GENERALIZED AUDIO CODING WITH MPEG-4 STRUCTURED AUDIO

ERIC D. SCHEIRER AND YOUNGMOO E. KIM

Machine Listening Group, MIT Media Laboratory, Cambridge MA USA

{eds,moo}@media.mit.edu

The MPEG-4 Structured Audio standard was created to enable high-quality, very-low-bitrate transmission of synthetic sound. However, structured-audio techniques also are suitable for flexible natural audio coding. This paper introduces the concept of *generalized audio coding*, in which the Structured Audio decoder is used to emulate the behavior of other audio decoders. We prove that the MPEG-4 Structured Audio tool can be used to mimic the behavior of any other kind of decoder and that structured-audio coding is a *universally minimal* coding technique. We provide examples of simple natural audio coders that use the SA toolset, and characterize the overhead that arises in the transcoding process. Generalized audio coding removes marketplace barriers to the use of special-purpose or signal-adaptive coding formats, and thus promotes greater overall efficiency in the world of audio coding.

INTRODUCTION

The MPEG-4 Structured Audio standard (ISO 14496-3 Section 5), which was completed in 1998, satisfied two goals. First, MPEG-4 requirements specified the need for normative low-bitrate transmission of synthetic sound. Second, the developers who contributed its technical components wished to create an advanced standard, in the hope of driving demand for the inclusion of advanced audio capabilities in future multimedia terminals.

The resulting tool is unusual compared to other audio coding technologies. It specifies a language for sound description and a bitstream format that allows synthesis techniques to be transmitted from sender to receiver, where they are dynamically executed to create sound. The sound-synthesis procedure is tightly specified by the standard (that is, strictly normative), thus ensuring that a given bitstream decodes into the same sound on every terminal. High-quality multichannel sound can be created at extremely low bitrates—less than 2 kbps in most cases. Transmission is lossless—the exact desired sound is constructed sample-by-sample in the decoder.

MPEG-4 Structured Audio (SA) is a new kind of audio coding technique, and its capabilities greatly exceed the original requirements for MPEG-4 audio synthesis. The flexibility of the SA decoding framework allows new synthesis techniques to be explored. Not only direct rule-based methods such as wavetable or FM synthesis, but also analysis-synthesis techniques such as sinusoidal synthesis or noise-shaping methods, can be used. In particular, the gray area between sound-generation methods normally considered *coding* techniques and those normally considered *synthesis* techniques may be explored.

In this paper, we introduce and develop a theory of *generalized audio coding*. In generalized audio coding, a structured-audio system such as SA is used to emulate the behavior of natural audio coders. We prove that the SA codec is, in fact, able to emulate *any* decoding algorithm. This immediately leads to the surprising result, which we also prove, that the SA format is universally minimal—the best coding of a sound in SA cannot be improved by any other coding technology, current or future. As concrete targets for discussion, we present three natural audio decoders: a perceptual transform decoder, an LPC speech decoder, and an LPC variant optimized for the transmission of singing, transcoded as SA bitstreams. We discuss the theory of structured sound and its application to the development of future audio coding standards, and speculate about new sorts of systems that could be used to make the generalized-coding concept more practical.

1. BACKGROUND

This section presents the background for new material that will follow in subsequent sections. We discuss the notion of *structured audio* and the way it interrelates the theories of audio synthesis and natural audio coding. Then we present an overview of the MPEG-4 Structured Audio coding format, for readers who may not be familiar with this tool.

1.1 Theory of structured audio

The term *structured audio* was introduced by Vercoe *et al.* [1] as a way of interrelating research on sound synthesis, audio coding, and sound recognition. The central point of this theory is that every parametric sound representation can be viewed as a tool for sound

understanding, transmission, and *rendering*. This includes not only representations that are typically considered parametric, such as the frequency and amplitude of a sinusoid, but also more complicated ones, such as the particular set of weight values transported as the perceptual coding of a natural audio soundtrack.

By always thinking of sound applications in terms of the parametric space they use, it becomes possible to interrelate and compare techniques from disciplines that are normally disparate. For example, an audio codec such as MPEG-AAC [2] can be considered as the combination of (1) a sound-understanding algorithm that sets the parameters in representation space based on analysis of an acoustic waveform, (2) the transmission of these parameters, and (3) a sound-synthesis algorithm that maps from the transmitted parameters to a new sound. We normally call the understanding step an *encoding* step, and the synthesis step a *decoding* step, but there is no real difference between these styles of terminology.

Similarly, imagine a simple MIDI-stream transmission scenario.[1] A performance on a solo instrument is analyzed by pitch-tracking, and the pitch values are transmitted to a synthesizer, which turns them back into sound. Just as in the example above, there are understanding, transmission, and rendering aspects to this process. Although these scenario is normally considered a "performance" situation, it can also be considered a type of audio compression, particularly if the performance and resynthesis steps are separated by a network.

Recognizing that these two different applications have a similar underlying structure allow us to directly compare the analogous stages of processing. The understanding (encoding) side is less computationally demanding for MIDI than for AAC, but more general for AAC than for MIDI. That is, it is easier to pitch-track a monophonic sound than to perform AAC encoding (both in terms of development difficulty and in terms of run-time complexity), but the AAC encoder can be used on every sound, whereas the pitch-tracking process only applies to a subset of well-defined monophonic sounds. The transmission process requires much less bandwidth for the MIDI case, since the parameterization is very succinct. The rendering process is more exact in the AAC case (for a standard, we would say more *normative*), in that a particular set of parameters is always synthesized into sound in the same way. In the MIDI-transmission case, the particular playback (rendering) method is not normative in the same way, and so different playback devices will present different sonic results.

Vercoe *et al*. [1] presented a taxonomy of many different kinds of sound representations, and the tradeoffs they create when used in applications. For example, continuing the comparison above, we can say that AAC would be preferred to MIDI in applications where normative sound quality is important, a wide set of soundtracks must be represented, there is adequate bandwidth for the transmission of the parameters, and there are enough resources to deploy an encoder. MIDI would be preferred to AAC in applications where normative sound quality is not important, we are only concerned with a narrow set of sounds, and there is restricted bandwidth or encoding resources.

The notion of application-driven (or "requirements-driven") tradeoffs is essential to the present paper. Every coding method presents tradeoffs; there is no "ideal" coding technique except in relation to a particular specific set of requirements. Most high-quality audio research has progressed along one particular path of requirements. As a result, great progress has resulted in this direction; however, we hope to show in the present paper that by taking a broader view, there is progress yet to be realized in the larger marketplace for coding technology.

1.2 MPEG-4 Structured Audio

MPEG-4 Structured Audio is one of the coding tools in the MPEG-4 International Standard, which was completed in 1998 and is scheduled for approval and publication in mid-1999. It is the first standard to allow the direct application of *algorithmic structured audio*[2] techniques to the transmission of sound in a multimedia context. Algorithmic structured audio is the idea of using a general-purpose software-synthesis language, and parameters to programs written in that language, to represent sound for transmission. This idea was suggested as early as 1991 [4], and first implemented in a prototype system called *Netsound* [5].

[1] MIDI, the Musical Instrument Digital Interface specification [3], was originally created as a protocol for controller-synthesizer communications, but can be naturally viewed as a sound parameterization as well. In MIDI, a sound is represented as a sequence of *notes*, each of which is parameterized by its pitch and "velocity," a sort of loudness function. The timbres of sounds are not represented in the MIDI protocol.

[2] Henceforth, Structured Audio or "SA" is used to refer to the MPEG-4 implementation of the structured-audio concept. We will continue to use the phrase *structured audio* with lowercase initials to refer to the theory outlined in the preceding section. These are different things: the former is one particular implementation of a general concept denoted by the latter.

Space in this paper does not permit a full description of the structure and capabilities of the SA format, but we present a brief overview here, and the interested reader is referred to other papers [6-10] or to the standard [11] for more information.

At the heart of the SA standard is a sound-synthesis language called SAOL, for "Structured Audio Orchestra Language" [9]. A program in SAOL describes a sound-processing or sound-synthesis algorithm. SAOL resembles C syntactically, but variables in SAOL contain audio signals, and built-in processing functions allow signals to be generated, mixed, filtered, processed and otherwise manipulated. Examples of SAOL programs, to be discussed later, are in Appendixes A, C, D, and E. As will be proved in Section 2, any algorithm for parametrically creating or altering sound can be described in SAOL.

In order to transmit sound in the SA format, the bitstream header contains one or more algorithms written in SAOL, and the streaming data consists of *Access Units* containing parametric events written in SASL ("Structured Audio Score Language"). The decoding process for the SA bitstream is shown in Figure 1.

At session startup, the SAOL algorithms (called *instrument definitions*) are communicated to a reconfigurable synthesis engine. This engine knows how to interpret the SAOL programs and configure itself for parametric sound manipulation accordingly. The exact properties of the reconfigurable synthesis engine are specified in the MPEG-4 standard; it implements the same functions in every compliant terminal.

During the streaming part of the session, the Access Units are decoded into events, which are stored in a time-sorted list. The *run-time scheduler*, also specified in the standard, keeps track of these events and dispatches them when their time arrives. Each event is either a parametric instruction that begins the execution of one of the SAOL algorithms (a *note*), or some new parameters for use by one of the algorithms that is already executing. At any given time in the decoding process, several *notes* are active and stored in a *note pool*. Each note corresponds to one of the synthesis algorithms delivered in the bitstream header. Multiple notes may use the same algorithm, in which case their data spaces are maintained separately, in the manner of object instances in the object-oriented programming paradigm.

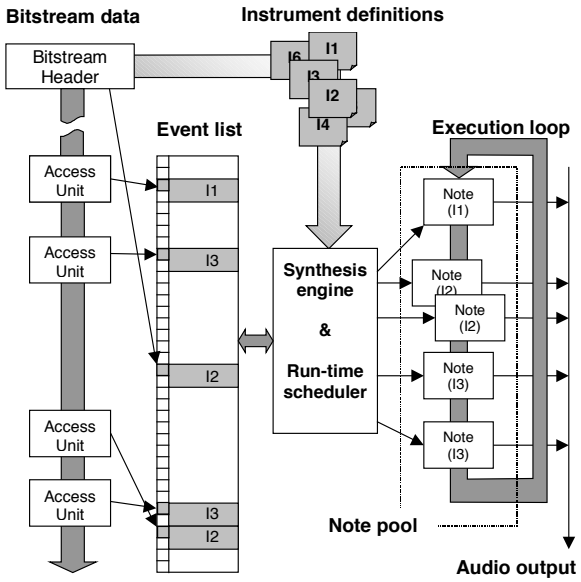

Figure 1: The SA decoding process. The *bitstream header* contains a number of *instrument definitions* written in SAOL. At session startup, the instrument definitions are used to configure a reconfigurable *synthesis engine*. During the streaming session, *access units* containing parametric *events* are used to control the synthesis engine. The *run-time scheduler* keeps track of multiple *note instances*, each corresponding to one of the SAOL instruments, that are executed in turn to produce audio output. Adapted from [10], copyright © 1999 Marcel-Dekker, used with permission.

At each time step, each of the notes that is active in the note pool is executed by running the corresponding SAOL algorithm for a short amount of time. This execution creates some output for each note instance. The outputs of all instances are summed to create the final audio output.

By using a general-purpose software-synthesis language to specify the creation of sound, the normative sound quality required for high-quality applications can be achieved. Since the exact operation of the SA decoding process is specified very precisely, content authors are guaranteed (if they wish) that sounds will be sample-by-sample identical on different MPEG-4 terminals.

Nothing has been written in this section about the nature of the sound-synthesis algorithms transmitted in SAOL and the parametric data that drive them. The text of the standard uses language normally associated with sound-synthesis applications ("instrument," "note," and so forth), but there is no reason that SAOL algorithms must be simple mappings from two or three parameters to rule-based sound generation. The built-in functions of SAOL include spectral transforms, filter structures, noise generators, and other techniques useful for the creation of *natural* sound decoders. Similarly, the transmitted parameters might consist of time-

synchronized frames of data for the control of natural decoders.

SAOL as a language can represent algorithms that are too complex for any particular system. In order to restrict the computational complexity of the decoding terminal, the MPEG-4 standard contains a simulation tool that allows content authors to determine the amount of complexity required to decode their bitstreams. *Levels* of the SA standard are set with regard to complexity as measured with this simulation tool—a Level 1 decoder, in order to conform to the standard, is required to perform a certain amount of computation in real-time, a Level 2 decoder somewhat more, and so forth. The responsibility for ensuring that decoding complexity stays within the bounds of the Level appropriate for the application is left to the content developer.

In practice, many useful Structured Audio bitstreams are actually simpler to decode than bitstreams represented in today's complex wideband audio coders.

2. GENERALIZED AUDIO CODING

This section introduces a theoretical basis for generalized audio coding. It begins with a discussion of the use of models to compress sound for transmission. Following that are the two key proofs of the present paper, which show that SA is capable of delivering any sound model and that it is a universal minimal audio format.

2.1 Model-based audio coding

Every sound-compression technique uses model-based assumptions to guide the process of selecting a parametric representation for a desired sound ("encoding") and turning the representation back into sound ("decoding"). This is necessary because it is impossible for a compression technique to compress all sounds equally with no loss of quality. This impossibility is argued easily with the *counting argument*: there are 2^N different digital sounds that are N bits long in some uncompressed representation. To compress an N-bit sound means to represent it in fewer than N bits. But there are only half as many (2^{N-1}) bitstreams that are even one bit shorter, and in general only the very small proportion

$$1 : 2^{(1-q)N} \qquad (1)$$

of bitstreams are q times as long, where $0 < q < 1$. For example, for any particular model, out of all the 100-bit long sounds there are, only 1 in 2^{50} of them may be losslessly compressed to half its length.

Thus, a sound model for compression achieves two purposes. First, it identifies a particular subset of sounds that may be compressed. This is most obvious in the case of model-based techniques such as CELP or sinusoidal coders. Not every sound may be represented with such a model, and so not every sound can be very much compressed. But the tradeoff that results is that the smaller set of sounds that *can* be compressed with the model are compressed more than they would be with general models. This follows directly from the counting argument: the only way to achieve lossless compression is to admit fewer sounds as candidates for compression.

Second, a sound model for compression embeds a model of psychoacoustics, which asserts that many sounds are identical to each other. If two sounds are perceptually identical, then coding one is the same as coding the other, and the model needs only to be able to compress one of the sounds. Thus, in this manner we eliminate more sounds as candidates for compression.

2.2 Lossy versus lossless coding

Typically, discussion of lossless audio coding—which is traditionally based only on information-theoretic compression of sounds—is kept somewhat distinct from discussion of lossy perceptual coding. In this section, we present an alternative view of lossless coding that highlights the connections between these models more clearly.

It is surprising on the face, but clear, that a lossy perceptual coder *is* actually a lossless coder for a small set of sounds: those sounds that are the possible outputs of the decoder. These sounds happen to have noise in them, masked just below the hearing threshold, happen to be losslessly quantizable with a certain filterbank structure, and so on. If it happens in a particular application that we want to send one of these special sounds, we can do so losslessly with the perceptual coder. In the case of the AAC system at its typical bitrate, these sounds are losslessly compressed by about a factor of 20 by the AAC algorithm. Thus, we can see from (1) that the proportion of such sounds compared to the world of all sounds is very small indeed.

Consider the space of all sounds as a vector space in which perceptually similar sounds are near each other. The goal of designing a perceptual codec is to select a particular (small) subset of losslessly code-able sounds from around the space. We wish this selection to be distributed around the space such that for every sound, there is guaranteed to be a losslessly code-able sound nearby. The design of the sound model in the codec describes the distribution of losslessly code-able sounds and makes the guarantee that each desired sound has a counterpart. It is the role of the encoder to "match up" the desired sound to its best losslessly code-able counterpart.

In this viewpoint, it is not the format that is lossy, but the encoding process. There is no such thing as a "lossy decoder" – every decoder simply produces the sound that the parameter stream describes. The lossy aspect of perceptual coding comes from *substituting* (at the encoder) the actual desired sound for a sound that is perceptually similar, but easier to compress.

2.3 Multi-model compression

In this discussion of sound coding, the sound model is specified as part of a standard, or otherwise given *outside* of the coding process. That is, compression is performed relative to a given model; even if we could discover that some other model was more applicable to a particular sound, we cannot make use of that information in a traditional coding paradigm. We simply must do the best we can with the model we have. The model-based assumptions about the sound are embodied in the bitstream data, and used implicitly by the encoder and decoder, but are not an explicit part of the transaction, as shown in Figure 2.

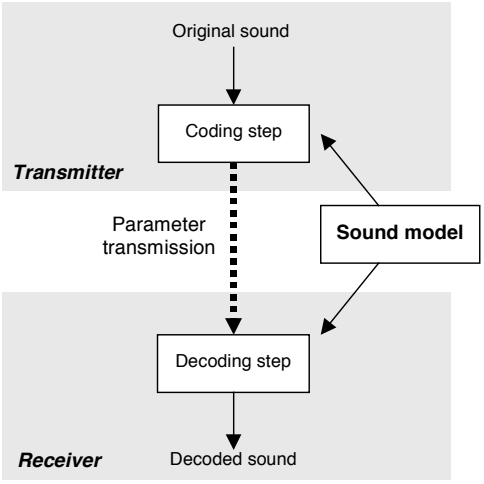

Figure 2: A model-based audio codec uses a fixed model to compress, transmit, and uncompress sound. The model is used at both ends of the transaction, but is standardized outside of any particular session.

Recently, new codecs have been developed with components that can be used or not used depending on the particular sound. The MPEG-4 General Audio coder allows techniques such as temporal noise shaping to be used only when they improve the results of the coding, and for scalability layers to be allocated in varying ways. The MPEG-4 standard as a whole allows different codecs to be selected for different applications. In both of these cases, the model to be used is dynamically selected, but still stands outside the transmitter-receiver relationship, as shown in Figure 3.

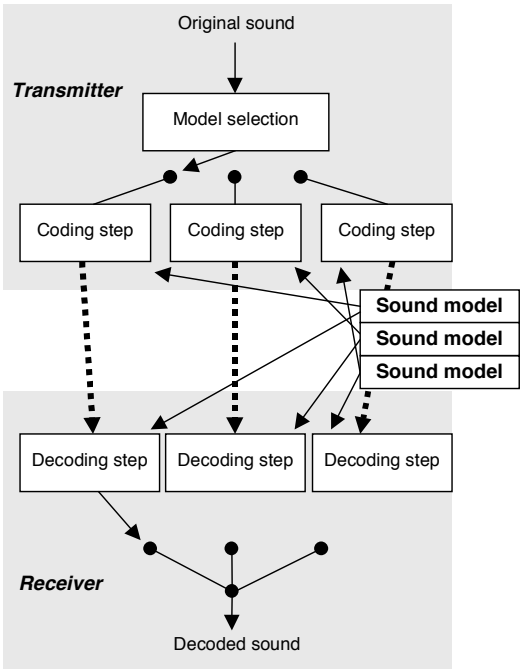

Figure 3: A multi-model codec selects one model from a set of fixed models to compress, transmit, and render sound. The models are used at each end of the transaction, but are standardized outside of any particular session. The models may be variants of one another (as in the configurable elements of MPEG-4 General Audio coding), or completely different techniques (as in the MPEG-4 standard as a whole).

A natural extension to the model-selection technique shown in Figure 3 is to allow the use of multiple models, and to divide a single soundtrack into components. Each component is coded using a different model, and then the results are dynamically mixed together. This is the overall philosophy of the MPEG-4 audio standard.

2.4 Generalized audio coding

In contrast to both of these techniques, in a generalized audio coding method, the model itself is transmitted as part of the bitstream, as shown in Figure 4.

In such a case, we are no longer limited to the model constraints created by the authors of a standard. We can still make use of the advances in previous standards—the sound model in Figure 4 could be the one from AAC or a CELP standard—but we are free to choose another model during the encoding process if we find one that is better.

This is the principal step made in generalized audio coding. There are now well-known methods for coding sound that work very well on certain subsets of sounds, but are difficult to apply to broad audio-coding problems. This leads to a paradox: on one hand, we

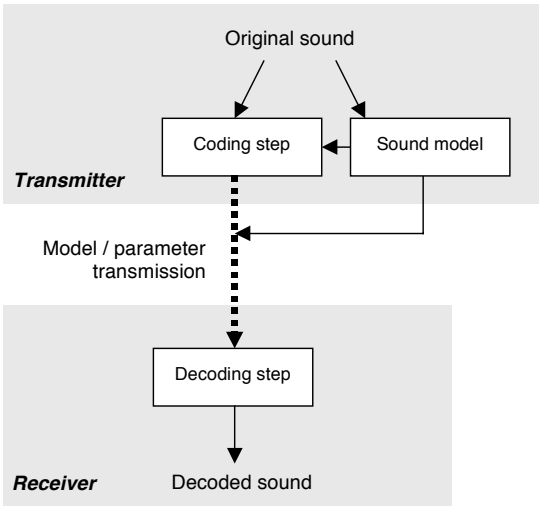

Figure 4: A generalized audio coder transmits its model along with its data. No fixed model is used for the audio coding. The decoder does not know what model is used until a particular session begins.

desire to implement these narrowly-targeted coding techniques so that we may take advantage of them for the right kinds of signals. On the other, it is a great deal of work to create standards for coding technologies and to deploy enough decoders to make this worthwhile.

Rather than creating a standard specifically targeted to each coding technique we might want to use (as in Figure 3), the generalized-audio-coding paradigm suggests that what we should create instead is a flexible *model-description framework* wherein we can make use of *any* technique. In this method, the content provider assumes the responsibility for deciding what methods are useful for compressing his particular sounds; the encoding step is moved outside the bounds of standardization even to a stronger degree than in MPEG audio standards. All that is required from the standardized framework is a method for describing models, and a guarantee that the decoder is flexible enough to use any desired model.

We now proceed to prove that the particular implementation of the structured-audio concept embodied in MPEG-4 Structured Audio can, in fact, produce this guarantee. The application-level discussion of generalized audio coding will be rejoined in Section 4.

2.5 MPEG-4 SA is a universal format

Let X be the set of all finite-length sounds, expressed as binary strings, that is

$$X = \{ [01]^* \} \qquad (2)$$

in the terminology of language theory [12]. For simplicity, the details of sampling rate and word length will be glossed over here. It is clear that with regard to a particular sampling rate and word length, any sound may be unambiguously expressed as a sequence of binary digits $x \in X$.

Let M be a desired sound model that maps from sets of parameters (a bitstream) to a sound. M has multiple variants represented by a "configuration parameter" $c \in \Im$, which chooses one particular variant of the model in M. That is,

$$M_c: \Im^k \rightarrow X, \qquad (3)$$

where $\Im$ represents the natural numbers (0,1,2,...). Assuming that the input domain of M is a finite number k of integers, we can embed the parameter space into the natural numbers and simply say

$$M_c: \Im \rightarrow X, \qquad (4)$$

where the number of dimensions is dropped. Conceptually, c is the decoder configuration and the number $y \in \Im$ represents the entire bitstream that serves as input to a decoder (it might be a very large number!). We then say $M_c(y) = x$ to indicate that the bitstream y decodes under model M_c to sound x. In general $M_c(y)$ is not a total function; the "decoded" sound x is not defined for all values of y relative to a particular M (we might say that the undefined values are the "illegal bitstreams" of M). The inverse function $M^{-1}_c(x)$ is also not a total function; it might be the case that there are some sounds x for M_c such that there is no y satisfying $M_c(y) = x$.

There is a special kind of model U that has the following property:

$$\forall M \; \forall c \; \forall y \; \exists d : U_d(y) = M_c(y). \qquad (5)$$

That is to say, given some other model M_c and an input bitstream y to M_c, no matter what M_c and y are, there is always an configuration of U, U_d, that gives the same result when decoding y as M does. U in this case is termed a *universal* model; the existence of such models was proved by Turing [13].

It is clear that the models associated with traditional audio coders do not have this property. For example, if we take M_c to be a CELP coder configured in a particular way, and wish U to be the AAC model, the test (5) fails. It is not possible to configure the AAC decoder so that we can send it CELP bitstreams and have it decode them properly. This is not because of the surface issues of framing, bitrate, and so forth, but because of the deeper reality that the CELP model is incompatible with the AAC model—it makes different assumptions about, and removes different sorts of information from, the signal being coded.

If we wish to know whether some model *M* is actually a universal model according to (5), there is a straightforward construction that provides sufficiency of proof. A *Turing machine* is a simple model of computation that consists of a scanning head and an infinite tape on which symbols are written. At each time step, the machine inspects the symbol at the current location on the tape according to its internal state. According to a lookup table associating the state and the symbol with a behavior, the scanning head changes the symbol to some other symbol and moves one square to the right or left, and the machine changes to some other state.

A Turing machine T is described using the 5-tuple

$$T = (\Sigma, Q, \delta, s_0, F) \quad (6)$$

where $\Sigma \subseteq \Im$ is a set of *states*, $Q \subseteq \Im$ is a set of *symbols,*

$$\delta: \Sigma \times Q \rightarrow \Sigma \times Q \times \{-1,1\} \quad (7)$$

is the *transition function* mapping from a state and symbol to a state, symbol, and direction, $s_0 \in \Sigma$ is the *starting state,* and $F \subseteq \Sigma$ is the set of *accepting states.*

It is known [12, 13] that any model capable of emulating this behavior is equivalent in computational power to every other such model, and is a universal model according to the definition above (5). Standard texts [12] explain how the TM can be used to describe various kinds of models (or "functions"), and prove that it can be used to describe any model.

We can thus conduct a proof that a particular sound model—the MPEG-4 SA model—is a universal computer via construction. Appendix A contains SAOL code that simulates a Turing machine (see [9] and [11] for extensive details on the operation of SAOL). The Turing machine to be simulated is specified as a parameter file as shown in Appendix B. Any Turing machine may be conveyed as such a set of parameters. Thus, MPEG-4 SA is a universal model capable of emulating any other sound model. As a result, we term it a *generalized audio coder.* □

2.6 Implications

Section 2.3 has the immediate, formal, implication that the SA model $SA_d(y)$ can emulate any other audio coding technique. That is, AAC cannot emulate CELP coding, nor the reverse, but SA can emulate both of them and every other coding model besides. If we want to configure SA so that it can receive AAC bitstreams and properly decode them, this is possible. If we want instead to configure it so that it decodes CELP bitstreams and decodes them, this is also possible. This property makes the SA format unique among today's audio coders.

More importantly, it allows us to prove an interesting result regarding the length of SA bitstreams. SA is an *universal minimal* audio coding format; no other coding technique can ever produce bitstreams that are smaller than the best SA coding by more than a constant that is independent of the length of the stream. As bitstreams become long, the significance of this constant vanishes.

Define the *bitstream length* associated with the decoding process $M_c(y)$ as

$$L\{M_c(y)\} = \log c + \log y. \quad (8)$$

This definition makes intuitive sense; if we want to send the configuration parameter and the bitstream over a wire to a decoder, it takes $\log c$ bits[3] to describe the configuration parameter and $\log y$ bits to send the bitstream.[4]

Suppose there is some SA coding y of a sound x that uses a model d. That is,

$$SA_d(y) = x, \quad (9)$$

and this coding has length $l = \log d + \log y$. Using some other coder M, we find a better coding, where

$$M_c(z) = x \quad (10)$$

and

$$L\{M_c(z)\} < L\{SA_d(y)\}. \quad (11)$$

From (5) and the universality of SA, we know there must be a configuration of SA d' such that

$$SA_{d'}(z) = M_c(z) = SA_d(y) = x. \quad (12)$$

That is, configuration d' allows SA to decode bitstream z and get the same sound x. The length of this coding is

$$L\{SA_{d'}(z)\} = \log d' + \log z, \quad (13)$$

and so

$$L\{SA_{d'}(z)\} - L\{M_c(z)\} = \log d' + \log z - (\log c + \log z) \quad (14)$$

$$= \log d' - \log c\ ; \quad (15)$$

[3] All logarithms in the present paper are base 2.

[4] If, as in our original definition (3), we view the bitstream as an ordered k-tuple of values, each in $\{1..z\}$, then each takes $\log y$ bits to describe and the ensemble takes $k \log z$. If we compose the k values into one, the composed value y is in $\{1..z^k\}$ and the composed value itself takes

$$\log y = \log z^k = k \log z$$

bits to transmit, and so no space has been mysteriously saved via this composition. A similar argument holds for considering the configuration parameter as a n-tuple of values instead of one large value.

that is, the lengths of the two codings vary only in the lengths of the configuration parameters (in fact, it is possible that the SA coding is strictly shorter).

More importantly,

$$\lim_{z\to\infty} \frac{L\{SA_{d'}(z)\}}{L\{M_c(z)\}} \tag{16}$$

$$= \lim_{z\to\infty} \frac{\log d' + \log z}{\log c + \log z} \tag{17}$$

$$= 1. \tag{18}$$

As the bitstream z becomes long, the lengths of the two codings become more and more similar.

Thus, whenever we find a coding model that is better than the best one we know about for SA, we immediately know that there is a new configuration of SA that achieves the same quality at the same bitrate (in the limit). In this sense, SA is a universal minimal coding format. □

This proof holds only because the universality property (5) maintains for the SA compression model. For formats that do not have this property, we cannot make the crucial configuration step (12). Conceptually, the configuration parameter d' represents the entire model M_c—that is, we program the SA decoder, via configuration d', to emulate the desired model M_c, and then run the emulator to decode the bitstream. The universality of SA guarantees that any M_c may be so emulated.

There is some universally minimum-length bitstream to code the sound x in SA; since there are only a finite number of combinations d and y for coding sounds, one of them must be the smallest. By the proof above, not only SA, but every other coding method, is bound within a constant of this length. In theoretical computer science, the compressibility is a measure of the randomness of the sound x and is termed the Kolmogorov complexity [14]. This value is generally of theoretical interest only, since it is formally impossible to determine the Kolmogorov complexity of a sound in general.

3. EXAMPLES

The preceding section presented a theoretical treatment of the generalized audio coding paradigm. However, there are practical issues of interest in this area as well: how large is the constant d' in (13), and how difficult is it to perform the emulation? In this section, we demonstrate emulators for two simple types of coding, an LPC speech model and a perceptual transform coder. Neither of the coders presented here are as sophisticated as the state-of-the-art implementations of these techniques, but the examples will suffice as a demonstration of the reasonable bounds of the practical issues involved.

Following these two standard types of coding, we present a third example, which is a nSA implementation of a novel singing-voice codec recently proposed by Kim [15]. The primary advantage of the generalized-audio coding framework is demonstrated with this example: we can rapidly apply new developments in narrowly-focused sound coding technology without having to go through the standardization process.

3.1 Perceptual transform coding

Appendix C contains SAOL code that implements the MPEG-1 Layer I decoder. By using this header file, natural audio, perceptually compressed at any desired bitrate, may be incorporated in a Structured Audio bitstream. The individual frames of MP1 data are stored as blocks of 16-bit samples and streamed into the SA decoder as wavetables. The **unpack()** function in the SAOL code does the bit-unpacking necessary to convert this to scaled subband data, which is then synthesized in the main instrument code.

A simple decoder such as this can be transmitted very efficiently. The entire decoder is compressed in the binary SAOL bitstream format to only 5 KB of data, including the large synthesis-window table that is not shown in Appendix C. The continuous transport overhead is actually smaller here than in the actual MPEG-1 audio standard, since the frame header is not used.[5]

It is readily apparent that signal-dependent modifications could be made to the decoder in order to achieve greater efficiency. For example, it would be easy to change the frame size or the number of subbands used, simply by modifying the decoder appropriately. More significant modifications are also possible; the exact nature and utility of such *signal-adaptive perceptual transform coding* is left as a topic for future research.

There is no inherent run-time complexity overhead due to implementing the decoder algorithms in SAOL rather than compiling them from a C program. For the time being, C implementations will be somewhat more efficient than SAOL implementations of the same algorithms, but this is only because today's C compilers are better than today's SAOL compilers. In the long run, it is conceivable that SAOL implementations would be more efficient in general for equivalent decoding

[5] Although the MPEG-4 bitstream containing the MP1 bitstream would itself have transport overhead due to MPEG-4 stream packaging.

algorithms, since it may be easier to compile audio algorithms in SAOL to high-performance architectures (see also Section 4.3). There is more session-startup complexity in the SA case, since the decoder must process and compile the SAOL code before it can begin decoding the bitstream.

It is also interesting to note that the content author now plays a role in determining the decoding efficiency—the delivered Layer I decoder may be a more or less efficient one, and this is an independent consideration from the efficiency of the MPEG-4 terminal that embeds the decoder. Clearly, the term "content author" in this case must encompass not only the musician creating the artistic part of the content, but the engineers who create and sell efficient SAOL-based decoder implementations.

3.2 LPC-based speech coding

Appendix D contains SAOL code that implements a simple linear-predictive decoder for speech coding. Low-bitrate speech, on the order of 6.3 kbps, is easily acheivable with this codec.

It is readily apparent from the main part of this decoder (the **lpc** instrument) that the description of operations such as pulse-train generation (with the SAOL **buzz** operation) and digital filtering (with **iirt**) is very concise in SAOL.

3.3 A singing-voice codec

As a final example, a novel singing-voice codec recently developed by Kim [15] is demonstrated in Appendix E. This codec can be considered a further specialization of the general LPC technique—by assuming that the domain of signals is that of singing in the *bel canto* style, we can make further reductions in bitrate. In Kim's singing coder, this is achieved by using a codebook to represent the allpole filter shapes for the different vowels. Since most sound in singing is voiced vowels, for most frames we only need to transmit the pitch, the gain, and an index into the codebook of filter shapes. Approximately 75% of the LPC filter shapes can be replaced with a single 4-bit index in this way. Vowel-to-vowel transitions and consonants are coded using the standard LPC method.

Solo singing samples can be compressed more tightly with this technique than with standard LPC, with comparable quality. The implementation in Appendix E gives good performance at an average of about 2.3 kbps, nearly a factor of three compared to the basic LPC implementation in Appendix D. The decoder here, plus the codebook tables needed, are about 365 bytes longer than the decoder in the previous example; therefore, this technique allows improvement in average coding efficiency whenever the segment to be coded is longer than about 1.25 seconds.

4. DISCUSSION

Sections 2.2, 2.3, and 3 provide a theoretical and practical basis for the claim that MPEG-4 Structured Audio is a generalized audio coder, and that it is reasonably practical to imagine using SA to emulate other audio coding methods. In this section, we return to the "requirements-level" discussion of generalized audio coding, and describe the advantage this paradigm supplies in the overall world of coding.

4.1 Generalized audio coding in the marketplace

The generalized-audio-coding paradigm has been developed to answer a particular marketplace-level question regarding new audio compression technology. This question is: given the current state of the art in speech coding and wideband audio coding, what should be done with new advances in coding research? General policy in ISO-MPEG when considering new codec technology is that the new technology should never perform worse than the current verification model, even if it performs better in some cases. This policy is well-conceived because it keeps MPEG standards from exploding in a multitude of different coders, each optimized for a special set of test signals. On the other hand, it is undeniably inefficient in a strict compression-quality sense to not be able to take advantage of new advances in narrowly-focused technology when they arise.

Compression rarely takes place in a vacuum. In nearly every case where coding and compression is needed, we know more about the structure of the sound than we tell the encoder—for example, we often know that a particular sound is a piece of classical music, or that it contains a vocal track, or that it is very repetitive. It is quite reasonable to believe that natural coding technology could be more optimized than it currently is if it were possible to take such structural knowledge into consideration. The example in Section 3.3 is one case of this—by understanding constraints that apply to the singing voice but not to the speaking voice, we can design a codec that is more efficient for the singing voice than for the speaking voice.

The problem with such a codec is that under the current practice of standardizing natural coding technology, it will not be standardized, since there are many sorts of signals (speech signals) on which it performs worse than less specific codecs. It therefore cannot be used for real applications. Generalized audio coding solves this problem by allowing any encoding model to be used for the representation of sound, and then relying on the delivery of decoders in a algorithmic-structured-audio

framework to perform the decoding. Since any decoder may be transmitted in this way, any sound representation may be used for a particular soundtrack.

In the generalized audio coding framework, we *augment* the use of coding techniques that are broadly tuned to work fairly well in most circumstances with other techniques that are narrowly tuned to work extremely well in narrow circumstances. Since the generalized decoder can be strictly standardized (as is the MPEG-4 Structured Audio decoder) and restricted by Levels as described in Section 1.2, this paradigm creates no new "dangers" to sound quality or decodability. It enables new research and deployment opportunities that are not available in the current world of standardization.

As discussed in Section 1.1, making a standard for sound compression inherently involves tradeoffs. There is no "ideal" solution to a compression problem except with respect to a particular set of requirements. The sort of codec embodied by the traditional perceptual coder chooses a particular set of requirements: we wish the decoder to have a fixed complexity and to be easy to implement, the encoding process to be straightforward, and the same technique to work well on every sound. Within these restrictions, we try to choose the best quality-for-bandwidth solution that we can.

In contrast, the generalized audio coding paradigm suggests that the decoder may be difficult to implement, and we might be satisfied with a *maximum* decoding complexity rather than a fixed decoding complexity. Further, we must be willing to think harder about the encoding process and how to best achieve the optimal bitrate for the sound quality we require in some application. But in return for this extra effort, a broad range of signals can be much more efficiently encoded than is possible with today's standards.

4.2 On the SA encoding problem

The framework presented here provides a new perspective on the problem of encoding bitstreams for structured-audio transmission. There has been natural reluctance to embrace the structured-audio technique as a general-purpose audio coder, because it has been viewed as dependent on difficult encoding problems such as sound-source separation [16] and automatic music transcription [17, 18]. This is not the case; structured-audio techniques allow us to access a wide variety of encoding techniques.

On one side of the spectrum of sound representations [1], there are those encoding schemes that are relatively close to the signal, such as perceptual transforms. These representations are relatively less efficient than more-structured methods, but easier to encode automatically and to apply to a broad range of signals. On the other side, there are fully structured representations such as musical-instrument-synthesis algorithms. These representations are extremely efficient, but are currently impossible to encode automatically.

In the middle of this spectrum is a largely unexplored territory of partly- or mostly-automated analysis-synthesis coding systems. Using a generalized audio decoder allows us to explore these types of sound representations within a single decoding framework. It allows us to think about many different types of sound representations as possible encoding formats. For example, a recent development is the new wave of sinusoidal analysis-synthesis tools [19, 20]. This sort of codec is ideally suited to implementation as an SA bitstream, since the components (sinusoids and filtered noise) are easily and efficiently generated in SAOL.

For example, a cross-coder for the computer-music composition format called SDIF (Sound Description Interface Format) [21] has recently been developed [22]. SDIF contains a variety of analysis/synthesis representations packaged in a single format, such as sinusoidal, LPC, phase-vocoder, and spectral-frame representations. Through cross-coding to MPEG-4, compositions created or represented in the SDIF format may be played back with the SA decoder. The resulting application is something of a hybrid encoder/authoring tool, since it allows natural sounds to be transformed into compressed representations, but also allows composers to specify manipulations to be applied as part of the playback process.

4.3 General-purpose computer languages

A natural extension to the ideas proposed in the present paper is to consider the use of a general-purpose computer language such as Java rather than a special language designed for structured-audio description. Perhaps in the future, general-purpose computing cycles will be cheap enough that significant portions of the host processor can be devoted to audio decoding, which seems required if a general-purpose language is used.

A sound-description language like the MPEG-4 Structured Audio format has the advantage that it already behaves like an audio decoder—it accepts blocks of data, communicates with a DAC, runs in realtime, and so forth. While not theoretically impossible, general-purpose programming languages do not satisfy the systems-level requirements today. There is no portable way to implement the connection to the DAC, to receive streaming data from a network connection, and so forth.

Further, the Structured Audio standard is written with implementation on custom hardware in mind—that is, in

which the entire decoder moves onto a DSP chip or similar multimedia acceleration card. Since the fundamental constructs of SAOL (signals, filters, delays) are those which are efficiently implemented on a DSP, it is likely that it will soon be possible to compile SAOL to DSP processors so that it runs more efficiently than C or Java implementations of the same algorithms. This has the dual advantages of being intrinsically more efficient and moving the computation off of the host and onto sideboard processors.

5. FUTURE DIRECTIONS

The results in the present paper demonstrate that theoretically and practically, the generalized-audio-coding model is a viable one for leveraging new advances in coding technology to the compression of real sounds. What is not demonstrated is any *particular* coding algorithm that compellingly requires use of this paradigm.

This is the main topic for future research, and it seems primarily divided into two areas. First, research into signal-adaptive coding, where the filterbank structure, quantization technique, or organization of coding blocks change radically in response to the properties of the input signal. Second, research into narrowly-focused sound models that can provide extremely high compression and sound quality for a narrow, but still useful, subset of audio signals. Either or both of these types of coding advances translate naturally into models for use with generalized audio coders.

In a more long-term sense, it remains to be determined whether the MPEG-4 Structured Audio toolset is the best possible one for description of sound models. Since the standard was not designed with this application specifically in mind, it might well be the case that a different sound-modeling language with other constructs and properties (bitwise operators would be very helpful for bitstream decoding!) is more well-suited to the delivery of natural-sound coding algorithms.

REFERENCES

[1] Vercoe, B. L., Gardner, W. G., Scheirer, E. D. 1998. Structured audio: The creation, transmission, and rendering of parametric sound representations. Proceedings of the IEEE, 85, 5, pp. 922-940.

[2] Bosi, M., Brandenburg, K., Quackenbush, S., Fielder, L., Akagiri, K., Fuchs, H., Dietz, M., Herre, J., Davidson, G., Oikawa, Y. 1997. ISO/IEC MPEG-2 advanced audio coding. Journal of the Audio Engineering Society, 45, 10, pp. 789-814.

[3] MIDI Manufacturers Association. 1996. The Complete MIDI 1.0 Detailed Specification. Protocol specification, Los Angeles, MIDI Manufacturers Association.

[4] Smith, J. O. 1991. Viewpoints on the history of digital synthesis. In Proc. 1991 Int. Computer Music Conf., Montreal. pp. 1-10.

[5] Casey, M. A., Smaragdis, P. J. 1996. Netsound: Real-time audio from semantic descriptions. In Proc. 1996 Int. Computer Music Conf., Hong Kong. pp. 143.

[6] Scheirer, E. D. 1998. The MPEG-4 Structured Audio standard. In Proc. 1998 IEEE Int. Conf. Acoust. Speech Sig. Proc., Seattle, May 1998. pp. 3801-3804.

[7] Scheirer, E. D., Ray, L. 1998. Algorithmic and wavetable synthesis in the MPEG-4 multimedia standard. In Proc. 1998 105th Convention of the Audio Engineering Society (reprint #4811), San Francisco, Sept 1998.

[8] Scheirer, E. D. 1999. Structured audio and effects processing in the MPEG-4 multimedia standard. Multimedia Systems, 7, 1, pp. 11-22.

[9] Scheirer, E. D., Vercoe, B. L. 1999. SAOL: The MPEG-4 Structured Audio Orchestra Language. Computer Music Journal, 23, 2, pp. 31-51.

[10] Scheirer, E. D., Lee, Y., Yang, J.-W. in press. Synthetic audio and SNHC audio in MPEG-4. In Advances in Multimedia: Systems, Standards, and Networks, A. Puri and T. Chen, eds. Marcel Dekker, New York.

[11] International Organisation for Standardisation. 1999. Coding of multimedia objects (MPEG-4). International Standard ISO/IEC 14496:1999, Geneva, ISO.

[12] Hopcroft, J. E., Ullman, J. D. 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA.

[13] Turing, A. M. 1936. On computable numbers with an application to the *Entscheidungsproblem*. Proc. of the London Math. Soc., Series 2, 42, pp. 230-265.

[14] Li, M., Vitányi, P. 1996. An Introduction to Kolmogorov Complexity and its Applications. Springer, New York.

[15] Kim, Y. E. 1999. Structured encoding of the singing voice using prior knowledge of the musical score. In Proc. 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, Mohonk, NY, Oct. 1999.

[16] Rosenthal, D. F., Okuno, H. G., eds. 1998. Computational Auditory Scene Analysis. Lawrence Erlbaum, Mahweh, NJ.

[17] Martin, K. D., Scheirer, E. D. 1997. Automatic transcription of simple polyphonic music: Integrating musical knowledge. In Proc. 1997 Society for Music Perception and Cognition.

[18] Martin, K. D. 1996. Automatic transcription of simple polyphonic music: Robust front-end processing. MIT Media Laboratory Perceptual Computing Technical Report #399, Cambridge MA.

[19] Edler, B., Purnhagen, H., Ferekidis, C. 1996. ASAC: Analysis/synthesis audio codec for very low-bit rates. In Proc. 1996 100th Convention of the Audio Engineering Society (reprint #4179), Copenhagen, May 1996.

[20] Levine, S. N. 1998. Audio Representations for Data Compression and Compressed Domain Processing. Ph.D. thesis. Stanford University CCRMA, Palo Alto, CA.

[21] Wright, M., Choudhary, A., Freed, A., Wessel, D., Rodet, X., Virolle, D., Woehrmann, R. 1998. New applications of the Sound Description Interchange Format. In Proc. 1998 Int. Computer Music Conf., Ann Arbor, MI. pp. 276-279.

[22] Wright, M., Scheirer, E. D. 1999. Cross-coding SDIF into MPEG-4 Structured Audio. In Proc. 1999 Int. Computer Music Conf., Beijing.

APPENDIX A. SAOL TURING MACHINE

This code implements a Turing machine in SAOL, for an alphabet of two symbols (0 and 1). The binary sequence left on the tape at the end of the program is assumed to be an N-times oversampled audio PCM sequence and is played out as data. The particular program is specified in a SASL score file as given in Appendix B.

```
#define MAX_OUTPUT_SIZE 3000
#define SHIFT 20  // maximum negative length on the tape
#define NUM_SYM 2 // size of the alphabet

global {
  srate 32000;
}

instr turing() {
  imports table state;
  imports table sym;
  imports table shift;
  imports table end_states;
  imports ksig start_state;
  imports ksig overfactor;

  imports table tape;
  ksig end_time,kct,run;
  ksig cur_state, cur_pos, index, newsym, delta, trans;
  ksig max_pos;
  table out(empty,MAX_OUTPUT_SIZE);

  // on the first k-cycle, run the program
  if (!itime) {
    cur_state = start_state;
    cur_pos = 0;

    while (!in_table(end_states,cur_state)) {
      index = cur_state * NUM_SYM +
                  tableread(tape,cur_pos+SHIFT);

      trans = tableread(state, index);
      newsym = tableread(sym,index);
      delta = tableread(shift,index);

      tablewrite(tape,cur_pos+SHIFT,newsym);
      cur_state = trans;
      cur_pos = cur_pos + delta;
      if (cur_pos > max_pos) { max_pos = cur_pos; }
      run = run + 1;
    }
    end_time = max_pos / s_rate / overfactor;
    reduce(tape,out,overfactor);
  }

  // atime; stream out the answer
  kct = kct + 1/k_rate;
  output(myoscil(out,max_pos/overfactor));
}


opcode in_table(table ft, xsig x) {
  xsig pos;
  pos = 0;
  while (pos < ftlen(ft)) {
    if (tableread(ft,pos) == x) { return(1); }
    pos = pos + 1;
  }
  return(0);
}

opcode reduce(table x, table y, xsig factor) {
  // reduce table x to table y by decimating it by averaging.
  xsig pos, fct, sum;
  while (pos < ftlen(x)) {
    sum = 0;
    fct = 0;
    while (fct < factor) {
      sum = sum + tableread(x,pos);
      pos = pos + 1;
      fct = fct + 1;
    }
    tablewrite(y,pos/factor,sum/factor);
  }
}
```

APPENDIX B. TURING MACHINE PROGRAM

This SASL code is used with the SAOL program in Appendix A, as an example of how to specify Turing machines. Any computable algorithm may be specified in this manner. (This is not to imply that it is the *only* way to specify algorithms – real SAOL programs are much more concise and understandable. It only serves to prove in a formal sense that every sound-coding algorithm may be transmitted in this manner.)

The **state** table contains a pair of elements for each state in the TM. The first gives the state transition on a "0" symbol, and second the state transition on a "1" symbol. The **sym** table contains similar pairs, which are the new symbols to write on the tape. The **shift** table contains similar pairs, which are the direction to shift the table after writing the new symbol (-1 = left, 1 = right). The **end_states** table contains a list of the end states of the TM. The **tape** table allocates sufficient space for the calculation.

According to the 5-tuple definition of TM given in Section 2.5, Σ is given by the length of **state**:

$$\Sigma = \{0.. |\mathbf{state}|/2\}.$$

Q is always {0,1} in this version; there is no additional power, only convenience, provided by allowing a richer alphabet [12]. δ is given by **state**, **sym**, and **shift:**

$$\delta(s,q) = (\mathbf{state}[2s + q], \mathbf{sym}[2s + q], \mathbf{shift}[2s + q]).$$

s_0 is given in the **control start_state** line. F is given in **end_states**.

After the table allocations, the **turing** line in the score starts the TM; the **control start_state** line tells it what state to begin in; and the **control overfactor** line tells it what the oversampling factor of the resulting table is.

```
0 table state data -1 0 0 2  3  3  4  1  2  5  6  3  1  0  0
0 table sym data   -1 0 0 1  0  1  1  1  0  0  1  1  1  0  0
0 table shift data -1 0 0 1 -1  1  1 -1  1  1  1 -1  1  0  0
0 table end_states data -1 6
0 table tape empty 2000

// This is a 5-state "busy beaver" machine.
// It runs 134467 steps and then halts.
// This is no longer close to the best known
// 5-state busy beaver, which runs for more than
// 47 million steps before halting!

t1: 0 turing 1
    0 t1 control start_state 1
    0 t1 control overfactor 1
```

APPENDIX C. SAOL MP1 DECODER

This code implements a decoder for the MPEG-1 Layer I audio standard. This implementation is the most straightforward one possible, and thus is somewhat inefficient. Note that the sampling rate is irregular in order to make the block rate integral, which is required in SAOL. This is a drawback of the SAOL cross-coding.

```
global {
   srate 44160;
   krate 115;   // 384 samples per frame
}

instr mp1() {
  imports table mp1_table;
  table synwin(data,512,[...]); // whole table there

  ksig subband[384];
  ksig i,j,v[1024],u[512],w[512],n[2048],s,samp,k;
  table sound(empty,384);
  asig snd;

  // i-rate
  i=0; while (i < 64) {
     k=0;while(k<32) {
       n[i*32+k] = cos((16+i) * (2*k+1) * 3.1415926535/64);
       k=k+1;
     }
     i=i+1;
  }

  // k-rate

  // we got a new mp1_table since last k-cycle
  // first, unpack it and get the subband data

  unpack(mp1_table,subband);

  // now, do the synthesis -- very simple, exactly
  // according to the standard
  samp = 0; while (samp < 12) {

    // shifting
    i=1023; while (i>63) { v[i] = v[i-64]; i=i-1; }

    // matrixing
    i=0; while (i<64) {
      v[i] = 0;
      k=0; while (k<32) {
         v[i] = v[i] + subband[samp*32+k] * n[i*32+k];
         k =k + 1;
      }
      i = i + 1;
    }

    // build U vector
    i=0; while (i < 8) {
      j=0; while (j < 32) {
         u[i*64+j] = v[i*128+j];
         u[i*64+32+j]=v[i*128+96+j];
         j = j + 1;
      }
      i = i + 1;
    }

    // convolve with synthesis window
    i=0;while (i<512) {
      w[i] = u[i] * tableread(synwin,i);
      i = i+1;
    }

    // calculate output samples
    j = 0; while (j < 32) {
      s = 0; i = 0; while (i < 16) {
         s = s + w[j+32*i];
         i = i + 1;
      }
      // put the output tables in a wavetable for playback
      tablewrite(sound,samp*32+j,s);
      j = j + 1;
    }
    samp = samp + 1;
  }

  // a-rate: play back the sound in the wavetable
  snd = oscil(sound,k_rate);
  output(snd);
}
```

```
kopcode unpack(table frame, ksig sb[384]) {
  ksig alloc[32];  // bit allocation for each subband
  ksig sf[32];     // scalefactor for each subband
  ksig i,j,t,v,x;
  oparray getnextval[1];

  i = 0; while (i < ftlen(frame)) {
    // turn float values back to big integers
    tablewrite(frame,i,floor(tableread(frame,i)*32768+32768));
    i = i + 1;
  }

  // reset the bitstream parser to the beginning of the table
  getnextval[0](frame,-1);

  // get the bit allocations
  i=0; while (i < 32) {
    alloc[i] = getnextval[0](frame,4);
    i = i + 1;
  }

  // get the scalefactors
  i = 0; while (i < 32) {
    if (alloc[i] > 0) {
      v = getnextval[0](frame,6);
      sf[i] = pow(2,1-v/3);
    }
    i=i+1;
  }

  // get the subband values
  i=0; while (i < 12) {
     j = 0;
     while (j < 32) {
       if (alloc[j] > 0) {
          t = getnextval[0](frame,alloc[j]);
          v = (t-pow(2,alloc[j]-1)) / pow(2,alloc[j]-1);
          x = (v + pow(2,-alloc[j])) / (1-pow(2,-alloc[j]));
          sb[i*32 + j] = x * sf[j];
        }
        else {
          sb[i*32 + j] = 0;
        }
        j = j+1;
      }
    i = i + 1;
   }
}


kopcode getnextval(table frame, ksig nbits) {
  ksig fpos, bpos, val, hi, lo, gap;

  // get nbits from the frame -- push fpos and bpos forward
  // as needed.  fpos keeps track of the current word,
  // bpos keeps track of the next bit within that word

  if (nbits < 0) { // reset
     fpos = 0; bpos = 0;
     return(0);
  }

  // shifts implemented as pow(2,x) operations
  if (bpos + nbits <= 16) {
     bpos = bpos + nbits;
     val = mod(floor(tableread(frame,fpos) /
                  pow(2,16-bpos)),pow(2,nbits));
     if (bpos == 16) { bpos = 0; fpos = fpos + 1; }
  } else {
     gap = nbits - (16-bpos);
     hi = mod(tableread(frame,fpos),pow(2,nbits-gap));
     lo = floor(tableread(frame,fpos+1) / pow(2,16-gap));
     bpos = gap;
     fpos = fpos + 1;
     val = hi * pow(2,gap) + lo;
  }

  return(val);
}

opcode mod(xsig x, xsig y) {
  return(floor(x - floor(x/y)*y + 0.5));
}
```

APPENDIX D. SAOL LPC DECODER

This is a simple LPC speech decoder implemented in SAOL. It allows medium-quality speech coding at a maximum bitrate of 6.3 kbps. Each frame contains a voiced/unvoiced parameter, an optional 7-bit voicing period, an 8-bit gain, and 10 12-bit filter coefficients. Each frame represents 500 samples of data at 11 kHz sampling rate. Overlap-and-add synthesis of frames is used for smooth playback.

```
global {
  srate 11000;
  krate 44;
}

instr lpc() {
  imports table lpc_table;
  table frame(empty,50);
  table frame2(empty,50);
  table ff(data,1,1);
  table hann(window,500,2);
  tablemap overlap(frame,frame2);
  ksig pitch, spitch, order, gain, iter;
  table coeffs(empty,50);
  asig glottal, out[2], win[2];
  oparray buzz[2], iirt[2], oscil[2];
  asig i; ksig v;

  // k-time

  // the 'lpc_table' table got updated since last pass
  // undo the packaging & scaling
  unpack(lpc_table, v, pitch, gain, coeffs);

  order = 10;

  // copy it into a local table -- we use two of them in
  // alternation to manage the overlap-add

  iter = 1-iter;
  if (iter) {
  copy_table(frame, coeffs);
  } else {
  copy_table(frame2, coeffs);
  }

  // smooth the pitch signal
  spitch = port(pitch,0.005);

  // Determine voicing
  if (v == 1) {
    // make a glottal pulse signal
    if (pitch > 0) {
       glottal = buzz(s_rate/spitch,-1,0,0.95);
    }
  }
  else { // use white noise as voicing
    glottal = arand(1)/10;
  }

  // multiply it by the gain
  glottal = glottal * gain;

  // filter the glottal-pulse signal with each of the
  // two active allpole filters (this frame, and the previous
  // frame)
  i = 0; while (i < 2) {
    out[i] = iirt[i](glottal,overlap[i],ff,order);
    i = i + 1;
  }

  // these are envelopes that handle the overlap-add
  win[0] = oscil(hann,k_rate/2)/9;
  win[1] = delay(oscil(hann,k_rate/2)/9,1/k_rate);

  // mix the outputs of the two filters
  output(win[0] * out[0] + win[1] * out[1]);
}


kopcode copy_table(table t1,table t2) {
  ksig i;

  i = 0; while (i < ftlen(t2)) {
    tablewrite(t1,i,tableread(t2,i));
    i = i + 1;
  }
}
```

```
kopcode unpack(table frame, ksig v, ksig pitch,
               ksig gain, table coeffs) {
  ksig i,val;
  oparray getnextval[1];

  i = 0;
  while (i < ftlen(frame)) {
    // turn float values back to big integers
    tablewrite(frame,i,floor(tableread(frame,i)*32768+32768));
    i = i + 1;
  }

  // reset the bitstream parser to the beginning of the table
  getnextval[0] (frame,-1);

  // get voicing decision
  v = getnextval[0] (frame,1);

  if (v == 1) { // voiced frame
    // get pitch (7 bits)
    pitch = getnextval[0] (frame,7);
  } else { // unvoiced frame
    pitch = 0;
  }

  // get gain value (8-bits)
  gain = getnextval[0] (frame,8) * 5/256;

  // get the filter coefficient values (10 coefficients,
     12-bits each)
  i = 0; while (i < 10) {
    val = getnextval[0](frame,12);
    tablewrite(coeffs,i+1,-(val * 10 / 4096 - 5) );
    i = i + 1;
  }
}


kopcode getnextval(table frame, ksig nbits) {
  ksig fpos, bpos, val, hi, lo, gap;

  // get nbits from the frame -- push fpos and bpos forward
  // as needed.  fpos keeps track of the current word,
  // bpos keeps track of the next bit within that word

  if (nbits < 0) { // reset
     fpos = 0; bpos = 0;
     return(0);
  }

  // shifts implemented as pow(2,x) operations
  if (bpos + nbits <= 16) {
     bpos = bpos + nbits;
     val = mod(floor(tableread(frame,fpos) /
                  pow(2,16-bpos)),pow(2,nbits));
     if (bpos == 16) { bpos = 0; fpos = fpos + 1; }
  } else {
     gap = nbits - (16-bpos);
     hi = mod(tableread(frame,fpos),pow(2,nbits-gap));
     lo = floor(tableread(frame,fpos+1) / pow(2,16-gap));
     bpos = gap;
     fpos = fpos + 1;
     val = hi * pow(2,gap) + lo;
  }

  return(val);
}

opcode mod(xsig x, xsig y) {
  return(floor(x - floor(x/y)*y + 0.5));
}
```

APPENDIX E. SINGING-VOICE DECODER

This coder allows singing voice sounds to be efficiently coded, by using the observation that many filtershapes are nearly the same, and can therefore be transmitted as codebook indices [15]. The resulting bitstream is smaller on average by a factor of 3 than one that results from using the codec in Appendix D, for similar quality. It is only appropriate for the coding of singing.

```
global {
   srate 11000;
   krate 44;
}

instr lpcplus() {
  imports table lpcplus_table;
  imports table A_table;
  imports table E_table;
  imports table U_table;
  table nA_table(empty,9);
  table nE_table(empty,9);
  table nU_table(empty,9);
  table frame(empty,50);
  table frame2(empty,50);
  table coeffs(empty,50);
  table ff(data,1,1);
  table hann(window,500,2);
  tablemap overlap(frame,frame2);
  ksig pitch, spitch, order, gain, iter, v, ki;
  asig glottal, out[2], win[2], i;
  oparray buzz[2], iirt[2], oscil[2];

  if (!itime) { // do only at initialization
      ki = 0;
      while (ki < ftlen(A_table)) {
         // Scale the vowel codebooks
         tablewrite( nA_table,ki+1,-tableread(A_table,ki)*5 );
         tablewrite( nE_table,ki+1,-tableread(E_table,ki)*5 );
         tablewrite( nU_table,ki+1,-tableread(U_table,ki)*5 );
         ki = ki + 1;
      }
   }

  // the 'lpcplus_table' table got updated since last pass:
  // undo the packaging & scaling
  unpack(lpcplus_table, v, pitch, gain, coeffs,
                        nA_table, nE_table, nU_table);

  order = 8;
  // copy it into the right local table -- we use two of
  // them in alternation to manage the overlap-add

  iter = 1-iter;
  if (iter) {
    copy_table(frame, coeffs);
  } else {
    copy_table(frame2, coeffs);
  }

  // smooth the pitch signal
  spitch = port(pitch,0.005);

  // Determine voicing
  if (v != 0) {
    // make a glottal pulse signal
    if (pitch > 0) {
      glottal = buzz(s_rate/spitch,-1,0,0.95);
    }
  }
  else { glottal = arand(1)/10; }

  // multiply it by the gain
  glottal = glottal * gain;

  // filter it with each of the two active allpole filters
  i = 0; while (i < 2) {
    out[i] = iirt[i](glottal,overlap[i],ff,order);
    i = i + 1;
  }

  // these are envelopes that handle the overlap-add
  win[0] = oscil(hann,k_rate/2)/9;
  win[1] = delay(oscil(hann,k_rate/2)/9,1/k_rate);

  // mix the outputs of the two filters
  output(win[0] * out[0] + win[1] * out[1]);
}

kopcode copy_table(table t1,table t2) {
  ksig i;

  i = 0; while (i < ftlen(t2)) {
    tablewrite(t1,i,tableread(t2,i));
    i = i + 1;
  }
}

kopcode unpack(table frame, ksig v, ksig pitch, ksig gain,
               table coeffs, table A_table, table E_table,
               table U_table) {
   // unpack the frame of data into the voicing info,
   // the pitch, the gain, and the filtershape.
   // A_table, E_table, and U_table are codebook vowels
   ksig i,val,phon;
   oparray getnextval[1];

   i = 0;
   while (i < ftlen(frame)) {
      // turn float values back to big integers
      tablewrite(frame,i,floor(tableread(frame,i)*32768+
                 32768));
      i = i + 1;
   }

   // reset the bitstream parser to the beginning of the table
   getnextval[0] (frame,-1);

   // get voicing decision
   v = getnextval[0] (frame,2);

   if (v != 0) { // voiced frame
      // get pitch (7 bits)
      pitch = getnextval[0] (frame,7);
   } else { pitch = 0; }  // unvoiced frame

   // get gain value (7-bits)
   gain = getnextval[0] (frame,7) * 2/128;

   if (v != 2) { // need a filtershape
      // get the filter coefficient values
      // (10 coefficients, 12-bits each)
      i = 0;
      while (i < 8) {
         val = getnextval[0](frame,12);
         tablewrite(coeffs,i+1,-(val * 16 / 4096 - 8) );
         i = i + 1;
      }
   } else { // use a codebook filter
      // get the codebook index (only 2 bits here)
      phon = getnextval[0] (frame,2);

      // copy the right codebook filter
      if (phon == 1) { copy_table(coeffs, A_table); }
      if (phon == 2) { copy_table(coeffs, E_table); }
      if (phon == 3) { copy_table(coeffs, U_table); }
   }
}


kopcode getnextval(table frame, ksig nbits) {
  ksig fpos, bpos, val, hi, lo, gap;

  // get nbits from the frame -- push fpos and bpos forward
  // as needed.  fpos keeps track of the current word,
  // bpos keeps track of the next bit within that word

  if (nbits < 0) { // reset
     fpos = 0; bpos = 0;
     return(0);
  }

  // shifts implemented as pow(2,x) operations
  if (bpos + nbits <= 16) {
     bpos = bpos + nbits;
     val = mod(floor(tableread(frame,fpos) /
                     pow(2,16-bpos)),pow(2,nbits));
     if (bpos == 16) { bpos = 0; fpos = fpos + 1; }
  } else {
     gap = nbits - (16-bpos);
     hi = mod(tableread(frame,fpos),pow(2,nbits-gap));
     lo = floor(tableread(frame,fpos+1) / pow(2,16-gap));
     bpos = gap;
     fpos = fpos + 1;
     val = hi * pow(2,gap) + lo;
  }

  return(val);
}

opcode mod(xsig x, xsig y) {
  return(floor(x - floor(x/y)*y + 0.5));
}
```