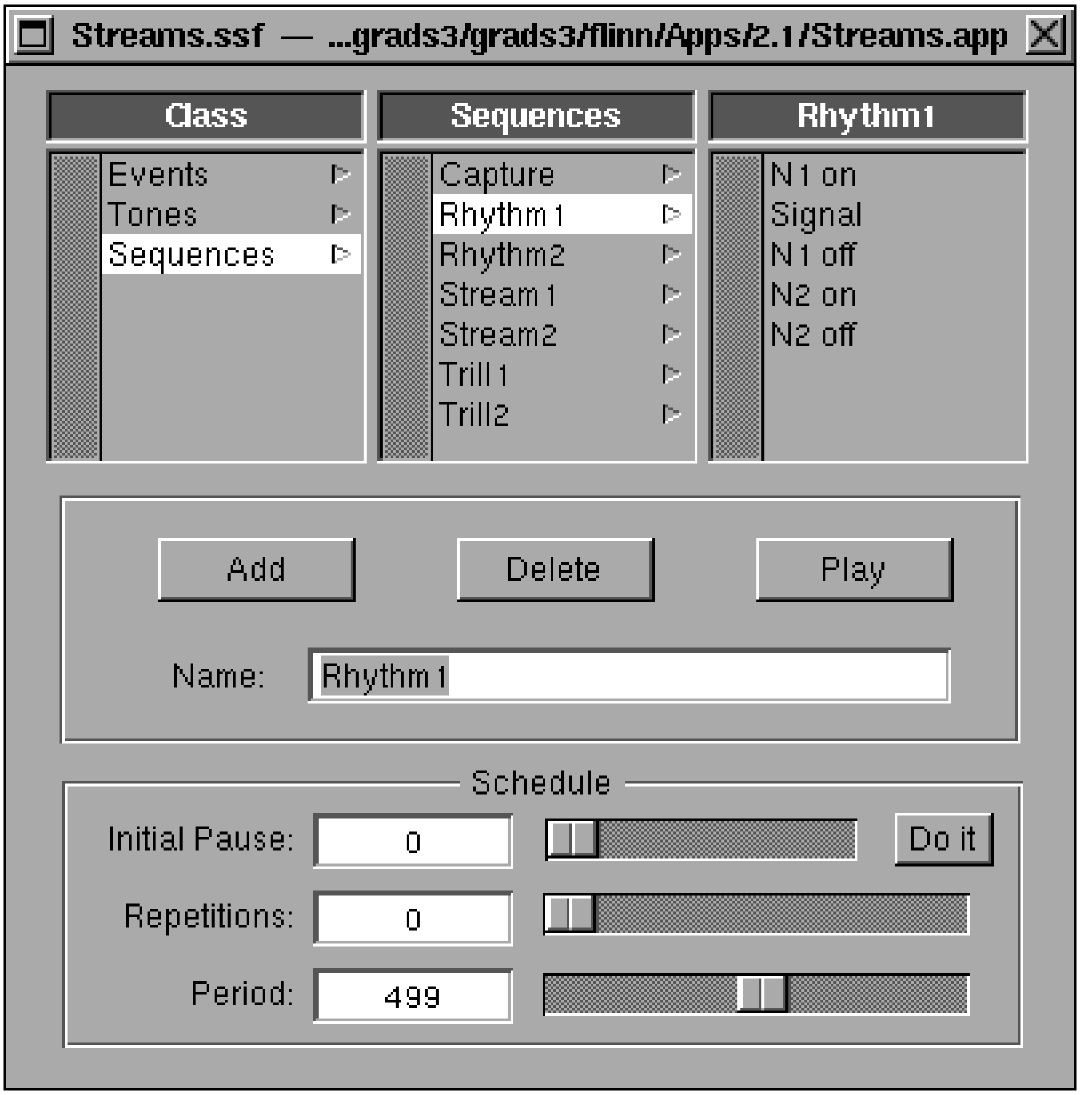

Figure 6: Event File Construction and Display

another add request is made, an event will be added to the selected sequence. The name assigned here is used in retrieving specific events or sequences from an *Eventfile* object.

The *Play* button requests immediate dispatch of the selected tone, event or sequence. The controls at the bottom of the window allow a sequence or event to be registered with the scheduler for repeated presentation. The actual period of the selected sequence is provided as the default in this portion of the display, but can be adjusted as necessary.

Once a sequence has been registered, various operations can be performed on it using the controls shown in Figure 7. Currently active sequences are shown in the scroller at the top right of the window, and may be selected either individually or in combination. The example shown in the figure has two active rhythmic sequences (created for the auditory streaming application described in Section 5.7). They can be synchronized by selecting them both and pressing the *Sync* button. The *Remove* button terminates the sequence scheduling, and the *Commit* and *Revert* buttons are used to save the modifications or revert to the original sequence parameters. The remaining controls allow the timing of a sequence to be shifted or scaled, and the pitch and amplitude of a sequence's synth events to be adjusted (the fact that these last two synth handler specific operations are hard coded into the general sequence operations panel reflects the scheduler's modest beginnings as a monolithic sound server). When the timing and period lock switch is selected, the timing and period scale factors are constrained to be equal. This ensures that the events will occupy the same proportion of the sequence, preventing such things as sequence overlapping on repeated iterations. Finally, sequences are always registered with a signal port owned by the modeller, even if the events are created with different port identifiers. This allows the signal value field at the bottom of the panel to display the last value received.

Events are modified using an event inspector panel. Since events can be of various types, different inspectors are required, each sharing the same basic event information. However, the current implementation provides only a synth event inspector. The framework exists for providing additional inspectors, but they have not yet been added. Figure 8 shows the collection of controls used to maintain the timing parameters and type information of a synth event. Figure 9 shows the controls for the remaining synthesis parameters. Not all parameters are used for all synthesis algorithms; those that are unneeded are ignored. The *Test* button at the top of Figure 8 is useful for adjusting the parameters of a tone. When the button is pressed, a tone is initiated according to the current selection of parameters. As the parameters are subsequently adjusted, the tone is immediately updated to reflect the changes. When the tone sounds right, it can be disengaged (using the *Test* button again) and the configuration saved to the *SynthEvent*.

The most significant deficiency of the current modeller implementation is that sequence timing must be crafted by editing the time stamp value of each event directly using the event inspector. This process can be time consuming, and confusing for complex sequences. It is easy to imagine a direct manipulation sequence editor in which events are dragged from a palette or from the event file display onto a time line representing the sequence. The rough timing of a sequence could be produced very quickly, and the scheduler's sequence time scaling operations could be used to fine tune the timing as the sequence was repeatedly

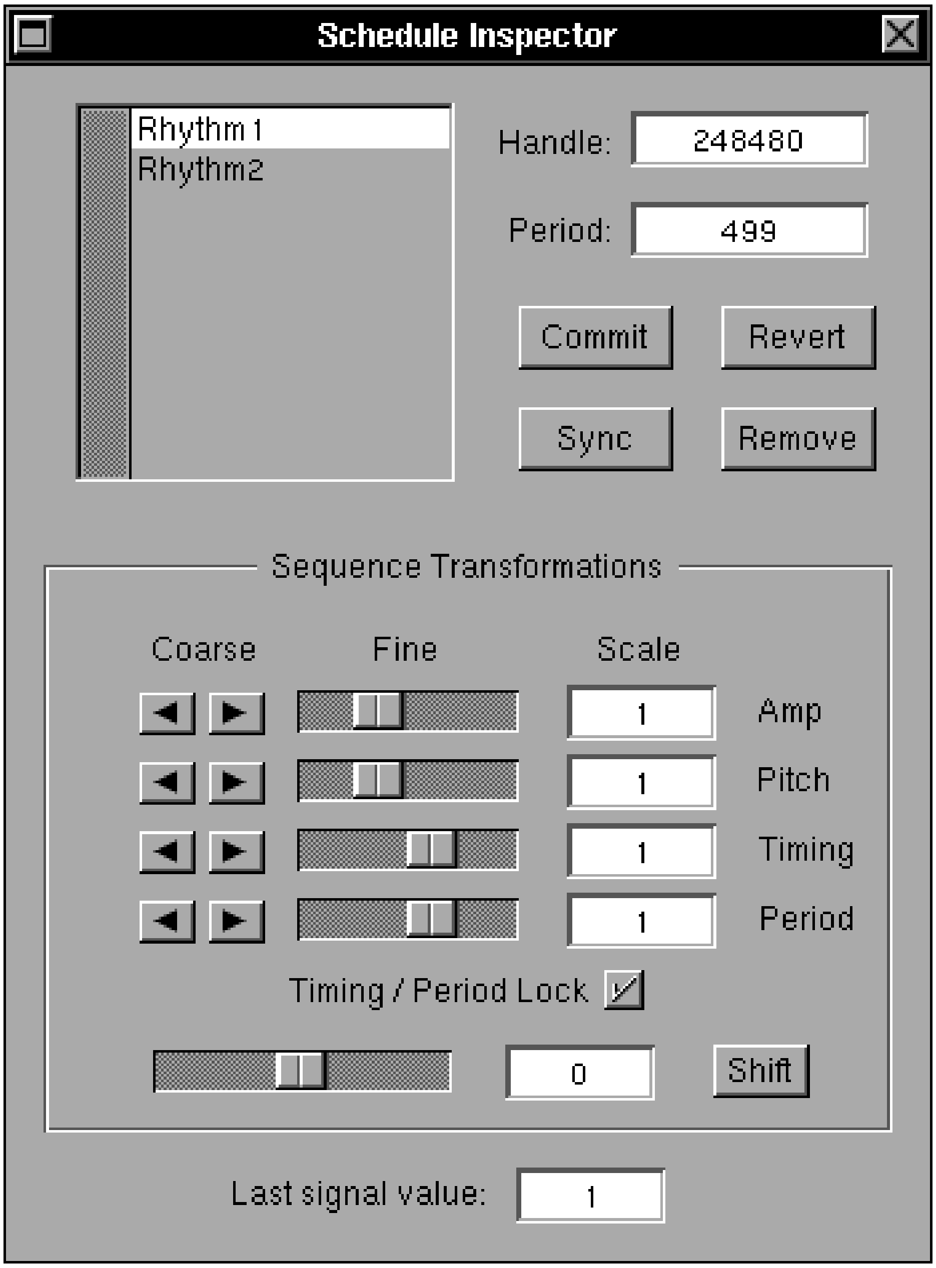

Figure 7: Schedule Inspector Controls

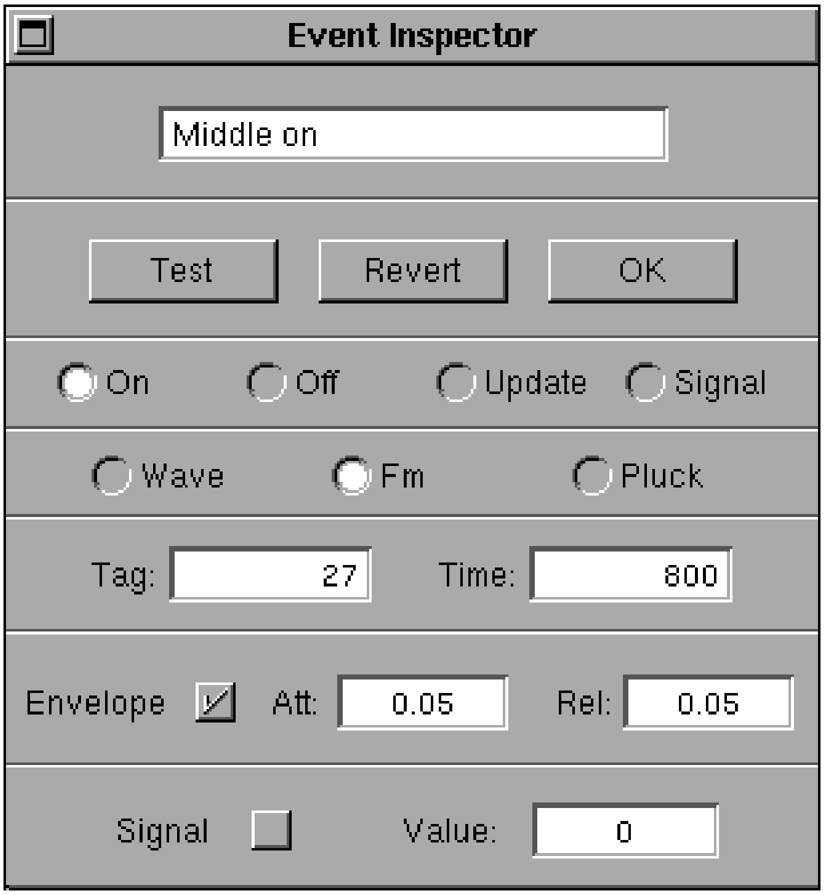

Figure 8: Event Inspector Controls

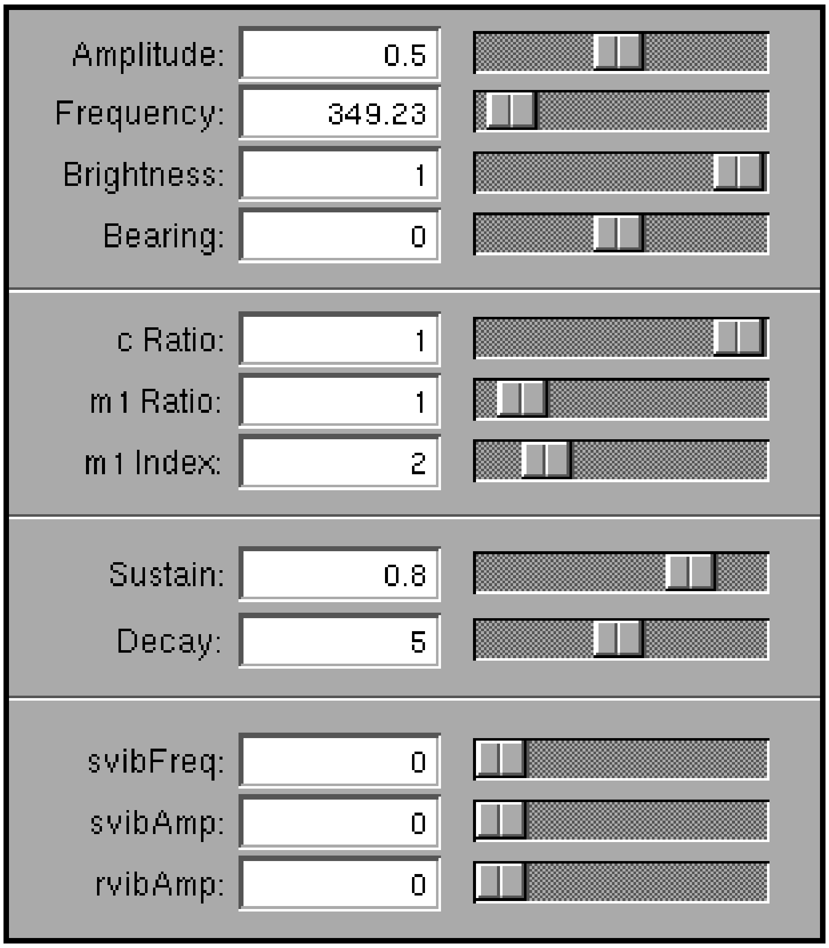

Figure 9: Synth Inspector Controls

presented. A stand alone prototype of such a sequence editor has been partially implemented.

5.6 Command line tools

The facilities of the interactive modeller are sufficient for most purposes, but it is occasionally useful to be able to build scripts of operations to be applied in a given sequence. This ability has proven useful, for example, in stress testing the scheduler. A small collection of command line tools has been constructed for this purpose. In most cases the tool is simply a command line wrapper around a client library event class that initializes an object and invokes a specific method with a set of parameters taken from the command line.

esState `<value>`
: Print the current server state at the level specified by *value*. A value of zero generates a terse summary; a value of one includes the synth handler status; a value of two displays most queue contents; and a value of three includes the event queue.

esDebugLevel `<value>`
: Change the debugging level to the given value. A value of zero turns off debugging output; a value of one produces a trace of scheduler requests; and higher values generate additional function specific information.

esReset (*no arguments*)
: Reset the server, emptying its queues and resetting its timers.

sequence `<event file> <delay> <reps> [<sync>]`
: Read a sequence from the given event file and register it with the event scheduler using the given delay, repetition count and optional synchronization mode.

unSchedule `<sequence handle>`
: Remove the sequence identified by the given handle from the sequence queue.

noteOn `<frequency>`
: Initiate a tone of the given frequency.

noteOff `<note tag>`
: Terminate the tone identified by the given tag.

esAddDSP (*no arguments*)
: Claim the DSP, moving active synth nodes from the virtual queue to the real one as described in Section 4.2.

esRemoveDSP (*no arguments*)
: Move active synth nodes from the real synth queue to the virtual one, as described in Section 4.2, and relinquish the DSP.

5.7 Interactive applications

A number of interactive applications have been constructed using these facilities (their purpose and implications are discussed at some length in a separate report [5]). Implementations of Shepard's Tones [15] and of the Tritone Paradox [4] utilize the ability of the synth handler and Music Kit to synthesize complex spectra of arbitrary description to create auditory illusions in which relative pitch discrimination cues have been eliminated. A third application reproduces a series of effects, as summarized by Bregman [2], to demonstrate the phenomenon of *auditory streaming* in which logically independent auditory streams can be made to fuse into a single percept, or to divide into separate ones, by making small adjustments to salient acoustic parameters. This implementation effort was extremely effective in demonstrating the need for greater software support in the creation of a rich acoustical environment. A fourth application provides an experimental "workshop" for exploring the parameters within which the synchronization of sound and image creates the illusion of *visual capture* (that is, that the actions of the image appear to create the sound).

In each case, the event sequences were created using the modeller and imported into the application. The visual capture application makes extensive use of the signaling feedback facility of the scheduler.

6 Conclusions

This report has presented the motivating factors, requirements and design of an *event scheduler* that facilitates the synchronization of independent, heterogeneous media streams. The design supports the following principal features:

- data is delivered to the scheduler prior to actual dispatch, reducing overhead at the time of dispatch;
- periodic sequences of events can be accurately scheduled without the need to deliver data for each repetition;
- modular media handlers mediate hardware resource contention and allow the scheduler to be extended to support new media and hardware device types; and
- long term synchronization with the hardware clock is guaranteed, even when the scheduler capacity is temporarily exceeded on platforms lacking real time system support.

The scheduler has been used successfully in support of several interactive applications.

This work provides a high level framework for coordinating system resources, although the current implementation leaves considerable room for utilizing existing solutions more effectively. New media synchronization and scheduling algorithms can easily be incorporated in additional media handler modules, but a mechanism for coordinating module interaction is needed.

The scheduler itself also leaves room for extension and improvement. For example, additional scheduling operations will be suggested as new applications are designed. Perhaps the most promising area for improvement is to provide the scheduler with a scripting language through which procedural specifications, utilizing primitives provided by individual media handlers, may be executed according to a given schedule. Simple periodic event sequences offer considerable flexibility, but lack features such as conditional execution and automatic modification of event parameters following each iteration.

References

[1] Barry Arons. Tools for Building Asynchronous Servers to Support Speech and Audio Applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pages 71–78, Monterey, California, November 15-18 1992. ACM Press.

[2] Albert S. Bregman. *Auditory scene analysis: the perceptual organization of sound.* MIT Press, Cambridge, Mass., 1990.

[3] Andrew W. Davis and Joe Burke. Under the Hood: The Mac Goes to the Movies. *Byte*, 18(2):225–230, February 1993. A detailed look at Apple's QuickTime architecture.

[4] Diana Deutsch. The tritone paradox: Effects of spectral variables. *Perception & Psychophysics*, 41(6):563–575, 1987.

[5] Scott Flinn and Kellogg S. Booth. The Creation, Presentation and Implications of Selected Auditory Illusions. Technical Report 95-15, Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4, July 1995. Available electronically as http://www.cs.ubc.ca/tr/1995/TR-95-15.

[6] Jim Gemmell and Stavros Christodoulakis. Principles of Delay-Sensitive Multimedia Data Storage and Retrieval. *ACM Transactions on Information Systems*, 10(1):51–90, 1992.

[7] F. Horn and J. B. Stefani. On Programming and Supporting Multimedia Object Synchronization. *Computer Journal*, 36(1):4–18, 1993. Special issue on distributed multimedia systems.

[8] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart. AudioFile: A Network-Transparent System for Distributed Audio Applications. In *Proceedings of the Summer 1993 USENIX Conference*, pages 219–236, Cincinnati, Ohio, June 21-25 1993. USENIX.

[9] Thomas D. C. Little and Arif Ghafoor. Synchronization and Storage Models for Multimedia Objects. *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, April 1990.

[10] Steven R. Newcomb, Neill A. Kipp, and Victoria T. Newcomb. The "HyTime" Hypermedia/Time-based Document Structuring Language. *Communications of the ACM*, 34(11):67–83, November 1991.

[11] Nuno M. Guimarães and Nuno M. Correia and Telmo A. Carmo. Programming Time in Multimedia User Interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pages 125–134, Monterey, California, November 15-18 1992. ACM Press.

[12] S. Ramanathan and P. V. Rangan. Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems. *Computer Journal*, 36(1):19–31, 1993. Special issue on distributed multimedia systems.

[13] P. V. Rangan, S. Ramanathan, H. M. Vin, and T. Kaeppner. Techniques for Multimedia Synchronization in Network File Systems. *Computer Communications*, 16(3):168–176, March 1993.

[14] Robert W. Scheifler and James Gettys. *X Window System.* Digital Press, Bedford, MA, 3rd edition, 1991.

[15] Roger N. Shepard. Circularity in Judgements of Relative Pitch. *The Journal of the Acoustical Society of America*, 36(12):2346–2353, 1964.

[16] Ralf Steinmetz. Synchronization Properties in Multimedia Systems. *IEEE Journal on Selected Areas in Communications*, 8(3):401–412, April 1990.