Eric D. Scheirer and Barry L. Vercoe
Machine Listening Group
E15-401D MIT Media Laboratory
Cambridge, Massachusetts 02139-4307, USA
eds@media.mit.edu
bv@media.mit.edu

SAOL: The MPEG-4 Structured Audio Orchestra Language

Since the beginning of the computer music era, tools have been created that allow the description of music and other organized sound as concise networks of interacting oscillators and envelope functions. Originated by Max Mathews with his series of "Music N" languages (Mathews 1969), this *unit generator* paradigm for the creation of musical sound has proven highly effective for the creative description of sound and widely useful for musicians. Languages such as Csound (Vercoe 1995), Nyquist (Dannenberg 1997a), CLM (Schottstaedt 1994), and SuperCollider (McCartney 1996b) are widely used in academic and production studios today.

As well as being an effective tool for marshalling a composer's creative resources, these languages represent an unusual form of digital audio compression (Vercoe, Gardner, and Scheirer 1998). A program in such a language is much more succinct than the sequence of digital audio samples that it creates, and therefore this method can allow for more dramatic compression than traditional audio coding. The idea of transmitting sound by sending a description in a high-level synthesis language and then performing real-time synthesis at the receiving end, which Vercoe, Gardner, and Scheirer (1998) term *structured audio,* was suggested as early as 1991 (Smith 1991). A project at the Massachusetts Institute of Technology (MIT) Media Laboratory called NetSound (Casey and Smaragdis 1996) constructed a working system based on this concept, using Csound as the synthesis engine, and allowing low-bit-rate transmission on the Internet. If it were possible to create a broad base of mutually compatible installed systems and musical compositions designed to be transmitted in this manner, this technique could have broad utility for music distribution.

Computer Music Journal, 23:2, pp. 31–51, Summer 1999
© 1999 Massachusetts Institute of Technology.

The Motion Pictures Experts Group (MPEG), part of the International Standardization Organization (ISO) finished the MPEG-4 standard, formally ISO 14496, in October 1998; MPEG-4 will be designated as an international standard and published in 1999. The work plan and technology of MPEG-4 represent a departure from the previous MPEG-1 (ISO 11172) and MPEG-2 (ISO 13818) standards. While MPEG-4 contains capabilities similar to MPEG-1 and MPEG-2 for the coding and compression of audiovisual data, it additionally specifies methods for the compressed transmission of synthetic sound and computer graphics, and for the juxtaposition of synthetic and "natural" (compressed audio/video) material.

Within the MPEG-4 standard, there is a set of tools of particular interest to computer musicians called Structured Audio (Scheirer 1998, 1999; Scheirer, Lee, and Yang forthcoming). The MPEG-4 Structured Audio tools allow synthetic sound to be transmitted as a set of instructions in a unit-generator-based language, and then synthesized at the receiving terminal. The synthesis language used in MPEG-4 for this purpose is a newly devised one called SAOL (pronounced "sail"), for Structured Audio Orchestra Language. By integrating a music-synthesis language into a respected international standard, the required broad base of systems can be established, and industrial support for these powerful capabilities can be accelerated. The sound-synthesis capabilities in MPEG-4 have a status equivalent to the rest of the coding tools; a compliant implementation of the full MPEG-4 audio system must include support for real-time synthesis from SAOL code.

In this article, we describe the structure and capabilities of SAOL. Particular focus is given to the comparison of SAOL with other modern synthesis languages. SAOL has been designed to be integrated deeply with other MPEG-4 tools, and a discussion of this integration is presented. However,

it is also intended to be highly capable as a stand-alone music-synthesis language, and we provide some thoughts on the implementation of efficient stand-alone SAOL musical instruments. Strengths and weaknesses of the language in relation to other synthesis languages are also discussed. A discussion of the role of the MPEG-4 International Standard in the development of future computer music tools concludes the article.

SAOL: Structure and Capabilities

SAOL is a declarative unit-generator-based language. In this respect, it is more like Csound (Vercoe 1995; Boulanger forthcoming) than it is like SuperCollider (McCartney 1996a, b; Pope 1997) or Nyquist (Dannenberg 1997a); Nyquist employs a functional-programming model in its design, and SuperCollider employs an object-oriented model. SAOL extends the syntax of Csound to make it more understandable and concise, and adds a number of new features to the Music-N model that are discussed below.

It is not our contention that SAOL is a superior language to the others we cite and compare here. In fact, our belief is somewhat the opposite: the differences between general-purpose software-synthesis languages are generally cosmetic, and features of the languages' implementations are much more crucial to their utility for composers. For the MPEG-4 project, we developed SAOL anew because it has no history or intellectual-property encumbrances that could impede the acceptance of the standard. SAOL is not a research project that presents major advances in synthesis-language design; rather, it is an attempt to codify existing practice, as expressed in other current languages, to provide a fixed target for manufacturers and tools developers making use of software-synthesis technology.

There were several major design goals in the creation of SAOL. These were: to design a synthesis language that is highly *readable* (so that it is easy to understand and to modify instrument code), highly *modular* (so that general-purpose processing algorithms can be constructed and reused without modification in many orchestras), highly *expressive* (so that musicians can do complex things easily), and highly *functional* (so that anything that can be done with digital audio can be expressed in SAOL). Additionally, SAOL as a language should lend itself to efficient implementations in either hardware or software.

As well as the new features of SAOL that are described below, many well-established features of Music-N languages (Mathews 1969; Pope 1993) are retained. SAOL, like other Music-N languages, defines an *instrument* as a set of digital signal-processing algorithms that produces sound. A set of instruments is called an *orchestra*. Other retained features include: the *sample-rate/control-rate distinction*, which increases efficiency by reducing sample-by-sample calculation and allowing block-based processing; the *orchestra/score distinction*, in which the parametric signal-processing instructions in the orchestra are controlled externally by a separate event list called the *score* (one of Nyquist's innovations is the removal of this distinction); the use of *instrument variables* to encapsulate intermediate states within instruments and *global variables* to share values between instruments; and a heavy dependency on *stored-function tables* or *wavetables* to allow efficient processing of periodic signals, envelopes, and other functions. These historical aspects of SAOL will not be discussed further here, but excellent summaries on the evolution and syntactic construction of synthesis languages may be found in other references (Roads 1996; Dannenberg 1997a, b; and Boulanger forthcoming, among others).

Readability

Where Csound is "macro-assembly-like," Nyquist is "Lisp-like," and SuperCollider is "Smalltalk-like," SAOL is a "C-like" language. In terms of making the language broadly readable, this is a good step, because C is the most widely used of these languages. The syntactic framework of SAOL is familiar to anyone who programs in C, although the fundamental elements of the language are still signal variables, unit generators, instruments, and so forth, as in other synthesis lan-

Figure 1. A SAOL instrument that makes a short tone.

```
// This is a simple SAOL instrument that makes a short tone,
// using an oscillator over a stored function table.

instr beep(pitch,amp) {
  table wave(harm,2048,1);  // sinusoidal wave function
  asig sound;               // 'asig' denotes audio signal
  ksig env;                 // 'ksig' denotes control signal

  env = kline(0,0.1,1,dur-0.1,0); // make envelope
  sound = oscil(wave, pitch) * amp * env;
   // create sound by enveloping an oscillator
  output(sound);  // play that sound
}
```

guages. (The exact syntax of C is not used; there are several small differences that make the language easier to parse.) The program in Figure 1 shows a simple SAOL instrument that creates a simple beep by applying an envelope to the output of a single sinusoidal oscillator.

A number of features are immediately apparent in this instrument. The instrument name (`beep`), parameters (or "p-fields": `pitch` and `amp`), stored-function table (`wave`), and table generator (`harm`) all have names rather than numbers. All of the signal variables (`sound` and `env`) are explicitly declared with their rates (`asig` for audio rate and `ksig` for control rate), rather than being automatically assigned rates based on their names. There is a fully recursive expression grammar, so that unit generators like `kline` and `oscil` may be freely combined with arithmetic operators. The stored-function tables may be encapsulated in instruments or in the orchestra when this is desirable; they may also be provided in the score, in the manner of Music V (Csound also allows both options). The unit generators `kline` and `oscil` are built into the language; so is the wavetable generator `harm`.

The control signal `dur` is a standard name, which is a variable automatically declared in every instrument, with semantics given in the standard. There is a set of about 20 standard names defined in SAOL; `dur` always contains the duration of the note that invoked the instrument.

Modularity

There is a highly capable set of unit generators built into the SAOL specification (100 in all; see Appendix 1). This set is fixed in the standard, and all implementations of SAOL must implement them. However, SAOL may be dynamically extended with new unit generators within the language model. While other Music-N languages require rebuilding the language system itself to add new unit generators, this capability is a fundamental part of SAOL. An example orchestra using this capability is shown in Figure 2.

The `beep2` instrument makes use of a unit generator, `voscil`, which is not part of the standard set. The *user-defined opcode* below it implements this unit generator from a certain set of parameters. The *aopcode* tag indicates that the new unit generator produces an a-rate (audio-rate) signal. Each opcode parameter (`wave, cps, depth,` and `rate`) is similarly defined with a rate type (`table, ivar, ksig,` and `ksig`, respectively) that indicates the maximum rate of change of each parameter. Using the same core functionality as instruments, user-defined opcodes perform a certain calculation and then return their results. In this case, the `voscil` user-defined opcode makes use of the `koscil` and `oscil` core opcodes to calculate its result. Any orchestra containing this user-defined opcode may now make use of the `voscil` unit generator.

Figure 2. A SAOL orchestra that uses the user-defined opcode construction. The instrument `beep2` *makes use of the opcode* `voscil`*, which is not defined in the standard. The opcode declaration beneath it implements the opcode so that it is available for use in the orchestra. Every SAOL system is required to provide the capability to dynamically extend the language in this manner.*

```
// This is part of a SAOL orchestra showing the use of the
// user-defined opcode syntax.  The instrument 'beep2' makes use
// of the 'voscil' opcode, which is not part of the core SAOL
// syntax.  The opcode definition beneath it implements the
// opcode.

instr beep2(pitch,amp) {
  table wave(harm,2048,1);
  asig sound;
  ksig env;

  env = kline(0,0.1,1,dur-0.1,0);
  sound = voscil(wave,pitch,0.05,5) * env;
    // 'voscil' is not a built-in ugen...
  output(sound * amp);
}

aopcode voscil(table wave, ivar cps,
               ksig depth, ksig rate) {
  // ... so we declare it here.
  // It's an 'oscil' with vibrato:
  //   'wave' is the waveshape, 'cps' the carrier freq,
  //   'depth' the vibrato depth as a fraction,
  //   'rate' the vibrato rate
  ksig vib,newfreq;
  asig sound;
  table vibshape(harm,128,1); // waveshape for vibrato

  vib = koscil(vibshape,rate); // sinusoidal vibrato
  newfreq = cps * (1 - vib * depth); // FM banking
  sound = oscil(wave,newfreq); // new output
  return(sound);  // return 'sound' to caller
}
```

It is easy to imagine the construction of standard libraries of desirable opcodes for use in various synthesis applications; for example, mathematical functions such as Bessel functions and Chebyshev polynomials for FM synthesis, or special filters for physical-modeling synthesis. Since the unit-generator set is arbitrarily extensible within the language model, the problem of so-called opcode bloat that other synthesis languages have encountered may be avoided. User-defined opcodes may themselves depend on other user-defined opcodes, so a complete abstraction model is provided. The only limit on this abstraction is that recursive and mutually user-defined opcodes are prohibited; this can simplify the runtime language model, because it means that careful macro expansion can be used to implement user-defined opcodes in a SAOL compiler if desired. However, each user-defined opcode has its own name space, so that procedural abstraction is

Figure 3. A SAOL orchestra that defines a bus-routing scheme for effects processing. The beep instrument is routed to the bus `echo_bus`*; the output of* `beep` *is not turned into sound, but placed on the bus for further processing. The bus* `echo_bus` *is sent to the instrument* `echo`*, which implements a digital-delay echo.*

```
// This is a complete SAOL orchestra that demonstrates the
// use of buses and routing in order to do effects processing.
// The output of the 'beep' instrument is placed on the bus
// called 'echo_bus'; this bus is sent to the instrument called
// 'echo' for further processing.

global {
  srate 32000; krate 500;

  send(echo; 0.2; echo_bus};
    // use 'echo' to process the bus 'echo_bus'
  route(echo_bus, beep);
    // put the output of 'beep' on 'echo_bus'
 }

instr beep(pitch, amp) {
  // as above
}

instr echo(dtime) {
  // a simple digital-delay echo.  'dtime' is the
  // cycle time.
  asig x;

  x = delay(x/2 + input[0],dtime);
  output(x);
}
```

not affected by this restriction. All instruments and user-defined opcodes in the orchestra live within a single global name space; there is no mechanism for "packages" or similar concepts.

As with unit generators, extensibility is provided for wavetable (function) generators; about 20 built-in wavetable generators are provided (see Appendix 2), but composers may also write opcodes that act as generators for new functions.

Another aspect of modularity in SAOL involves its flow-of-control processing model. In Csound, the only way to allow instruments to post-process sound (for example, to add reverb to another instrument's output) is to shuttle signals between them with global variables. In SAOL, a metaphor of *bus routing* is employed that allows the concise description of complex networks. Its use is shown in Figure 3.

In this orchestra, a *global block* is used to describe global parameters and control. The `srate` and `krate` tags specify the sampling rate and control (LFO) rate of the orchestra. The send instruction creates a new bus called `echo_bus`, and specifies that this bus is sent to the effects-processing instrument called `echo`. The `route` instruction specifies that the samples produced by the instrument `beep` are not turned directly into sound output, but instead are "routed onto" the bus `echo_bus` for further processing.

The instrument `echo` implements a simple exponentially decaying digital-echo sound using the `delay` core opcode. The `dtime` p-field specifies the cycle time of the digital delay. Like `dur` in Figure 1, `input` is a standard name; `input` always contains the values of the input to the instrument, which in this case is the contents of the bus

Figure 4. A SAOL instrument that can be controlled continuously from a SASL score. The variables amp *and* off *are exposed to the score, allowing their values to be modified there.*

```
// This is a SAOL instrument that can be controlled with
// continuous controllers in the score.  The variables 'amp'
// and 'off' are exposed to the score.

instr beep3(pitch) {
  imports ksig amp, off; // controllers
  ksig vol;
  table wave(harm,2048,1);
  asig sound;

  if (!itime) { // first time we're called
    amp = 0.5;
  }

  if (off) { turnoff; } // we got the 'off' control
  vol = port(amp,0.2);  // make a smooth volume signal
  sound = oscil(wave,pitch);
  output(sound * vol);
}
```

echo_bus. Note that echo is not a user-defined opcode that implements a new unit generator, but an effects-processing instrument.

This bus-routing model is modular with regard to the instruments beep and echo. The beep sound-generation instrument does not "know" that its sound will be modified, and the instrument itself does not have to be modified to enable this. Similarly, the echo instrument does not "know" that its input is coming from the beep instrument; it is easy to add other sounds to this bus without modification to echo. The bus-routing mechanism in SAOL allows easy reusability of effects-processing algorithms. There are also facilities that allow instruments to manipulate busses directly, if such modularity is not desirable in a particular composition.

Expressivity and Control

SAOL instruments may be controlled through Musical Instrument Digital Interface (MIDI) files, real-time MIDI events, or a new score language called SASL (pronounced "sazzle," an acronym for Structured Audio Score Language). For the cases when MIDI is used, a set of standard names pertaining to MIDI allows access to the standard MIDI control, pitch-bend, and after-touch parameters; channel and preset mappings are also supported in SAOL. In SASL, more-advanced control is possible, as shown in Figures 4 and 5. The SAOL orchestra in Figure 4 can be controlled with the SASL score in Figure 5.

In the orchestra (see Figure 4), the control signals amp and off are declared with the tag imports, which indicates that they may be updated by the score. The amp signal allows continuous control of the amplitude of the instrument output, and the off signal allows the instrument to be instructed to turn itself off. Notice that the meanings of these control signals are not fixed in the standard (unlike MIDI); the composer is free to specify as many controllers as needed, with whatever meanings are musically useful. When the off control is received, the instrument uses the built-in turnoff command to turn itself off; the built-in port (portamento) unit generator is used to convert the discrete changes in the amp control signal into a continuous amplitude envelope.

Figure 5. A score that can be used to control the orchestra shown in Figure 4. The control lines set the values of variables in the various instances of instrument `beep3`*. The note labels (*`n1, n2, n3`*) control the mapping of control changes to note instances.*

```
// This is a score for controlling the orchestra shown
// above.

n1: 0.0 beep3 1 440
    0.5 beep3 1 480

n2: 1.0 beep3 -1 220
n2: 1.0 beep3 -1 440
n3: 1.0 beep3 -1 660

    2.0 control n2 amp
    2.5 control n2 amp 0.5
    3.0 control n3 amp 0.2
    3.0 control n2 amp 0.2
    4.0 control n2 off 1
    4.0 control n3 off 1
```

The `if (!itime)` clause is used to control the behavior of the first pass through each instance of the instrument. Like `dur` in Figure 1, `itime` is a standard name; `itime` always contains the amount of time the instrument instance has been executing. Thus, testing it for 0 allows initializations of "k-rate" (control-rate) variables to only be performed once. All variables in SAOL are like `static` variables in C; that is, they preserve their values between iterations. Thus, the assignment to `amp` is preserved in the next iteration.

In the SASL score (see Figure 5), `n1, n2,` and `n3` are *labels* that control the mapping of control information to note events. Two types of score lines are shown; each has an optional label and a time stamp that indicates the time at which the event is dispatched. The *instrument lines* specify the instrument that is to be used to create a note (`beep3` in each case), the duration of the note (–1 indicates that the duration is indefinite), and any other p-fields to be passed to the note, as defined in the orchestra. The *control lines* begin with a time stamp and the tag `control`, and then specify a label, a variable name, and a new value. The variable name given will be set to the new value in every note that was instantiated from an instrument line with the given label. In this way, score-based control is more general and flexible than in MIDI or Csound.

More advanced control mechanisms are also possible in SAOL and SASL. The built-in `instr` and `extend` commands allow instruments to spawn other instruments (for easy layering and synthetic-performance techniques) and dynamically change the durations of notes. A standard name (see the discussion of Figure 1), `cpuload`, allows dynamic voice-stealing algorithms to be included in an orchestra; `cpuload` always contains the current load of the processor on which an instrument is running, and is expressed as a percentage of capability.

SASL is a relatively simple format compared to other scoring languages; it provides only tempo and event-dispatch methods, not commands relating to sections, repeats, looping, expressions, stochastic performance, or many of the other features seen in advanced score formats. SASL's design assumes that in the long run most scores will actually be written by composing tools such as sequencers, which provide advanced capabilities to the musician; textual scores will not be written directly by the musician. Thus, the primary design goal for SASL was simplicity, so that it would be easy to construct such tools.

This is not to say that control is "easier" or "less essential" than synthesis. In fact, the MPEG-4 viewpoint is precisely the opposite. The time is right for standardization of the underlying signal-

processing structure of synthesis languages, because more than 30 years of research on this topic have greatly refined and proven the efficacy of the unit-generator paradigm. But sensitive, expressive control is still a matter of art, not of engineering, and so it is best not to try to standardize a control representation; rather, that is left up to future experiments in interface design and composition tools. The best that can be accomplished at this time is to create a standardized signal-processing substrate for such experiments. All that is needed to satisfy MPEG-4 requirements for efficient transmission is to provide a simple format that can flexibly control synthesis and play back a desired composition; this format need not be the one "in which the composer thinks."

Other recent work, especially at Berkeley's Center for New Music and Audio Technologies (CNMAT), has explored the creation of more powerful protocols for communication and control. ZIPI (McMillen, Wessel, and Wright 1994; Wright 1994) is a protocol for communication of musical parameters; it expands on MIDI by standardizing a large set of specific controls (pitch, volume, pan, spectral information, articulation, spatial position, etc.) and allowing more extensibility. ZIPI embeds a Music Parameter Description Language, and includes a physical-layer specification for transport over wires. Most of the functionality of ZIPI could be cross-coded into SASL without much difficulty, just as MIDI functionality can be embedded in MPEG-4 Structured Audio (see below). One notable aspect of ZIPI that SASL cannot emulate is the ability to query the capabilities of a synthesizer and make control decisions based on the results.

OpenSound Control (Wright and Freed 1997), or OSC, is another new format that moves entirely away from the "channel" and "note" fixation of most other control formats. It uses an open-ended symbolic addressing scheme, including a powerful wild-card syntax, to enable hierarchical control over real-time synthesis parameters. Researchers at CNMAT have been developing powerful tools to enable the integration of OSC into other systems (Wright 1998). Perhaps in the future it will be possible to converge the flexible synthesis capabilities of SAOL with the advanced control scheme of OSC. Note that there is no prohibition in the MPEG-4 standard from making a SAOL implementation respond to controls expressed in OSC; there is simply no requirement to do so.

The SAOL language is powerful enough for algorithmic composition and score-generation facilities to be written directly in the orchestra code. Using this method, synthetic performers may be included in the orchestra to mediate performance between musical intent (scoring) and sound creation (instrument dispatch). For example, the bass line of a jazz composition could be created by delivering the algorithms to dynamically improvise note choices and line shapes, and then controlling this virtual bassist with high-level commands about chords, note density, tempo, and playing style. Such dynamic, stochastic compositions can be written directly in SAOL and do not need an external-language interface or powerful score language to generate note sequences.

Advanced Functionality

There are many advanced features of SAOL that were added during the evaluation stage to enable easier construction of complex instrument models. For example, arrays of signals and unit generators may be created and easily manipulated, as shown in Figure 6.

This instrument creates Shepard tones, octave-complex tones that have only relative, not absolute, pitch height (Shepard 1964). The `freq, amp,` and `part` variables are defined as *signal arrays*; each holds eight signals rather than one. Depending on the application, such signal arrays may represent multichannel sounds or, as in this case, multiple components of a single sound. An *opcode array* is used to declare a bank of eight oscillators, using the `oscil` unit generator. Any built-in or user-defined opcode may be used in this construction.

The opcode array is used in the sixth line from the bottom of the program: `oscil[j](...)`. It allows for the concise description of a bank of parallel oscillators with independent state and parameters. In this case, all of them use the same fundamental waveshape, but this could also be

Figure 6. A SAOL instrument that produces Shepard tones. The oparray construction `oscil[j]` *allows multiple instances of the* `oscil` *opcode to be easily addressed in a loop. Any built-in or user-defined opcode can be used in this construction.*

```
// This is a SAOL instrument that uses 'opcode arrays' to
// implement Shepard-tone sounds.

instr shepard(pc) {
  table env(window,120,4);  // a Gaussian envelope
  table wave(harm,2048,1);
  ivar i, freq[8], amp[8]; // initialization variables
   // vectors to hold component freqs and amps
  asig s, j, part[8];
   // an eight-channel sound
  oparray oscil[8];
   // an 8-channel bank of oscillators

  // i-time (done at instrument startup)
  freq[0] = 60 * pow(2,pc/12);  // calculate base freq
  amp[0] = tableread(env,10*log(freq[0]));
          // look up amplitude in table.
  i = 1; while (i < 8) { // for each component...
    freq[i] = freq[i-1]*2;
          // it's an octave up from the last one
    amp[i] = tableread(env,10*log(freq[i]));
          // look up amplitude in table
    i = i + 1;
  }

  // a-time (each sample of synthesis)
  s = 0; j = 0;
  while (j < 8) { // for each channel...
    part[j] = oscil[j](wave,freq[j]);
          // run the j'th oscillator
    s = s + part[j] * amp[j];
          // scale it by its amplitude and add it to
         // the total
    j = j + 1;
  }

  output(s);  // output the total
}
```

controlled with an array index if desired. Arithmetic operations on arrays automatically account for the number of channels involved; pointwise vector operations and operations involving both vectors and scalars are built into the language. Vector operations such as `sum`, `mean`, etc. are not built in as core unit generators, but may easily be added if they are useful in a certain composition.

Other advanced features of SAOL include built-in spectral manipulation unit generators (`fft`, for the fast Fourier transform, and `ifft`, for its inverse), which allow spectral-domain techniques

based on overlap-add windowing; a sophisticated fractional and multitap delay-line unit generator (`fracdelay`); and built-in support for granular synthesis (`grain`), Karplus-Strong plucked-string synthesis (`pluck`), waveshaping synthesis, high-quality parametric compression (`compressor`), and others. A template mechanism is provided for the concise description of multiple instruments that differ only in a simple way. For details on these and other features, readers are referred to the final draft of the standard (International Standardization Organization 1999), which contains the definition of SAOL.

Efficiency in SAOL

The use of control signals and block-based processing has been an area of some recent debate in the literature on music-synthesis languages. Dannenberg (1997b) provides an excellent, thoughtful summary of the relevant issues. In the earliest synthesis languages, the semantics of processing were sample-by-sample; Barry Vercoe's languages Music-11 and Csound innovated the use of the control rate and block-based processing to add efficiency. In block-based processing, the semantics of each line are evaluated for an entire block of samples before the next line is evaluated. Other modern languages, such as Nyquist and Supercollider, have adopted this mechanism as well. SAOL returns to the earlier model and uses sample-by-sample semantics for processing, although it preserves the audio-rate/control-rate distinction to allow dual-rate operation (that is, to allow some signals to be updated more slowly than others) and to specify the semantics of controllers and global variables (for example, only control-rate variables may be shared between instruments).

The change back to sample-by-sample semantics has two primary implications. First, instruments that make use of single-delay feedback in their synthesis algorithms can be constructed in the "obvious" manner, without having to artificially set the sampling rate and control rate to be equal. Second, not every algorithm may be implemented in a strictly block-processing manner by the SAOL interpreter or compiler. A SAOL implementation that wishes to make use of block processing must statically detect, through optimization methods, when it is and is not possible in a particular instrument. An informative note in the MPEG-4 standard discusses this point.

In practice, this is not a barrier to efficient orchestras once these sorts of sophisticated implementations become available. Any algorithms that cannot be optimized into block-processing modes are simply algorithms that are impossible to construct in a strictly block-processing language (except with `srate = krate`). A composer who wishes to have the maximum efficiency in operation for a real-time composition simply refrains from making use of these algorithms.

It is reasonable to believe that such optimizing implementations are not too difficult to develop; the technology required is very similar to that in parallelizing compilers for vector supercomputers. There is a large literature in the compiler-design literature on this topic (Allan, Jones, Lee, and Allan 1995) that may be readily applied to the design of SAOL compilers.

The specification of SAOL in the MPEG-4 standard does not restrict the form of an implementation. The standard specifies only what an implementation must do—that is, turn SAOL code into sound in a particular manner—not how it must operate. SAOL systems may be implemented in hardware or software, using DSP or native processors, using block-based or single-sample processing, embedded in larger systems or as stand-alone instruments, depending on the needs of the particular music application.

SAOL in the MPEG-4 Standard

Although, as demonstrated above, SAOL is a powerful and flexible music-synthesis language on its own, it has additional capabilities stemming from its design as part of the MPEG-4 International Standard. Space does not permit a full description of these capabilities here; a summary is provided below, and interested readers are referred to other articles for more details.

MPEG-4 Structured Audio

The MPEG-4 Structured Audio standard (Scheirer 1998; Scheirer, Lee, and Yang forthcoming) contains five main sections, two of which (SAOL and SASL) have been described above. The other three pieces are as follows.

A *rich wavetable* format called SASBF (pronounced "sazz-biff"), for Structured Audio Sample Bank Format, allows banks of samples for use in wavetable synthesis to be efficiently transmitted. The format of SASBF derives from the MIDI Downloadable Sound (DLS) (MIDI Manufacturers Association 1996) and E-Mu SoundFonts formats, and combines the best features of both; it has been designed in collaboration with the MIDI Manufacturers Association (MMA) and various companies interested in this technology. The MMA is independently standardizing the same format as DLS Level 2.

A restricted *profile* of MPEG-4 Structured Audio allows the use of only wavetable synthesis in low-cost applications where sophisticated sound control is not needed. In the full profile of Structured Audio, wavetable synthesis (based on SASBF) and general-purpose software synthesis (based on SAOL) may be mixed and combined as needed (Scheirer and Ray 1998).

A set of *MIDI semantics* allows the use of MIDI data to control synthesis in SAOL. These instructions in the standard specify how to generate sound in response to MIDI note events and controllers. Through these semantics, the wealth of existing content and composition tools (sequencers) may be used to create MPEG-4–compatible soundtracks until MPEG-4–native tools are available.

Real-time data in the MIDI protocol (or standard MIDI files) may be used to control synthesis in conjunction with, or instead of, SASL control files. As MIDI events are received in the MPEG-4 terminal, they are translated into SAOL events according to the MIDI semantics specified in the standard. Most of the MIDI semantics provide the "obvious" behavior for MIDI events in MPEG-4 when possible.

Finally, a *scheduler semantics* describes exactly how control events, SASL scores, MIDI data, and the SAOL orchestra interact to produce sound. These instructions are *normative* in the MPEG-4 standard, which means that every different MPEG-4 implementation must follow these rules to ensure that musical content sounds exactly the same on every synthesis device. The scheduler is specified in terms of a set of careful rules given on the order of execution of instrument instances, order of dispatch of events, and order of processing sound with buses.

Other MPEG-4 Audio Coders

Structured Audio tools are not the only audio tools in MPEG-4. There is also a set of highly functional natural-audio tools that allow traditional compression of streaming audio (Quackenbush 1998). The MPEG-4 General Audio (GA) coder is based heavily on the powerful MPEG-2 Advanced Audio Coding (AAC) method, with extensions that allow for more scalability and better performance at low bit rates. MPEG's psychoacoustic tests (Meares, Watanabe, and Scheirer 1998) have demonstrated that AAC can provide quality nearly indistinguishable from uncompressed digital audio at 64 kb/sec (kbps) per channel; AAC showed significantly higher quality than the MPEG-1 Layer 3, Dolby AC-3, or Lucent PAC methods in tests at an independent laboratory (Soulodre et al. 1998). Using the MPEG-4 extensions, GA coding provides excellent audio quality at rates as low as 16 kbps per channel.

The MPEG-4 Codebook Excited Linear Prediction (CELP) coder allows the coding of wide-band or narrow-band speech signals, with high quality at 24 kbps and excellent compression down to 12 kbps. The MPEG-4 Parametric Speech coder allows ultra-low-bit-rate compression of speech and simple music signals down to 4 kbps per channel.

AudioBIFS

The tools described above, both synthetic and natural, represent the state of the art in sound synthesis and sound compression. However, another level of power is possible in MPEG-4 with the

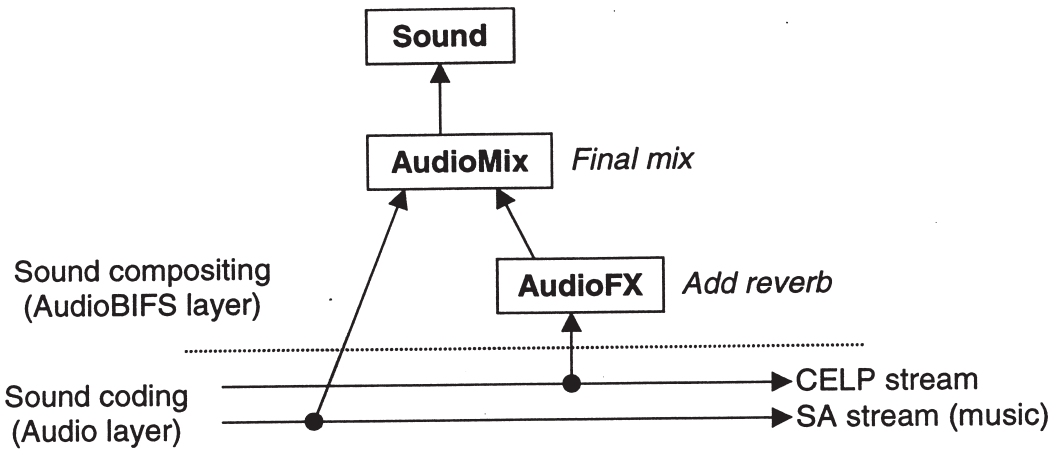

Figure 7. Two sound streams are processed and mixed using the AudioBIFS scene graph. A speech sound is transmitted with MPEG-4 CELP, and reverb is added with an AudioFX node, then the result is mixed with a musical background transmitted with the MPEG-4 Structured Audio system.

AudioBIFS system (Scheirer, Väänänen, and Huopaniemi forthcoming), part of the MPEG-4 Binary Format for Scene Description (BIFS). AudioBIFS allows multiple sounds to be transmitted using different coders and then mixed, equalized, and post-processed once they are decoded. This format is structurally based on the Virtual Reality Modeling Language (VRML) 2.0 syntax for scene description (International Standardization Organisation 1997), but contains more powerful sound-description features.

Using MPEG-4 AudioBIFS, each part of a soundtrack may be coded in the format that best suits it. For example, suppose that the transmission of voiceover with background music is desired in the style of a radio advertisement. It is difficult to describe high-quality spoken voice with sound-synthesis techniques, and so Structured Audio alone cannot be used; but speech coders are not adequate to code speech with background music, and so MPEG-4 CELP alone cannot be used. In MPEG-4 with AudioBIFS, the speech is coded using the CELP coder, and the background music is coded in Structured Audio. Then, at the decoding terminal, the two "streams" of sound are decoded individually and mixed together. The AudioBIFS part of the MPEG-4 standard describes the synchronization provided by this mixing process.

AudioBIFS is built from a set of *nodes* that link together into a tree structure, or *scene graph*. Each of these nodes represents a signal-processing manipulation on one or more audio streams. In this, AudioBIFS is somewhat itself like a sound-processing language, but it is much simpler (there are only seven types of nodes, and only "filters," no "generators"). The scene-graph structure is used because it is a familiar and tested mechanism for computer-graphics description, and AudioBIFS is only a small part of the overall BIFS framework. The functions performed by AudioBIFS nodes allow sounds to be switched (as in a multiple-language soundtrack), mixed, delayed, "clipped" for interactive presentation, and gain-controlled.

More-advanced effects are possible by embedding SAOL code into the AudioBIFS scene graph with the *AudioFX* node. Using AudioFX, any audio effect may be described in SAOL and applied to the output of a natural or synthetic audio decoding process (see Figure 7).

For example, if we want to transmit reverberated speech with background music, we code the speech with MPEG-4 CELP, and provide an AudioFX node containing SAOL code that implements the desired reverberator. As above, we code the background music using MPEG-4 Structured Audio. When the transmission is received, the speech stream is decoded and the reverberation processing is performed; the result is added to the background music, which might or might not also be reverberated. Only the resulting sound is played back to the listener. Just as MPEG-4 Structured

Audio allows exact, terminal-independent control of sound synthesis for the composer, MPEG-4 AudioBIFS allows exact, terminal-independent control of audio-effects processing for the sound designer and producer.

The combination of synthetic and natural sound in the same sound scene with downloaded mixing and effects processing is termed *synthetic/natural hybrid coding*, or SNHC audio coding, in MPEG-4. SNHC is a powerful coding concept that MPEG-4 is the first standard to use (Scheirer 1999; Scheirer, Lee, and Yang forthcoming).

Other features of AudioBIFS include 3-D audio spatialization for sound presentation in virtual-reality applications, and the creation of sound scenes that render differently on different terminals, depending (for example) on sampling rate, speaker configuration, or listening conditions.

Finally, the AudioBIFS component of MPEG-4 is part of the overall BIFS system, which provides sophisticated functionality for visual presentation of streaming video, 3-D graphics, virtual-reality scenes, and the handling of interactive events. The audio objects described with AudioBIFS and the various audio decoders may be synchronized with video or computer-graphics objects, and altered in response to user interaction.

Streaming Media versus Fixed Media

Like previous MPEG standards, MPEG-4 has been designed with streaming media in mind. There is an extensive set of tools in the MPEG-4 systems component for efficiently handling the transport and delivery of streaming data, including methods for multiplexing, synchronization, and back-channel communications. MPEG-4 Structured Audio was designed in this framework, and so the sound-synthesis capabilities described in this article may all be transmitted as streaming data. In this model, the *bitstream header* contains the orchestra file and any necessary preparatory data; at the beginning of the session, the client terminal processes this header and prepares the orchestra for synthesis. Then, the streaming data contains score and control information that drives the synthesis.

High-quality synthesis may be performed at very low bit rates, down to less than 1 kbps, and the synthetic content may be synchronized with natural sound content and multiplexed into a single program bit stream. The streaming data uses a tokenized, compressed binary format, instead of the human-readable "textual SAOL format" that is used in this article's SAOL code examples.

The Structured Audio components have also been designed to be efficient and useful in fixed-media applications, such as a studio environment or composition workstation. The standard includes the textual SAOL format for this purpose. Additionally, the standard contains suggestions for stand-alone implementation authors, regarding good ways to allow access to sound samples and user data, and ways to provide debugging information. The *saolc* implementation, which is the MPEG-4 Structured Audio "reference software" (the official implementation of the standard), contains a stand-alone mode in which SAOL is used from a command line, very much like Csound. Suggestions for implementing real-time local MIDI control by attaching a MIDI keyboard are also provided in the standard.

Standardization in Computer Music

This section discusses the role that the international standardization process has played in the development of SAOL, and the advantage this process can serve for the computer music community. SAOL did not exist as a language prior to the MPEG-4 standard; the standards work was not undertaken in order to place an ISO "seal of approval" on previously existing technology. Rather, a concerted effort was made to design the best synthesis language possible, with the broadest application, so that MPEG-4 Structured Audio could serve the best purpose of standards: to unify a sometimes fragmented field and marketplace and prevent duplication of effort. If the MPEG-4 Structured Audio standard becomes broadly accepted, all computer musicians will benefit from the resulting explosion of hardware accelerators and composition tools around it.

Languages and Implementations

When evaluating synthesis languages as languages, it is important to keep in mind the distinction between language and implementation. For a particular music system, the language is the set of syntactic rules that describe what sequences of characters make up legal programs, and the set of semantic rules that describe how to make sound from a program. An implementation, by contrast, is a particular computer program that can perform the required mapping from a sequence of characters to sound. Depending on the music system, other information such as scores, MIDI files, control sequences, or real-time interaction may affect this mapping process.

It has traditionally been the case in the computer music field that languages have been defined in regard to implementations. That is, the Csound language is not defined anywhere except as a computer program that accepts or rejects particular sequences of characters. As this implementation evolves and changes over time, the set of "legal" Csound orchestras changes as well (generally by expansion, in a backward-compatible manner). This is not the case in the computer-language world at large; languages such as C, Fortran, Lisp, Ada, C++, and Java have clearly written specifications that describe exactly what a legal implementation must and must not do.

The primary advantage of the specified-language model is that it promotes compatibility between implementations. For a large set of nonpathological programs, a programmer is guaranteed that a C program will execute the same way under any legal C compiler. The ANSI C standard tells the developer of a new C compiler exactly what the compiler must do in response to a particular program. This interoperability promotes the development of an efficient marketplace for compilers, debuggers, editors, and other programming tools. If a company develops a new compiler that is much faster than others available, the marketplace can shift quickly because existing code is still useful, and so there is a competitive advantage to a company to develop such powerful compilers. If languages are not compatible between implementations, programmers (musicians) become "locked into" a certain implementation (because it is the only one that will run their instruments), and there is not a competitive reason to develop a better one.

The disadvantages of specified languages are that standards have a certain stifling effect on innovation (C and C++ have had a stronghold on the development of production code for many years), and they might not utilize the resources of a particular computing environment with maximal efficiency. Even after years of compiler development, hand-coded assembly, especially for high-performance architectures such as DSPs or vector processors, still may have performance advantages over code compiled from high-level languages. But since assemblers are generally incompatible from architecture to architecture, the development tools for assembly language coding are not as advanced as those for high-level languages. With sufficient effort and programming skill, advanced development environments and high-performance signal-processing capabilities can be merged in a single tool.

James McCartney's language and software system, SuperCollider (McCartney 1996a, b), is an excellent example of this: it provides highly efficient execution, a sophisticated front end, and a synthesis language with more-advanced capabilities than SAOL, but only in a proprietary framework on a single platform. Note, though, that the quality of SuperCollider's front end and the efficiency of its execution are not properties of SuperCollider as a synthesis language, but rather a testament to Mr. McCartney's skill in creating an implementation of the language. Similarly, the restriction to a single platform is not intrinsic to the language; the language could, once clearly specified, be implemented on other platforms with other front ends.

As computers and signal-processing units get ever faster, eventually the need for high-quality and compatible development tools for synthesis languages will be more pressing than the need to squeeze every cycle out of existing processors. This is the world that the design of SAOL targets: one in which musicians are willing to accept 75-percent performance on state-of-the-art hardware in exchange for inexpensive yet powerful imple-

mentations, sophisticated interfaces, cross-platform compatibility, and a broad and competitive marketplace for tools, music, and hardware.

Although, as of this writing, there are not yet implementations of SAOL whose speed is competitive with tools that have more development time behind them, this is not a property of the language. The SAOL standard requires real-time implementation for compliance to the standard, and so real-time SAOL processors will rapidly emerge from companies who wish their tools to be MPEG-4 compliant. The technology required to implement such tools efficiently is fairly well understood today. Thanks to the advances of innovators such as James McCartney and Roger Dannenberg, creating implementations from a well-specified language design is (mostly) a matter of engineering. This is especially true when one considers the amount of development resources that will be directed toward SAOL implementation as a result of its inclusion in MPEG-4.

The MIDI Standard

A widely used existing standard for the representation of sound-control parameters is the MIDI protocol (MIDI Manufacturers Association 1996; Loy 1985). In some ways, representation of music as a set of MIDI bytes is similar to the Structured Audio concept, as compression is achieved in similar ways. However, MPEG-4 Structured Audio is a standard for the representation of sound, not just musical control. In MIDI-based music exchange, musicians have little control over the exact sounds produced (since this is left to the particular hardware/software system that implements the synthesis), and as a result, MIDI is not generally an appropriate exchange format for serious musical content (Moore 1991).

In contrast, sound transmission in SAOL is much more exact; if composers desire the exact recreation of certain sounds, this is always possible in MPEG-4. The MPEG-4 standard places tight control—in many places, sample-by-sample specification—on the output of the synthesis in response to a particular SAOL program.

The MIDI standard was created to enable data exchange between fixed hardware synthesizers. In contrast, the MPEG-4 Structured Audio standard was created to enable data exchange between flexible software synthesizers. The MIDI standard, although perhaps not as capable as many musicians would like, provides a valuable demonstration that manufacturers and tool developers will indeed rally around a public standard for the exchange of music data.

Compatibility

The concrete specification of SAOL as part of the MPEG-4 standard means that any SAOL-compliant tool will be compatible at the SAOL level with any other such tool. Such compatibility has many implications. It makes it much easier to rapidly develop new software-synthesis tools such as graphical user interfaces (GUIs) for instrument construction. Such a tool may be built to interact with a SAOL-compliant hardware synthesizer, and so the GUI author does not have to construct the real-time synthesis capabilities as part of the tool. Similarly, a production musician who wishes to use new sounds does not have to develop them on his or her own. Because of the modular nature of SAOL, instruments, effects algorithms, and unit generators may be easily traded (or bought and sold) between developers, and then rapidly used as part of a new orchestra. Finally, since the MPEG-4 standard will be widely used in low-bit-rate communication applications such as digital broadcasting and satellite-to-mobile-platform transmission, there will be a great need for computer musicians to build the synthesis hardware and composition tools, and to compose the music that will be used in such everyday applications.

Open Standards

The ISO in general, and the MPEG working group in particular, are emblematic of the open-standards process in the development of new technology. Any researcher or company may make

suggestions regarding the future development of MPEG-4 and its Structured Audio tools (such as proposals for corrigenda, if parts of the standard are found to be "broken" or incomplete). According to MPEG's rules of procedure, these contributions must be formally evaluated, and if they are judged to be important and to move in a positive direction, they will be incorporated into the standard. Anyone can read and evaluate the capabilities of the standard for themselves. In addition, MPEG maintains a software implementation of the entire MPEG-4 standard, which is available to anyone interested in developing his or her own tools compliant with MPEG-4. The MIT Media Lab wrote and maintains the reference source code for the SAOL tools, and has released this source code into the public domain for free use by the community. The Media Lab maintains no intellectual-property rights or proprietary control over the direction of the standard, and will not gain materially from its acceptance in any way.

The reference implementation of SAOL, called saolc, is not intended to be a tool that is usable by musicians or competitive with modern synthesis-programming environments. It is a simple, text-based implementation of SAOL, SASL, SASBF, the MIDI rules, and the scheduler semantics in MPEG-4 Structured Audio that is supposed to be exactly conformant to the standard and relatively easy to read and understand. MPEG provides reference software for MPEG-4 as a secondary reference for normative behavior of the standard. Organizations interested in developing more practical tools can use saolc as a guide.

The source base for saolc is about 32,000 lines of C and C++ code totaling about 1 MB of data. On the Silicon Graphics, Inc. (SGI) platform, this compiles into an executable with a 900-KB footprint; on Windows 95 under Visual C++, the executable footprint is about 380 KB. The code base is intended to be widely portable and compiles as-is on (at least) SGI, Alpha, Linux, Sun, and Win32 platforms. The performance of saolc is extremely poor compared to modern real-time software synthesizers; even on a high-performance machine such as an SGI Octane, only one or two voices of very simple instruments will run in real time. There has been essentially no attempt to optimize saolc for speed, so the performance of saolc should not be taken as indicative of the performance of an optimized SAOL implementation. Moreover, saolc provides no user interface except command-line and bit-stream processing capabilities.

The Big Picture

Julius Smith (1991) wrote an incisive critique of the direction of progress in the digital musical instrument industry. Unfortunately, many of his observations still hold true today: "The ease of using MIDI synthesizers has sapped momentum from synthesis algorithm research by composers… MIDI synthesizers offer only a tiny subset of the synthesis techniques possible in software…The disadvantage [of MIDI] is greatly reduced generality of control, and greatly limited synthesis specification" (Smith 1991). Even with the development of the MIDI-DLS specification, the limitations of MIDI (and now DLS) continue to be a powerful force shaping the capabilities of computer sound processing.

This is a particularly disturbing state of affairs, given the vast developments in graphics processing, multimedia systems, and programming interfaces that have taken place since Julius Smith's article was written. The world of graphics on personal computers has evolved through several generations and a great deal of technological sophistication since 1991, but the fundamental sound architecture of the average PC has taken only a single step: instead of FM synthesis controllable with "note-on, note-off" instructions, now we have wavetable synthesis controllable with note-on, note-off instructions. Soon, new developments will allow composers to specify wavetables of their own, rather than having them provided by the sound-card manufacturers, and control them with note-on, note-off instructions.

The MPEG-4 Structured Audio tools have the potential to change this situation. The structure and format will be taken seriously, for MPEG standards are an important touchstone in the computer industry at large. Through the inclusion of

general-purpose software-synthesis tools in the MPEG-4 standard, the needs and inventions of the computer music community have been exposed to the computer industry for a very broad hearing. We hope that other computer musicians will join us in supporting the new standard, developing conformant implementations, and creating music and tools that can help it to prosper.

Conclusion

SAOL, the MPEG-4 Structured Audio Orchestra Language, is a powerful and flexible new synthesis language. Further, as part of the MPEG-4 International Standard, there will soon be many real-time implementations, composer's desktops, and other tools available for using it. The move to acceptance of powerful open standards in the computer music field will create an explosion of opportunity for musicians and technologists with the creativity and skill these tools demand, and will create a "rising tide" supporting other real-time software-synthesis tools.

For readers interested in using or developing tools based on SAOL, the SAOL home page on the World Wide Web may be found at http://sound.media.mit.edu/mpeg4. This site contains current information on the progress of the standard, up-to-date software implementations, example compositions, a library of user-defined unit generators, complete documentation on the SAOL language, and mailing lists that support the SAOL community.

Acknowledgments

The first author is grateful to the Interval Research Corporation (Palo Alto, California) for its fellowship support over the first year of this work, and to the Digital Life consortium of the MIT Media Laboratory for ongoing research funding. As always, the Machine Listening Group of the Media Lab, especially Keith Martin and Youngmoo Kim, have been essential through their comments and critiques on this article. Thanks also to two anonymous referees whose critiques greatly improved the article.

The work described herein has had many contributors, including, but not limited to: Paris Smaragdis, Bill Gardner, Michael Casey, Adam Lindsay, Giorgio Zoia, Jyri Huopaniemi, Riitta Väänänen, Itaru Kaneko, Shigeki Fujii, Lee Ray, Brian Link, Luke Dahl, Dave Sparks, Billy Brackenridge, and Tom White. Thanks to Pete Schreiner, Pete Doenges, and Leonardo Chiariglione for overseeing the MPEG-4 project and the Structured Audio components, and to Ron Burns and Don Mead from Hughes Aircraft Company for energizing the Media Lab's contribution to MPEG.

References

Allan, V. H., R. B. Jones, R. M. Lee, and S. J. Allan. 1995. "Software Pipelining." *ACM Computing Surveys* 27(3):367–432.

Boulanger, R., ed. Forthcoming. *The Csound Book*. Cambridge, Massachusetts: MIT Press.

Casey, M. A., and P. Smaragdis. 1996. "Netsound: Real-Time Audio from Semantic Descriptions." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, p. 143.

Dannenberg, R. B. 1997a. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21(3):50–60.

Dannenberg, R. B. 1997b. "The Implementation of Nyquist, a Sound-Synthesis Language." *Computer Music Journal* 21(3):71–82.

International Standardization Organisation (ISO). 1997. *ISO/IEC 14472–1 International Standard: Virtual Reality Modeling Language (VRML).* Available at http://www.vrml.org.

International Standardization Organisation (ISO). 1999. *ISO 14496-3:1999 (MPEG-4 Audio)*. Geneva: International Standardization Organization.

Loy, D. G. 1985. "Musicians Make a Standard: The MIDI Phenomenon." *Computer Music Journal* 9(4):8–25.

Mathews, M. V. 1969. *The Technology of Computer Music.* Cambridge, Massachusetts: MIT Press.

McCartney, J. 1996a. *SuperCollider: A Real-Time Sound Synthesis Programming Language* (program reference manual). Austin, Texas. Available at http://www.audiosynth.com.

McCartney, J. 1996b. "SuperCollider: A New Real-Time

Sound Synthesis Language." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 257–258.
McMillen, K., D. L. Wessel, and M. Wright. 1994. "The ZIPI Music Parameter Description Language." *Computer Music Journal* 18(4):52–73.
Meares, D., K. Watanabe, and E. D. Scheirer. 1998. "Results of the MPEG-2 AAC Stereo Verification Tests." ISO/IEC JTC1/SC29/WG11 (MPEG) document N2006. San Jose, California: International Standardization Organisation. Available at http://www.cselt.it/mpeg.
MIDI Manufacturers Association (MMA). 1996. "The Complete MIDI 1.0 Detailed Specification v. 96.2." Ordering information available at http://www.midi.org.
Moore, F. R. 1991. "The Dysfunctions of MIDI." *Computer Music Journal* 12(1):19–28.
Pope, S. T. 1993. "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal* 17(2):23–54.
Pope, S. T. 1997. *Sound and Music Processing in SuperCollider.* Available at http://www.create.ucsb.edu/htmls/sc.book.html.
Quackenbush, S. 1998. "Natural Audio Coding in MPEG-4." *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. Washington, DC: Institute for Electrical and Electronics Engineers, pp. 3797–3800.
Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, Massachusetts: MIT Press.
Scheirer, E. D. 1998. "The MPEG-4 Structured Audio Standard." *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. Seattle: Institute for Electrical and Electronics Engineers, pp. 3801–3804.
Scheirer, E. D. 1999. "Structured Audio and Effects Processing in the MPEG-4 Multimedia Standard." *Multimedia Systems* 7(1):11–22.
Scheirer, E. D, Y. Lee, and J.-W. Yang. Forthcoming. "Synthetic Audio and SNHC Audio in MPEG-4." In A. Puri and T. Chen, eds. *Advances in Multimedia: Signals, Standards, and Networks.* New York: Marcel Dekker.
Scheirer, E. D., and L. Ray. 1998. "Algorithmic and Wavetable Synthesis in the MPEG-4 Multimedia Standard." *Proceedings of the 105th AES Convention.* San Francisco: Audio Engineering Society. (Available as reprint #4811.)
Scheirer, E. D., R. Väänänen, and J. Huopaniemi. Forthcoming. "AudioBIFS: Describing Audio Scenes with the MPEG-4 Multimedia Standard." To appear in IEEE Transactions on Multimedia.
Schottstaedt, W. 1994. "Machine Tongues XVII: CLM: Music V Meets Common Lisp." *Computer Music Journal* 18(2):30–37.
Shepard, R. N. 1964. "Circularity in Judgments of Relative Pitch." *Journal of the Acoustical Society of America* 36(12):2346–2353.
Smith, J. O. 1991. "Viewpoints on the History of Digital Synthesis." *Proceedings of the 1991 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 1–10.
Soulodre, G. A, T. Grusec, M. Lavoie, and L. Thibault. 1998. "Subjective Evaluation of State-of-the-Art Two-Channel Audio Codecs." *Journal of the Audio Engineering Society* 46(3):164-177.
Vercoe, B. L. 1995. *Csound: A Manual for the Audio Processing System* (program reference manual). Cambridge, Massachusetts: MIT Media Laboratory.
Vercoe, B. L., W. G. Gardner, and E. D. Scheirer. 1998. "Structured Audio: The Creation, Transmission, and Rendering of Parametric Sound Descriptions." *Proceedings of the IEEE* 86(5):922–940.
Wright, M. 1994. "A Comparison of MIDI and ZIPI." *Computer Music Journal* 18(4):86–91.
Wright, M. 1998. "Implementation and Performance Issues with OpenSound Control." *Proceedings of the 1998 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 224–227.
Wright, M., and A. Freed. 1997. "OpenSound Control: A New Protocol for Communicating with Sound Synthesizers." *Proceedings of the 1997 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 101–104.

Appendix 1: Core Opcodes in SAOL

This appendix contains a complete list of the built-in "opcodes," or unit generators, in SAOL. Space does not permit a full description of the parameters, syntax, and semantics of each; interested readers are referred to the standard or to our online documentation. In the standard, the operation of each unit generator is defined at the sample-by-sample level. All of these unit generators must be present in any compliant SAOL implementation.

Math Functions

Each of these implements the corresponding mathematical function:

```
int, frac, abs, sgn, exp, log, sqrt,
sin, cos, tan, asin, acos, atan, pow,
log10, floor, ceil, min, max.
```

These allow conversion from amplitude multipliers to decibels and back:

```
dbamp, ampdb.
```

Pitch Converters

These convert from one of SAOL's pitch formats to another. The pitch formats are: octave-fraction (oct), pitch class (pch), frequency in Hz (cps), and MIDI note number:

```
octpch, pchoct, cpspch, pchcps, octcps,
cpsoct, midipch, pchmidi, midioct,
octmidi, midicps, cpsmidi.
```

These allow the "global tuning" of the orchestra to be inspected and changed:

```
gettune, settune.
```

Stored-Function-Table Operations

These allow access to the length, loop point, loop end point, sampling rate, and base frequency (if any) of a stored function table:

```
ftlen, ftloop, ftloopend, ftsr,
ftbasecps.
```

These allow the loop point, loop end point, base frequency, and sampling rate of a stored function table to be modified:

```
ftsetloop, ftsetend, ftsetbase,
ftsetsr.
```

These provide direct read/write access to the sample data in a stored-function table:

```
tableread, tablewrite.
```

These generate a periodic audio signal from a wavetable, using simple loops, frequency matching, or sample-rate matching, respectively:

```
oscil, loscil, doscil.
```

This generates a periodic control signal from a wavetable:

```
koscil.
```

Signal Generators

These generate a line-segment signal at the control rate or audio rate, respectively:

```
kline, aline.
```

These generate an exponentially curved signal at the control rate or audio rate, respectively:

```
kexpon, aexpon.
```

These generate a control-rate or audio-rate continuous-phase signal, respectively:

```
kphasor, aphasor.
```

These perform Karplus-Strong or parametric granular synthesis:

```
pluck, grain.
```

This generates band-limited pulse-train signals:

```
buzz.
```

Noise Generators

These generate "white" random numbers or noise:

```
irand, krand, arand.
```

These generate random numbers or noise from a triangular distribution:

```
ilinrand, klinrand, alinrand.
```

These generate random numbers or noise from an exponential distribution:

```
iexprand, kexprand, aexprand.
```

These generate a Poisson-distributed random-impulse sequence or signal:

`kpoissonrand, apoissonrand.`

These generate random numbers or noise from a Gaussian distribution:

`igaussrand, kgaussrand, agaussrand.`

Filters

Turn a discrete sequence of values into a continuous control signal with:

`port.`

Parametric second-order filters include:

`hipass, lopass, bandpass, bandstop.`

Exactly normative filtering using the canonical second-order section is:

`biquad.`

For IIR all-pass and comb filters of specified delay and feedback gain, use:

`allpass, comb.`

These are general FIR and IIR filters. The first two operate from parametric coefficients; the last two store the coefficients in a stored-function table:

`fir, iir, firt, iirt.`

Spectral Analysis

To perform windowed sliding-block DFTs, placing the result in a stored-function table, use

`fft.`

To perform windowed sliding-block IDFTs, converting spectral frames in stored-function tables into an audio signal, use:

`ifft.`

Gain Control

Calculate the power in an audio signal with:

`rms.`

To rescale an audio signal so that it has specified power, or power that matches a reference signal, use:

`gain, balance.`

To perform parametric power-level compression on an audio signal, use:

`compressor.`

Sample-Rate Conversion

Use the following to decimate an audio signal to a control signal, or upsample and downsample between audio and control signals:

`decimate, upsamp, downsamp.`

To gate an audio signal with a control signal, use:

`samphold.`

Place blocks of samples from an audio signal into a wavetable with:

`sblock.`

Delays

To delay one sample or to delay a specified amount of time, respectively, use:

`delay1, delay.`

A flexible fractional and multi-tap delay-line tool is:

`fracdelay.`

Effects

Apply reverberation, chorusing, flanging, or time shifting (pitch-preserving speed change) to an audio signal with the following:

`reverb, chorus, flange, speedt.`

Tempo Control

To query or set the playback tempo of the orchestra, use:

`gettempo, settempo.`

Appendix 2:
Core Function-Table Generators in SAOL

This appendix contains a complete list of the built-in function-table generators in SAOL. Space does not permit a full description of the parameters, syntax, and semantics of each; interested readers are referred to the standard or to our online documentation. In the standard, the operation of each is defined at the sample-by-sample level. All of these function-table generators must be present in any compliant SAOL implementation:

sample—place a sound sample in a stored-function table
data—place a sequence of specific data values in a stored-function table
random—place random values, drawn from one of several distributions, in a stored-function table
step—place a step function in a stored-function table
lineseg—place a function made up of linear segments in a stored-function table
expseg—place a function made up of exponential curves in a stored-function table
polynomial—place an arbitrary polynomial function in a stored-function table
spline—place a spline curve on a given set of control points in a stored-function table
window—place a window function (Boxcar, Hamming, Bartlett, Kaiser, Gaussian) in a stored-function table
harm—place a sum of zero-phase, harmonically related sinusoids in a stored-function table
harm_phase—place a sum of phased, harmonically related sinusoids in a stored-function table
periodic—place an arbitrary sum-of-sinusoids function in a stored-function table
buzz—place a ban-limited pulse train in a stored-function table
concat—concatenate two or more function tables together, and place the result in a new stored-function table
empty—create space for an empty stored-function table (composers may write i-rate user-defined opcodes that effectively act as user-defined table generators)