# Cross-Coding SDIF into MPEG-4 Structured Audio

Matthew Wright (1)
Eric D. Scheirer (2)

(1) Center for New Music and Audio Technologies, UC Berkeley, matt@cnmat.berkeley.edu
(2) Machine Listening Group, MIT Media Laboratory, eds@media.mit.edu

## Abstract

We have created a link between the Sound Description Interchange Format ("SDIF") and MPEG-4's Structured Audio ("SA") tools. We cross-code SDIF data into SA bitstreams, and write SA programs to synthesize this SDIF data. By making a link between these two powerful formats, both communities of users benefit: the SDIF community gets a fixed, standard synthesis platform that will soon be widespread, and the MPEG-4 community gets a set of powerful, robust analysis-synthesis tools. We have made the cross-coding tools available at no cost.

## 1. Introduction

The International Standards Organization completed the MPEG-4 standard, ISO/IEC 14496-3, in October 1998, and will publish and designate it as International Standard in mid-1999 [1]. One of the tools in MPEG-4 is a new sound-coding format called Structured Audio ("SA") [2]. SA allows audio to be transmitted from a server to a receiver as a set of instructions in a software-synthesis language. Upon receipt, a real-time synthesizer converts the parametric instrument definitions and sound-generation instructions into audio for playback. The synthesis language in MPEG-4 is a newly devised one called SAOL (for Structured Audio Orchestra Language, pronounced "sail") [3]. Timing and control is provided by a "score" written in the Structured Audio Score Language (SASL).

All compliant decoders of the full MPEG-4 standard—such as those that will be included in set-top boxes, Internet browser plug-ins, and portable digital players—must include an implementation of the SAOL language and be able to synthesize sound in real-time from a given SAOL program and SASL score. Because of this, we expect that considerable industry and academic resources will be devoted to building implementations of the MPEG-4 Structured Audio tools. Several such projects are already underway.

While MPEG-4 was being developed, the sound analysis and synthesis community developed and embraced the Sound Description Interface Format ("SDIF"), a general-purpose framework for representing various high-level sound descriptions such as sum-of-sinusoids, noise bands, time-domain samples, and formants [4]. Many sound-analysis systems now output to SDIF or to a format that can be converted to SDIF, and there is a growing body of tools for manipulating sound descriptions represented in SDIF.

Composers, sound designers, and analysis/synthesis researchers can benefit from the combined strengths of MPEG-4 and SDIF by using the MPEG-4 Structured Audio decoder as an SDIF synthesizer. The multimedia compression community uses the term *cross-coding* to denote the process of converting data from one format into another. Cross-coding SDIF into MPEG-4 allow musicians to use sophisticated SDIF tools to create musical works, while leveraging the broad availability anticipated of MPEG-4 playback devices.

We have created a cross-coding tool that uses SA to synthesize SDIF representations of a variety of sound descriptions. It consists of a SAOL synthesis program for each type of sound description, for example, additive synthesis for sinusoidal track data, and an SDIF-data-to-SASL-score converter. The resulting SAOL program and SASL score comprise a valid MPEG-4 bitstream that, when decoded, produces a sound appropriate to the contents of the SDIF data.

This paper provides background on SDIF and the MPEG-4 Structured Audio format, describes the operation of the cross-coder in detail, shows examples of SAOL instruments that synthesize SDIF representations, and speculates on future directions for the further connection of the two formats.

## 2. About SDIF

The developers of SDIF conceived it as an interchange format for spectral descriptions of sound, in order to enable collaboration among sound analysis-synthesis researchers. SDIF has evolved and become more general; it now includes support for non-spectral descriptions of sound such as time-domain samples and fundamental frequency estimates.

SDIF is defined in two parts: a fixed, general-purpose format framework, and an extensible collection of types of sound descriptions represented within this framework. The framework consists of three kinds of objects: *matrices*, *frames*, and *streams*. Matrices are two-dimensional arrays of primitive data elements such as integers, floating-point numbers, or text. A *matrix header* specifies the number of rows, the number of columns, and a four-byte *matrix type ID* indicating what kind of matrix it is.

A frame contains a collection of matrices. The *frame header* includes a *frame type ID*, indicating what kind of frame it is, and a *time tag*, indicating the time to which that frame applies. SDIF frames must appear in ascending order of time tag. A *stream* is a sequence of frames of the same frame type that represents a single "sonic object" evolving through time. The frame header contains a *stream ID*, and all of the frames in a stream have the same stream ID value. An SDIF file may contain one stream, or multiple streams with interleaved frames. We find that this framework is general enough to encompass the features of most representations of sound.

The second part of SDIF is a collection of standard matrix and frame types. The definition of a matrix type includes the following:

- The meaning of each column. Generally, the different columns represent different parameters of a sound object, such as frequency, amplitude, and phase. A column definition may include units, minimum and/or maximum values, and a default value.
- The meaning of each row. Generally, each row represents an individual "primitive object" such as a filter, sinusoid, or time-domain sample frame.

The definition of a frame type includes a list of the types of all the matrices required in a frame of that type, and an explanation of the semantics of those matrices in the context of the given frame type.

The full SDIF specification is available on the SDIF web site [5].

### 2.1. SDIF Extensibility

SDIF's set of frame and matrix types is extensible; SDIF users can invent new frame and matrix types to represent new types of sound descriptions. As these frame and matrix types mature they can be added to the official collection.

There are other areas where researchers can extend SDIF: they may add extra columns to standard matrix types, and extra matrices to standard frame types.

In order to be useful for interchange, each of these extensions to SDIF must be documented: the inventor of the new representation must explain how to interpret it. One way to explain the meaning of an SDIF type is to provide source code for a synthesis program that converts the SDIF data into a time-domain audio signal. We will show that SA is powerful enough to express such synthesis algorithms; because SA is standardized, nonproprietary, and expected to become widely available, it is an attractive tool for this job.

### 2.2. Time in SDIF

The time tags of SDIF frames are measured in units of seconds from an arbitrary "time zero." Typically, when an SDIF stream represents a sonic event of finite duration, its first frame will be at time zero, but negative time tags are allowed.

Time tags in an SDIF file are *descriptive*; they provide the timing information for a representation of sound that is appropriate to that representation. When SDIF is used as the result of sound analysis, SDIF time tags generally refer to the time axis of the original analyzed sound.

For most frame types, the data in the frame applies to a single instant of time. The exact behavior of a sound "between" the time points represented in an SDIF stream is outside of the representation both for analysis and synthesis. For example, on the analysis side, the choice of a technique for interpolation of phase in additive synthesis determines the model that is being fit by the analysis procedure, and can have a profound impact on the result [6; 7]. On the synthesis side, synthesizers need to interpolate between frames so that parameter values can vary smoothly, rather than allowing a discontinuity at each new frame.

Many of the sound descriptions supported by SDIF afford time-scale modification—that is, compressing and expanding the time scale to produce a new sound description based on the original but timed differently [8]. In this case, a synthesizer might move through the time axis of SDIF data in more complicated ways than simply processing one SDIF second per synthesis second [9].

## 3. About MPEG-4 Structured Audio

The MPEG-4 standard represents both audio and video, and standardizes a common framework for decoding playback of synchronized media. This standard is somewhat richer than previous MPEG standards; it supports compression of "natural" content—that is, prerecorded audiovisual material—and transmission of high-level audiovisual models to be rendered or synthesized by the MPEG-4 decoder. This so-called *synthetic/natural hybrid coding* (SNHC) enables both greater compression, and greater interactive flexibility on playback, than traditional compression techniques [10].

MPEG-4 Structured Audio (SA) is the audio-synthesis toolset in MPEG-4. It provides a powerful, standardized framework for low-bitrate, high-quality audio for interactive multimedia. MPEG-4 soundtracks may combine both compressed recorded sound and synthesized sounds through the use of a mix-down and post-production format called AudioBIFS [11].

SA is built around the music language SAOL, which stands for Structured Audio Orchestra Language [3]. SAOL comes from the Music–N and Csound traditions; its definition is fixed in the MPEG-4 standard and the technology has been released into the public domain. SAOL standardizes many of the techniques in the common practice of software-synthesis; it uses the model of an *orchestra* of *instruments* playing

*notes* from a *score*. Each instrument is a unit-generator-based computer program that implements a digital-synthesis or digital-effects-processing algorithm. Unit generators in SAOL are termed *opcodes*; there is a set of primitive opcodes fixed in the standard, and musicians may also design and transmit new opcodes written in SAOL.

Many synthesis algorithms make use of *wavetables* that contain audio-sample data. There are a number of standard generators for filling wavetables with data in SAOL. Wavetables can also be delivered dynamically in an MPEG-4 bitstream; this *dynamic wavetable* capability brings to SAOL the possibility of implementing traditional audio coders or analysis/synthesis functions. The idea of using the SA format to act as a programmable natural audio decoder, adapted in different ways to different audio signals, is termed *generalized audio coding* [12].

### 3.1. Time in SA

The timing model in SA is in part derived from its Music-N heritage and in part based on new requirements imposed on it by the streaming-media framework.

Score-based control in SA is provided through a score format called SASL (Structured Audio Score Language). SASL is a simple format that provides note instructions, control of existing notes, tempo changes, and dynamic wavetable delivery. It does not have the powerful capabilities such as looping, sections, or repeats that are found in other score formats. The only major innovation in SASL compared to the Csound score format is the ability to direct a control change to one particular note instance rather than only to a global variable.

Each event in the score is labeled with a timestamp that specifies when the event is *dispatched*, or executed. In a streaming context, the timestamp may be explicit, which allows notes to be scheduled for future dispatch, or implicit, in which notes are played as they are received by the synthesizer (similar to live MIDI control of a synthesizer). Both types of timing may be used in the same composition.

When a *note event* is dispatched, one of the instruments in the orchestra is *instantiated* to create an *instrument instance* or *note*. Each note performs the signal processing described in the orchestra and produces some output. *Control events* may be used to send commands from the score to individual notes, groups of notes, or the orchestra as a whole. When the control event is dispatched, the value of a *controller* variable in one or more notes is changed.

All score-based control is quantized to the *control period* of the orchestra. The control period specifies the "block processing" rate of the synthesizer, and is set by the author of the composition. Only one control change per controller, or one new frame of data per wavetable, may be processed in each block or *control cycle.*

Time tags in SASL scores are *imperative*. They tell the synthesis engine when it is time to perform some action. The mismatch between the descriptive time model of SDIF and the imperative model of SA is less important in practice than in theory. It only becomes a problem when an SDIF frame is tagged with a time that is later than the time that synthesis for that frame must begin. For example, the time tag for the **1STF** frame, which contains windowed and overlapped STFT data, points to the middle of the segment corresponding to the window that resulted in that spectrum estimate. If the SAOL program waits until this point in time to begin synthesis, it is too late.

We solve this problem in the cross-coder. Rather than naively mapping SDIF time directly to SASL time, we put each SDIF frame in the SASL score at the time that the synthesizer needs the data in the frame.

### 3.2. The SA streaming model

The streaming model in MPEG-4 makes sophisticated time-based manipulation of score playback difficult. For example, the "MPEG-4 radio" is a popular application target for the standard. If we imagine that the score commands are being received via a digital-radio transmission, with no ability for the receiver to talk back to the transmitter, then it is not possible to seek forward in time, since future portions of the score have not yet been received, and allowing arbitrarily long "rewind" requires lots of memory.

In the case where the receiver and transmitter do communicate, such as in Internet applications, then these sorts of time manipulations are possible. The receiver sends an interactive request back to the server, which sends out new score commands in response.

## 4. Mapping SDIF to SA

SDIF is a format for analysis/synthesis research; SA is a tool for building systems that need to use sound synthesis. In this project, we think of SDIF as the "front end" and SA as the "back end," because we start with data in SDIF format, then convert the SDIF data into an SA bitstream, then send this SA bitstream to an MPEG-4 decoder for synthesis.

Our SDIF to SA converter is called **sdif2mp4**. The **sdif2mp4** program reads and processes SDIF data one frame at a time, in a single pass, with bounded look-ahead, continuously outputting an SA bitstream. Therefore, the cross-coding part of the program can be used to read from and/or write to network streams as well as files.

The high-level task of the cross-coder is to produce an SA bitstream that decodes into a sound appropriate to the contents of the SDIF file. This is accomplished in **sdif2mp4** by mapping each stream into one note in the sound-synthesis process, and the frames of each stream into dynamic wavetables that

control the synthesis process for each note. The MPEG-4 output of our conversion consists of

- a SAOL instrument to synthesize the SDIF data
- a repackaging of the SDIF frames into a SASL score that provides the synthesis parameters for this instrument as they change over time.

### 4.1. SAOL instruments

Different representations for sound naturally suggest different analysis and synthesis methods [13]. Table 1 gives a list of the SDIF frame types we have implemented in the cross-coder and the way that they are synthesized. The cross-coder is extensible in that new SDIF frame types and their associated synthesis instruments can be added easily.

Table 1: Table of SDIF frame types and their appropriate synthesis methods.

| *Frame type* | *Type of data* | *Appropriate synthesis method* |
|---|---|---|
| 1FQ0 | Fundamental freq. estimate | Drive a wavetable oscillator at that frequency |
| 1TRC | Sinusoidal tracks | Additive synthesis |
| 1STF | STFT frames | IFFT synthesis |
| 1TDS | Time-domain samples | Sample playback |
| 1LPC | LPC coefficients | Source-filter model |

SDIF supports many types of sound descriptions that are too abstract to be synthesized unambiguously. In the analysis/synthesis context, this means that SDIF supports analysis techniques that are not fully invertible, i.e., where it is not possible to resynthesize the original analyzed sound perfectly. However, the job of a SAOL instrument is just to generate an appropriate sound, so in some cases we make arbitrary decisions in the synthesis. For example, we "synthesize" a fundamental frequency estimate by generating a synthetic waveform of strong and unambiguous pitch with the given frequency envelope.

The SAOL orchestra **sdif.saol** contains the synthesizers for all of these data types. New synthesizers can be added to this as needed. If the file becomes too large, it would be easy to have the cross-coder only include synthesizers that are needed for a particular SDIF file, but this hasn't been done yet.

### 4.2. A simple example

Program Box 1 contains a simple example of a SAOL instrument that can receive commands from a SASL score and drive synthesis accordingly, and a score to control it. Such a score could easily be derived from an SDIF **1FQ0** by inspecting the contents of the data matrices and converting the pitch estimates to controllers. This example demonstrates how SAOL and SASL interact, but does not show the actual manner of operation of the cross-coder.

```
instr onepitch() {
  imports exports ksig newfreq, newamp;
  ksig freq, smoothfreq, amp;
  table wave(1024,1,0,0.9,0,0.5,0.4,0.4);
  asig sound;

  if (itime) { // init on first pass
    freq = 1;
    amp = 0;
  }

  if (newfreq) { // receive new data
    freq = newfreq;
    amp = newamp;
    newfreq = 0; // clear the new data
  }

  // smooth the frequency
  smoothfreq = port(freq,0.01);

  // audio-rate: just run an oscillator
  sound = oscil(wave, smoothfreq) * amp;
  output(sound);
}
```

---

```
note1: 0.0  onepitch –1
       0.15 control note1 newfreq 440
       0.15 control note1 newamp 0.9
       0.23 control note1 newfreq 442.2
       0.23 control note1 newamp 0.93
       0.25 control note1 newfreq 438.7
       0.25 control note1 newamp 0.94
       0.51 control note1 newfreq 385
       0.51 control note1 neamp 0.81
   ...
```

Program Box 1: Orchestra (top) and score (bottom) demonstrating score-based control. The first line in the score is an *instrument line*, which generates a note, labelled **note1**, with the instrument **onepitch**. The rest are control lines, which send new values to the controllers **newfreq** and **newamp** in the note labeled **note1**. The instrument **onepitch** watches for new values on the **newfreq** controller and updates the synthesis parameters accordingly. Synthesis is done through oscillating over a waveshape (called **wave**).

### 4.3. Streaming data

In the actual cross-coder, all of the SDIF frame data are included as dynamic wavetables in the MP4 bitstream. The challenges in this process are, first, how to associate each SDIF stream with its appropriate synthesis method; and second, how to associate each wavetable with the right note.

As described in Section 3, each note is associated with an *instrument* of the orchestra. Whenever a new stream begins in an SDIF file, we put an instrument event into the score to instantiate the appropriate instrument from the SAOL orchestra. This event is conveyed at the time of the first frame of the stream. As the SA bitstream is decoded, a note is created at this time.

The next step in cross-coding is to repackage the frames of SDIF data in the SA bitstream and to make sure they are received by the right instrument. Conceptually, this is as simple as sending each frame to the running note instance that corresponds to the appropriate stream. However, only scalar variables may be used as controllers in SAOL. All dynamic wavetables must live in the global namespace. Thus, rather than sending the frame to the instrument, we send the frame as a global wavetable, and send a message to the instrument telling it to look at the wavetable. Each instrument has a special controller named **changed** that is used for this purpose.

Another detail is that several streams may each have a frame at the same time. Thus, it is necessary to have several global wavetables available in which the SDIF frames may be placed. If there is only one frame at a particular time, then only one of these tables is used; if multiple frames appear at one time, then several tables are used. The **changed** controller for the appropriate note is used to indicate which of the tables contains the new data for that note.

This number of global wavetables available for receipt of the data limits the number of simultaneous frames. It would be easy to overcome this limit by having the SAOL code dynamically generated in whole or in part as one segment of the cross-coding process. That is, the number of available global wavetables, which is static in the SAOL code, could be made to vary for each SDIF file by updating the SAOL code itself depending on the contents of the SDIF file.

The control rate of the SAOL orchestra must be set high enough that successive frames of SDIF data are maintained as part of separate control cycles. This is because each wavetable can only be changed in the score at most one time during a control cycle. The frame rate is set in the header of the SAOL code. Again, the cross-coder could set this automatically by editing the SAOL code before it is included in the bitstream.

## 5. How SAOL Instruments Access SDIF Frames

Each SDIF-synthesizing SAOL instrument must get frame data from the wavetable-packed format generated by the encoder. We have provided helper procedures for this task to make it easier for other developers to write SDIF synthesizers in SAOL.

A single note in the synthesis process synthesizes an entire SDIF stream. The score communicates with this note to tell it when a new frame is available for the stream to which it corresponds. SA's model of score/instrument interaction is not conceptually a message-passing metaphor, but one in which the score has "write access" to the variable data space. Each wavetable and controller appears as a variable that is accessible to the SAOL program.

We have written several helper functions in SAOL to manage SDIF frames represented as SAOL wavetables. By doing this, we have provided an abstract data type that corresponds to the SDIF matrix, so that instruments can use matrix data without being aware of the details of matrix representation and delivery. The primary methods for using this data type are:

- **getmatrix()**, which looks to see if a new frame of data has a matrix of a certain type, and if so copies it into a local "matrix variable."
- **numrows()** and **numcols()**, which return the number of rows and columns in a matrix.
- **matread()**, which reads a value from a particular row and column of a matrix.

Because dynamic wavetables are always global, as discussed in the preceding section, it is the responsibility of each user-defined instrument to inspect the value of its **changed** controller to see if any new matrices have arrived for it. This can be accomplished by code similar to that shown in Program Box 2.

The **changed** variable is marked in the instrument as a controller by use of the **imports** and **exports** tags.

```
instr my_instrument() {
  imports table sdif_table1;
  imports table sdif_table2;
  imports table sdif_table3;
  tablemap tab(sdif_table1,
               sdif_table2,
               sdif_table3);
  table new(empty,1000);
  imports exports ksig changed;

  if (changed) {
    getmatrix(tab[changed], 1, new);
    changed = 0;

    // update synthesis parameters
    ...
  }
 }
```

Program Box 2: SAOL instrument section demonstrating how to use the **changed** controller and **getmatrix** opcode to load new matrices when they arrive in the score.

This allows the score to update its values as discussed in the previous section. Whenever the **changed** controller takes on a non-zero value, the value shows which of several wavetables contains the new data for the instrument. The instrument makes a local copy of the matrix called **mydata**, unpacks the data from it, and uses the new data to update its synthesis parameters accordingly.

The syntax **tab[changed]** is a *tablemap expression*—it references the corresponding entry in the tablemap list given in the definition of **tab**. The table actually changed by the score is one of the three imported tables; **tab[]** allows them to be addressed indirectly.

```
instr track() {
  imports table sdif_table1;
  imports table sdif_table2;
  imports table sdif_table3;
  imports table sdif_table4;
  tablemap tab(sdif_table1, sdif_table2,
               sdif_table3, sdif_table4);
  table mydata(empty,1000);
  table pure(harm,1024,1);
  imports exports ksig changed;
  ksig freq, smfreq, max, f[1024],
       amp[1024], ind[1024];
  asig i, sum;
  oparray oscil[1024];

  // control-rate
  if (changed) {
    getmatrix(tab[changed],1,mydata);
    changed = 0;
  }
  maketracks(f,amp,ind,mydata,max);

  // audio-rate
  i=0; sum = 0; while (i < max) {
    sum = sum +
        oscil[i](pure,f[i]) * amp[i];
    i = i + 1;
  }

  output(sum);
}

kopcode maketracks(ksig freq[1024],
  ksig amp[1024], ksig ind[1024],
  table mat, ksig max) {
  ksig i, nr, ix, a, f, k;

  nr = numrows(mat);

  i = 0; while (i < nr) {
    ix = matread(mat,i,0);
    f = matread(mat,i,1);
    a = matread(mat,i,2);

    k = get_ind(ix,ind,max);
    if (k > max) { max = k; }
    freq[k] = f; amp[k] = a;
    ind[k] = ix;
    i = i + 1;
  }
}
```

Program Box 3: SAOL code implementing the **track** instrument, which performs additive synthesis for the **1TRC** frame type.

## 6. Example SAOL Instruments

This section describes the SAOL instruments for three SDIF frame types. From these examples, it will be clear how designers of new custom frame types could easily write SAOL synthesizers for them.

### 6.1. Sinusoidal Tracks

The synthesizer for the **1TRC** frame type, **track()**, is shown in Program Box 3.

The user-defined opcode **maketracks()** is used to parse the data in the data matrix, now stored in the wavetable **mydata**, into frequency and amplitude values for the set of sinusoids, stored in the arrays **freq[]** and **amp[]**. (This synthesizer ignores phase values specified in SDIF data.)

```
instr f0_syn() {
  imports table sdif_table1;
  imports table sdif_table2;
  imports table sdif_table3;
  tablemap tab(sdif_table1,sdif_table2,
               sdif_table3);
  table mydata(empty,1000);
  table wave(harm,512,1,0,1,0.2,0,0.4);
  imports exports ksig changed;
  ksig freq, smoothfreq;

  if (changed) {
     getmatrix(tab[changed], 1, mydata);
     changed = 0;
  }

  freq = find_best_pitch(mydata);
  smoothfreq = port(freq, 0.01);
  output(oscil(wave, smoothfreq));
}
```

Program Box 4: The **f0_syn()** instrument, which synthesizes the **1FQ0** frame type.

As defined by SDIF's standard **1TRC** frame type, each partial tone has an "index number" that is used to monitor births and deaths of partials over time. The **get_ind()** user-defined opcode (not shown) keeps track of the currently-known index numbers and matches them up with the incoming values.

The **track()** instrument code is the main synthesis code and the only part of the instrument that runs at the audio rate; it uses the **freq[]** and **amp[]** values to drive a set of sinusoidal oscillators. The oscillators read from the wavetable **pure**, which contains a sine wave by virtue of the specifier **harm** in its declaration.

A SAOL *oparray* is a bank of oscillators; the expression **oscil[i](...)** invokes the **i**th oscillator in the bank. If the array index was not used, i.e., **oscil(...)**, each iteration of the **while (i < max)** loop would refer to the *same* oscillator, giving incorrect results.

It is readily clear from the code that for this frame type, the code used for bookkeeping the frame data—just reading and parsing the frequencies and amplitudes—greatly outweighs the code for the actual synthesis, which is only five lines long. This is the typical case, as can be seen by examining the full source code of **sdif.saol** available on the SDIF website [5]. Further, all of the details of interpolation, block management, input/output processing, and sound generation are handled by the MPEG-4 decoder according to the rules of the standard. The musician or sound designer (who naturally wishes to focus on the details of the synthesis algorithm) is not concerned with these aspects of the synthesis process.

Because it is so easy to write synthesizers of various sorts in SAOL, using the cross-coder is a straightforward way to rapidly prototype different synthesis

methods for SDIF frame types. Synthesis of many frame types can be accomplished in only a half-dozen lines of SAOL code.

### 6.2. Fundamental Frequency Estimates

The SAOL instrument for **1FQ0** is called **f0_syn()** and is shown in Program Box 4. The SDIF frame for **1FQ0** contains (frequency, likelihood) pairs, specifying the degree of confidence in each of several pitch estimates. To synthesize from this data representation, **f0_syn()** simply generates a synthetic tone at the frequency corresponding to the largest likelihood (calculated by **find_best_pitch()**, not shown). The built-in **port()** function of SAOL is used to interpolate between the asynchronously-arriving pitch estimates.

### 6.3. Short-Time Fourier Transform

SDIF's **1STF** frame type represents sound as DFT frames. The SAOL synthesis instrument uses overlap-add inverse FFTs, as shown in Program Box 5.

In SDIF, the frame spacing can be arbitrary, and the overlap between successive frames can change over time in an implicit way (that is, the overlap-add is asynchronous). The SAOL **ifft()** built-in opcode, which is otherwise very useful for doing this sort of synthesis, requires that the frame length, IFFT size, and overlap length all be known at startup and not vary thereafter. Therefore the instrument shown in Program Box 5 works on the restricted set of **1STF** streams in which the frames appear at regular intervals.

Given this restriction, the synthesis part of **stft()** is only one line long. The rest of the instrument manages the frame data by separate tables containing the real and imaginary parts of each spectral component.

It would be possible to remove this restriction. One solution would be for the cross-coder to re-code and interpolate frames to make them regular. Another solution would be to write a user-defined IFFT function that does not suffer from the limitations of the built-in **ifft().**

```
instr stft(size, len) {
  imports table sdif_table1;
  imports table sdif_table2;
  imports table sdif_table3;
  imports table sdif_table4;
  tablemap tab(sdif_table1,sdif_table2,
               sdif_table3, sdif_table4);
  imports ksig changed;
  table info(empty,10);
  table data(empty,10000);
  table re(empty, 2048);
  table im(empty, 2048);
  table win(window,len*s_rate,2); // hamming
  table invwin(empty,len*s_rate);
  ksig i; ivar fftlen;

  // i-time
  make_inv_win(win,invwin);
  len = len * s_rate;
  idump(len,size);

  // k-time
  if (changed) {
    getmatrix(tab[changed],1,info);
    getmatrix(tab[changed],2,data);
    // split data into RE and IM parts
    i = 0; while (i < size) {
      tablewrite(re,i,matread(data,i,0));
      tablewrite(im,i,matread(data,i,1));
      i = i + 1;
    }
    changed = 0;
  }

  // a-time
  // IFFT parameters: real, im, output
  //  length, window shift, IFFT size,
  //  synthesis window

  output(ifft(re,im,len,
              len/4,size,invwin));
}
```

Program Box 5: The **stft** instrument, which synthesizes the **1STF** frame type.

## 7. Efficiency

Naturally, different SAOL implementations will execute the cross-coded MP4 bitstream at different speeds. The **saolc** reference software from the MIT Media Lab [14] is very inefficient and will not play sound in real-time. Other implementations available, such as the **sfront** MP4-to-C translator in development at the University of California at Berkeley CS Division, are much more efficient. As more advanced software tools and integrated hardware-software systems, for MPEG-4 processing come onto the market, very efficient SAOL-based synthesis will be possible. The technology required for creating optimizing SAOL compilers is available today.

The "conformance" part of the MPEG-4 specification will be finalized over the next few months, and will contain a SAOL simulation tool that measures the approximate number of calculations per sample required to synthesize a particular Structured Audio bitstream in real-time. Decoder manufacturers will rate the performance of their systems in terms of this tool, thus providing guarantees to musicians that bitstreams of a certain complexity will play back in real-time on a particular brand of system. The musician can use the same tool to understand the complexity of his/her sounds, and so know exactly which multimedia terminals have the necessary horsepower to synthesize them (and reduce their complexity if necessary).

In writing the cross-coder, we haven't paid any particular attention to the size of the cross-coded representations. This project wasn't conceived as a study of sound compression, although there are natural similarities with other recent work in sinusoidal-model compression of sound [15].

## 8. Deeper Integration of SDIF and SAOL

A previous paper [16] discussed the ways in which bringing wavetable synthesis, based on the MIDI DLS-2 format, into the MPEG-4 standard enables an effective merger of algorithmic (rule-based) and wavetable-sampling synthesis techniques. Similarly, now that easy cross-coding of SDIF data into MPEG-4 is possible, it is natural to consider the musical possibilities created by using direct SAOL sound-description in tandem with sounds cross-coded from SDIF.

SAOL's standard *bus routing* capabilities can be used for audio effects processing of synthesized SDIF data, for example, reverberation. These sorts of effects are easily realized in a joint SDIF-SAOL composition by using SAOL to post-process the SDIF-synthesized data.

Other possibilities for creating joint SDIF-SAOL compositions include writing some instruments in SAOL (such as FM synthesizers) and mixing the results of those instruments with SDIF synthesis. Also, rule-based manipulation of the SDIF playback can be achieved in SAOL; for example, in **1TRC** synthesis, to modify or eliminate certain tracks based on other things going on in the music, or based on user interaction.

These explorations remain topics for future research.

## 9. Acknowledgements

Adrian Freed, John Lazzaro, Xavier Rodet, Diemo Schwarz, David Wessel.

## REFERENCES

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

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

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

[4] Wright, M., Chaudhary, A., Freed, A, Wessel, D., Rodet, X., Virolle, D., Woehrmann, R., and Serra, X. (1998). New applications of the Sound Description Interchange Format. In *Proceedings of the 1998 Int. Computer Music Conf.* (pp. 276-279). Ann Arbor, MI: International Computer Music Association.

[5] SDIF Web Site: http://www.cnmat.berkeley.edu/SDIF

[6] Quatieri, T. F. & McAulay, R. J. (1998). Audio signal processing based on sinusoidal analysis/synthesis. In M. Kahrs & K. Brandenburg (eds.), *Applications of Digital Signal Processing to Audio and Acoustics* (pp. 343-411). New York: Kluwer Academic.

[7] Xiaoshu, Q. & Yinong, D. (1997). A phase interpolation algorithm for sinusoidal model based music synthesis. In *Proceedings of the 1997 No.97CB36052) 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing* (pp. 451-4). Munich, Germany: IEEE Comput. Soc. Press.

[8] Laroche, J. (1998). Time and pitch scale modification of audio signals. In M. Kahrs & K. Brandenburg (eds.), *Applications of Signal Processing to Audio and Acoustics* (pp. 279-310). New York: Kluwer Academic.

[9] Wessel, D., Wright, M. & Khan, S. A. (1998). Preparation for Improvised Performance in Collaboration with a Khyal Singer. In *Proceedings of the 1998 International Computer Music Conference.* Ann Arbor, Michigan: International Computer Music Association.

[10] Doenges, P. K., Capin, T. K., Lavagetto, F. *et al* (199). MPEG-4: Audio/video and synthetic graphics/audio for mixed media. *Signal Processing - Image Communication* **9**(4), 433-463.

[11] Scheirer, E. D., Väänänen, R. & Huopaniemi, J. (in press). AudioBIFS: Describing audio scenes in the MPEG-4 multimedia standard. *IEEE Transactions on Multimedia*.

[12] Scheirer, E. D. & Kim, Y. E. (1999). Generalized audio coding with MPEG-4 Structured Audio. In *Proceedings of the 1999 AES 17th International Convention (High-Quality Audio Coding)*. Florence, IT.

[13] 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), 922-940.

[14] Scheirer, E. D. (1999). External documentatation and release notes for **saolc**. MIT Media Laboratory Machine Listening Group Technical Report, Cambridge, MA. Available from

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

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