Kelly Fitz,[*] Lippold Haken,[†] Susanne Lefvert,[††] Corbin Champion,[§] and Mike O'Donnell[¶]
[*]Department of Electrical Engineering and Computer Science
Washington State University
kfitz@eecs.wsu.edu
[†]CERL Sound Group
University of Illinois at Urbana-Champaign
lippold@cerlsoundgroup.org
[††]Lulea University of Technology and Department of Computer Science,
University of Chicago
slefvert@hotmail.com
[§]Department of Electrical Engineering and Computer Science
Washington State University
corbinchampion@hotmail.com
[¶]Department of Computer Science
University of Chicago
odonnell@cs.uchicago.edu

Cell-Utes and Flutter-Tongued Cats: Sound Morphing Using Loris and the Reassigned Bandwidth-Enhanced Model

The reassigned bandwidth-enhanced additive sound model is a high-fidelity sound representation that allows manipulations and transformations to be applied to a great variety of sounds, including noisy and inharmonic sounds. Combining sinusoidal and noise energy in a homogeneous representation, the reassigned bandwidth-enhanced model is ideally suited to sound morphing and is implemented in the open-source software library Loris. This article presents methods for using Loris and the reassigned bandwidth-enhanced additive model to achieve high-fidelity sound representations and manipulations, and it introduces software tools that allow programmers (in C/C++ and various scripting languages) and non-programmers to use the sound modeling and manipulation capabilities of the Loris package.

The reassigned bandwidth-enhanced additive model is similar in spirit to traditional sinusoidal models (McAulay and Quatieri 1986; Serra and Smith 1990; Fitz and Haken 1996) in that a waveform is modeled as a collection of components, called *partials*, having time-varying amplitude and frequency envelopes. Our partials are not strictly sinusoidal, however. We employ a technique of bandwidth enhancement to combine sinusoidal energy and noise energy into a single partial having time-varying frequency, amplitude, and noisiness (or bandwidth) parameters (Fitz, Haken, and Christensen 2000a). The bandwidth envelope allows us to define a single component type that can be used to manipulate both sinusoidal and noisy parts of sound in an intuitive way. The encoding of noise associated with a bandwidth-enhanced partial is robust under time dilation and other model-domain transformations, and it is independent of other partials in the representation.

We use the method of reassignment (Auger and Flandrin 1995) to improve the time and frequency estimates used to define our partial parameter envelopes. The breakpoints for the partial parameter envelopes are obtained by following ridges on a reassigned time-frequency surface. Our algorithm shares with traditional sinusoidal methods the notion of temporally connected partial parameter estimates, but by contrast, our estimates are non-uniformly distributed in both time and frequency. This model yields greater resolution in time and frequency than is possible using conventional additive techniques and preserves the temporal envelope of transient signals, even in modified reconstruction (Fitz, Haken, and Christensen 2000b).

Computer Music Journal, 27:3, pp. 44–65, Fall 2003

The combination of time-frequency reassignment and bandwidth enhancement yields a homogeneous model (i.e., a model having a single component type) that is capable of representing at high fidelity a wide variety of sounds, including inharmonic, polyphonic, impulsive, and noisy sounds. The homogeneity and robustness of the reassigned bandwidth-enhanced model make it particularly well-suited for such manipulations as cross synthesis and sound morphing.

Reassigned bandwidth-enhanced modeling and rendering and many kinds of manipulations, including sound morphing, have been implemented in an open-source software package called Loris. However, Loris offers only programmatic access to this functionality and is difficult for non-programmers to use. We begin this article with an introduction to the selection of analysis parameters to obtain high-fidelity, flexible representations using Loris, and we continue with a discussion of the sound morphing algorithm used in Loris. Finally, we present three new software tools that allow composers, sound designers, and non-programmers to take advantage of the sound modeling, manipulation, and morphing capabilities of Loris.

Reassigned Bandwidth-Enhanced Analysis Parameters

We have designed the reassigned bandwidth-enhanced analyzer in Loris to have parameters that are few and orthogonal. That is, we have minimized the number of parameters required and also minimized the interaction between parameters, so that changes in one parameter would not necessitate changes in other parameters. Moreover, we have made our parameters hierarchical, so that in most cases, a good representation can be obtained by adjusting only one or two parameters, and only rarely is it necessary to adjust more than three. Consequently, and in contrast to many other additive analyzers, the parameter space of the reassigned bandwidth-enhanced analyzer in Loris is smooth and monotonic, and it is easy to converge quickly on an optimal parameter set for a given sound.

The reassigned bandwidth-enhanced analyzer can be configured according to two parameters: the instantaneous frequency resolution (or minimum instantaneous frequency separation between partials) and the shape of the short-time analysis window, specified by the symmetrical main lobe width in Hz.

The frequency resolution parameter controls the frequency density of partials in the model data. Two partials will, at any instant, differ in frequency by no less than the specified frequency resolution. The frequency resolution should be slightly less than the anticipated difference in frequency between any two adjacent partials. For quasi-harmonic sounds (sounds having energy concentrated very near integer multiples of a fundamental frequency), the anticipated frequency difference between adjacent partials is equal to the harmonic spacing, or the fundamental frequency, and the frequency resolution is typically set to 70–85% of the fundamental frequency. For inharmonic sounds, some experimentation may be necessary, and intuition can often be obtained using a spectrogram tool.

The shape of the short-time analysis window governs the time-frequency resolution of the reassigned spectral surface, from which bandwidth-enhanced partials are derived. An analysis window that is short in time, and therefore wide in frequency, yields improved temporal resolution at the expense of frequency resolution. Spectral components that are near in frequency are difficult to resolve, and low-frequency components are poorly represented, having too few periods in each window to yield stable and reliable estimates of frequency and amplitude. A longer analysis window compromises temporal resolution but yields greater frequency resolution. Spectral components that are near in frequency are more easily resolved, and low-frequency components are more accurately represented; however, short-duration events may suffer temporal smearing, and short-duration events that are near in time may not be resolved. (See, for example, Masri, Bateman, and Canagarajah 1997 for a discussion of issues surrounding window selection in short-time spectral analysis.)

The use of time-frequency reassignment improves the time and frequency resolution of the reassigned bandwidth-enhanced model relative to traditional short-time analysis methods (Fitz, Haken, and Christensen 2000b). Specifically, it allows us to use long (narrow in frequency) analysis windows to obtain good frequency resolution without smearing short-duration events. However, multiple short-duration events occurring within a single analysis window still cannot be resolved. Fortunately, the improved frequency resolution from time-frequency reassignment also allows us to use short-duration analysis windows to analyze sounds having a high density of transient events without greatly sacrificing frequency resolution.

The choice of analysis window width depends on the anticipated partial frequency density. The window width is the width of the main lobe of the Fourier transform of the Kaiser analysis window, measured between zeros in the magnitude spectrum. Generally, the window width is set equal to the anticipated minimum instantaneous frequency difference between any two partials, or the fundamental frequency in the case of quasi-harmonic sounds. For quasi-harmonic sounds, it is rarely necessary to use windows wider than 500 Hz, although good results have been obtained using windows as wide as 800 Hz to analyze a fast bongo roll. Similarly, for very low frequency quasi-harmonic sounds, best results are often obtained using windows as wide as 120 Hz.

All other parameters of the Loris analyzer can be configured automatically from the specification of the frequency resolution and analysis window width parameters, but they are also independently accessible and configurable.

The frequency drift parameter governs the amount by which the frequency of a partial can change between two consecutive data points extracted from the reassigned spectral surface. This parameter is generally set equal to the frequency resolution, but in some cases, for example in quasi-harmonic sounds having strong noise content, the frequency of some low-energy partials may tend to occasionally "wander" away from the harmonic frequency, resulting in poor harmonic tracking. In these cases, reducing the frequency drift to, say, 0.2 times (one-fifth) the fundamental frequency may greatly improve harmonic partial tracking, which is important for manipulations such as morphing.

The hop time parameter specifies the time difference between successive short-time analysis window centers used to construct the reassigned spectral surface. Data are generally obtained from each analysis window for all partials active at the time corresponding to the center of that window, so the hop time controls, to some degree, the temporal density of the analysis data (though, thanks to the use of time-frequency reassignment, it controls the temporal resolution of the data to a much lesser degree). The hop time used by the reassigned bandwidth-enhanced analyzer in Loris is normally derived from the analysis window width according to a heuristic for short-time Fourier analysis described by Allen and Rabiner (1977). In many cases, it is possible to increase the hop time, thereby reducing the volume of data, by a factor of two without compromising the quality of the representation. In other cases, it may be desirable to decrease the hop size, although we have never encountered such a situation.

In some instances, it is convenient to set the minimum instantaneous partial frequency independently of the frequency resolution. This can be accomplished by setting the frequency floor parameter, which is otherwise (by default) set equal to the frequency resolution. Similarly, it is occasionally useful to raise the amplitude floor parameter from its default value of –90 dB. This parameter represents an amplitude threshold, relative to a full-amplitude sinusoid, below which reassigned spectral components are considered insignificant and are not used to form partials.

Figure 1 demonstrates the use of the Loris procedural interface (in the C language) to perform a reassigned bandwidth-enhanced analysis of a clarinet tone having a fundamental frequency of approximately 415 Hz. The reassigned bandwidth-enhanced partials obtained from the analysis are exported to a Sound Description Interchange Format (SDIF) data file (Wright et al. 1999).

Sound Morphing in Loris

We distinguish sound morphing from other kinds of sound transformations that are sometimes

Figure 1. C code using the Loris procedural interface to perform a reassigned bandwidth-enhanced analysis of a clarinet tone that has a fundamental frequency of approximately 415 Hz.

```
/*
 * analysis_example.c
 */
#include <stdio.h>
#include <loris.h>

#define FUNDAMENTAL 415.0

int main( )
{
  Analyzer * anal = NULL;
  PartialList * partials = NULL;
  SampleVector * vec = NULL;
  double sr;
  const char * infilename = "clarinet.aiff";

  /* allocate a SampleVector */
  vec = createSampleVector(0);

  /* store the sample rate in sr, ignore the number of channels */
  importAiff(infilename, vec, &sr, NULL);

  /* create and configure an Analyzer */
  anal = createAnalyzer( .8 * FUNDAMENTAL, FUNDAMENTAL );
  analyzer_setFreqDrift( anal, .2 * FUNDAMENTAL );

  /* analyze and store partials */
  partials = createPartialList();
  analyzer_analyze( anal, vec, sr, partials );

  /* export to SDIF file */
  exportSdif("clarinet.sdif", partials);

  /* cleanup */
  destroyAnalyzer(anal);
  destroyPartialList(partials);
  destroySampleVector(vec);

  return 0;
}
```

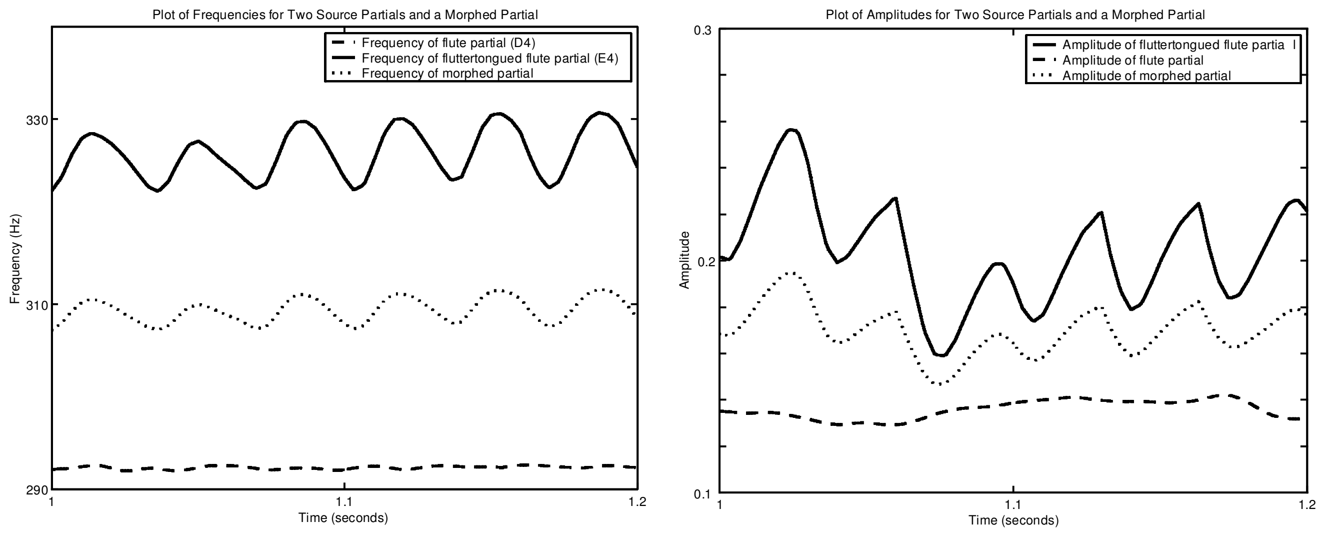


Figure 2. Equal-weight morphing of a hypothetical pair of partials by interpolation of their frequency (left plot) and amplitude (right plot) envelopes. The source partial envelopes, from a flute tone, pitch D4, and a flutter-tongued flute tone, pitch E4, are plotted with solid and dashed lines, and the envelopes corresponding to a 50% (equal weight) morph are plotted with dotted lines.

loosely characterized as "morphing," such as cross synthesis. In sound morphing, the original source timbres are obtained at the extrema of the morphing envelopes. That is, one source sound is produced when the envelopes have value 0, the other source is produced when the envelopes have value 1, and hybrid sounds are obtained from other envelope values.

Sound morphing using traditional additive sound models is conceptually straightforward. For quasi-harmonic sounds, in which each harmonic is represented by a single sinusoidal partial, the time-varying frequencies and amplitudes of the quasi-harmonic partials in the morphed sound can be obtained by a weighted interpolation of the time-varying frequencies and amplitudes of corresponding partials in the source sounds (Dodge and Jerse 1997). This process is illustrated in Figure 2.

In Loris, although the process of partial construction is different, the morphing process is fundamentally similar. Sound morphing is achieved by interpolating the time-varying frequencies, amplitudes, and bandwidths of corresponding partials obtained from reassigned bandwidth-enhanced analysis of the source sounds. Three independent morphing envelopes control the evolution of the frequency, amplitude, bandwidth, and noisiness of the morph.

The description of sounds as "quasi-harmonic" implies a natural correspondence between partials with the same harmonic number. Even very noisy quasi-harmonic sounds, which, in traditional sinusoidal models, are represented by many short, jittery partials in noisy spectral regions, can be represented by a single partial for each harmonic using the reassigned bandwidth-enhanced additive model (Haken, Fitz, and Christensen forthcoming). This property of the model greatly simplifies the morphing process in Loris and improves the fidelity of morphs for such sounds.

For inharmonic or polyphonic sounds, however, there may be no obvious correspondence between partials in the source sounds, or there may be many possible correspondences. Loris provides mechanisms for explicitly establishing correspondences between source partials.

Our sound-morphing technique bears a superficial resemblance to object or character morphing as practiced in the computer graphics community. The process of transforming one three-dimensional shape into another may be divided into two complementary problems, the *correspondence problem* and the *interpolation problem*. As with the sound-morphing technique we describe, the latter is relatively simpler (Kent, Carlson, and Parent 1992). Lazarus and Verroust (1998) present a survey of morphing techniques for geometric models, and Wolberg (1998) presents an extensive discussion of image-based morphing techniques. While goals and terminology are shared between the two fields, the problem domains are sufficiently different to have precluded the development of a unified theory of morphing.

Establishing Partial Correspondences

Correspondences between partials in the source sounds are established by *channelizing* and *distilling* partial data for the individual source sounds. Partials in each source sound are assigned unique identifiers, or labels, and partials having the same label are morphed by interpolating their frequency, amplitude, and bandwidth envelopes according to the corresponding morphing function. The product of a morph is a new set of partials, consisting of a single partial for each label represented in any of the source sounds.

In Loris, channelizing is an automated process of labeling the partials in an analyzed sound. Partials can be labeled one by one, but analysis data for a single sound may consist of hundreds or thousands of partials. If the sound has a known, simple frequency structure, an automated process is much more efficient.

Channelized partials are most often labeled according to their adherence to a harmonic frequency structure with a time-varying fundamental frequency. The frequency spectrum is partitioned into non-overlapping channels having time-varying center frequencies that are harmonic (i.e., integer) multiples of a specified reference frequency envelope, and each channel is identified by a unique label equal to its harmonic number. Each partial is assigned the label corresponding to the channel containing the greatest portion of the partial's energy.

The reference (fundamental) frequency envelope for "channelization" can often be constructed automatically by tracking a long, high-energy partial in the analysis data. The reference envelope can also be constructed point by point using data obtained from some other fundamental frequency evaluation algorithm and imported as, for example, SDIF data (Wright et al. 1999). Pitch estimation algorithms for speech signals are discussed extensively in Hess (1983), and a recent comparative study of such techniques is found in de Cheveign and Kawahara (2001).

The sound-morphing algorithm described above requires that partials in a given source be labeled uniquely; that is, no two partials can have the same label. In Loris, *distillation* is the process for enforcing this condition. All partials identified with a particular channel, and therefore having a common label, are distilled into a single partial, leaving at most a single partial per frequency channel and label. Channels that contain no partials are not represented in the distilled partial data. Partials that are not labeled (that is, partials having label 0) are unaffected by the distillation process. All unlabeled partials remain unlabeled and unmodified in the distilled partial set.

Note that, owing to the symmetry of the frequency channels employed by the Loris channelizer, a frequency region below half the reference (fundamental) channel frequency is not covered by any channel, and therefore partials concentrated at frequencies far below the reference frequency envelope will remain unlabeled after channelizing. In practice, few partials, if any, are found in this region.

Labeled and distilled sets of partials are morphed by interpolating the envelopes of corresponding partials according to specified morphing functions. Partials in one distilled source that have no corresponding partial in the other source are cross-faded according to the morphing function (Tellman, Haken, and Holloway 1995). Source partials may also be unlabeled, or assigned the label 0, to indicate that they have no correspondence with other sources in the morph. All unlabeled partials in a morph are cross-faded according to the morphing function.

When there is no temporal overlap of partials in a frequency channel, then distillation is simply a process of linking partials end to end and inserting silence between the endpoints. When partials in a frequency channel overlap temporally, then an algorithm is needed to determine a single frequency, amplitude, and noisiness value for the distilled partial at times in the overlap region. In Loris, the distiller resolves overlap issues by choosing the strongest of the overlapping partials (i.e., those with the most spectral energy) to construct the distilled partial. The energy in the rejected partials is not lost; rather, it is added to the distilled partial as noise energy in a process called *energy redistribution* (Fitz, Haken, and Christensen 2000a).

In some cases, the energy redistribution effected by the distiller is undesirable. In such cases, the partials can be *sifted* before distillation. The *sifting* process in Loris identifies all the partials that would be rejected (and converted to noise energy) by the distiller and assigns them a label of 0. These sifted partials can then be identified and treated separately or removed altogether, or they can be passed through the distiller unlabeled, and cross-faded in the morphing process.

The various morph sources need not be distilled using identical sets of frequency channels. However, dramatic partial frequency sweeps will dominate other audible effects of the morph, so care must be taken to coordinate the frequency channels used in the distillation process. Though the harmonic frequency structure described by the channelization process may not be a good representation of the frequency structure of a particular sound (as in the case of a inharmonic bell sound for example), it may still yield good morphing results by labeling partials in such a way as to prevent dramatic frequency sweeps.

Temporal Feature Alignment

Significant temporal features of the source sounds must be synchronized to achieve good morphing results. Many sounds—particularly, many familiar monophonic, quasi-harmonic sounds—share a temporal structure that includes such features as attack, sustain, and release. Additionally, many such sounds have recurring temporal features such as vibrato and tremolo cycles. A morph of such sounds may be unsatisfying if the sources have very different temporal feature sets or different numbers of temporal features, or if related temporal features (such as the end of the attack, or the beginning of the release) occur at different times. Moreover, when synchronizing a sound morph with a visual sequence, such as a computer animation, temporal features of the morphed sound must be aligned with visual events in the animation to make the relationship between sound and image believable or seemingly "natural" (Bargar et al. 2000).

Loris provides a *dilation* mechanism for non-uniformly expanding and contracting the partial parameter envelopes to redistribute temporal events. For example, when morphing instrument tones, it is common to align the attack, sustain, and release portions of the source sounds by dilating or contracting those temporal regions.

The process of resolving conflicts between different numbers of temporal features, such as different numbers of vibrato cycles, is beyond the scope of this article but has been addressed in Tellman, Haken, and Holloway (1995). This process of time dilation can occur before or after distillation, but is an essential component in controlling the evolution of the morph.

Other Deformations

Deformation and temporal dilation of the partial parameter envelopes can be applied as needed at any point in the morphing process. The reassigned bandwidth-enhanced additive model is highly robust under such transformations (Fitz, Haken, and Christensen 2000a). Because the product of the morphing process in Loris is a set of partials like any other, it can be further deformed or manipulated in any of its parameters (time, frequency, amplitude, and noisiness) or morphed with yet another source sound to achieve so-called "N-way" morphs.

For example, quasi-harmonic sounds of different pitches may be pitch-aligned (i.e., shifted to a common pitch) before morphing, and then the morphed partials may be shifted again to a desired pitch. (Of course, in this example, the desired pitch could also have been chosen as the common pitch before the morph.)

In some cases, the dramatic effect of a morph and its apparent "realism" are enhanced by applying frequency or amplitude deformations that are synchronous with the evolution of the morph. This enhanced realism is particularly important when the sound morph is coupled with a visual presentation. In a computer animation described by Bargar et al.

(2000), *Toy Angst*, slight pitch and amplitude deformations were applied to morphed sounds to accentuate the spasms of a child's squeaky ball as it was deformed into toys of other shapes. These were found to greatly increase the realism of the presentation and the fusion of the audio and visual morphs into a single percept.

Programming Using the Loris API

Loris consists of a C++ class library, a C-linkable procedural interface, interface files that allow Loris extension modules to be built for a variety of scripting languages using David Beazley's Simplified Wrapper Interface Generator (SWIG; available at www.swig.org/ [Beazley 1998]), and standard UNIX/Linux tools that build and install the Loris library, headers, and extension modules for Python and Tcl. (Figure 1 illustrates the use of the C-linkable procedural interface.) Loris is distributed as free software under the GNU General Public License (GPL), and it is available at the Loris web site (www.cerlsoundgroup.org/Loris/).

Figure 3 demonstrates the use of the Loris C++ application programmer's interface (API) to perform a simple sound morph between a clarinet tone and a flute tone. Reassigned bandwidth-enhanced partials for the two tones are imported from SDIF analysis files ("clarinet.sdif" and "flute.sdif," respectively) presumably generated from a reassigned bandwidth-enhanced analysis of the two source sounds.

The partials are channelized and distilled using reference frequency envelopes that track partials at the fundamental frequencies of the source tones (approximately 415 Hz and 291 Hz, respectively). The FrequencyReference object in Loris constructs a reference frequency envelope by finding the strongest partial in a group whose energy is concentrated in a specified range of frequencies. For example, the reference envelope for the flute partials is constructed by finding the strongest partial (i.e., the partial with the greatest sinusoidal energy) having energy concentrated between 250 Hz and 350 Hz. This corresponds to the fundamental partial for the flute tone. The final argument in the FrequencyReference constructor in the example indicates that the reference envelope should consist of 100 samples.

After distilling, the clarinet partials are shifted downward in pitch by six half steps to match the pitch of the flute tone. The distilled and pitch-aligned partials are dilated to align their attack and decay portions, and then a simple morph is performed using a morphing function that transforms the partial frequency, amplitude, and bandwidth envelopes from those of the clarinet to those of the flute between 1 second (the end of the attack portion) and 2 seconds (a half-second before the start of the decay portion). Finally, the morphed partials are synthesized and exported to an AIFF file.

Loris includes a set of interface files that support automatic wrapper-interface generation using SWIG (Beazley 1998). The wrapper code generated by SWIG can be used to build extension modules for a variety of scripting language interpreters, such as Python, Tcl, Perl, and Scheme. Figure 4 shows the same morphing procedure as Figure 3 using the Loris Python API. The Python code is somewhat shorter than the corresponding C++ code because many Loris classes (Channelizer, Distiller, Dilator, etc.) are invoked procedurally in the scripting interface (e.g., channelize(), distill(), dilate(), etc.), and also because Python variables need not be declared before they are used.

Figure 5 shows another morphing operation using the Loris Python API. In this case, a cello tone is morphed with a flute tone. Distilled partials for the two sources are imported from SDIF files, and temporal features are aligned as in previous examples. The flute partials are raised by one half step so that they represent a tone one octave higher than the cello tone. Before morphing, the labels of the flute partials are all doubled so that all the partial labels are even numbers, and the fundamental is labeled 2, the first harmonic above the fundamental is labeled 4, and so on. Consequently, no pitch sweep is perceived in the morph between the cello tone and the flute tone one octave higher. Instead, the even partials of the cello tone are morphed with the flute partials to which they are very near in frequency, and the odd partials of the cello tone fade out over the morph.

Figure 3. Morphing a clarinet tone with a flute tone, using the Loris C++ API. For brevity, `#include` directives have been omitted. Complete code examples are available on the Loris Web site.

```
/*
 *  morph_example.cpp
 */

/* #include directives omitted here, for brevity. */

using namespace Loris;

int main( )
{
  try
  {
    // import raw (undistilled) clarinet partials:
    SdifFile sdifclar("clarinet.sdif");
    PartialList & clar = sdifclar.partials();

    // compute a frequency reference envelope based on the
    // clarinet's fundamental partial (about 415 Hz), and
    // distill the clarinet partials (in place):
    FrequencyReference clRef( clar.begin(), clar.end(), 350, 500, 100 );
    Channelizer ch( clRef.envelope() , 1 );
    ch.channelize( clar.begin(), clar.end() );
    Distiller still;
    still.distill( clar );

    // shift the pitch of the clarinet partials down by six half
    // steps to match the pitch of the flute tone:
    double pscale = std::pow(2., (0.01 * -600) /12.);
    PartialList::iterator pIter;
    for ( pIter = clar.begin(); pIter != clar.end(); ++pIter )
    {
      Partial::iterator bpIter;
      for ( bpIter = pIter->begin(); bpIter != pIter->end(); ++bpIter )
      {
        bpIter->setFrequency( bpIter->frequency() * pscale );
      }
    }

    // import raw (undistilled) flute partials:
    SdifFile sdifflut("flute.sdif");
    PartialList & flut = sdifflut.partials();
```

Figure 3. Continued.

```
    // compute a frequency reference envelope based on the
    // flute's fundamental partial (about 291 Hz), and
    // distill the flute partials (in place):
    FrequencyReference flRef( flut.begin(), flut.end(), 250, 350, 100 )
    ch = Channelizer( flRef.envelope(), 1 );
    ch.channelize( flut.begin(), flut.end() );
    still.distill( flut );

    // align temporal features in the clarinet and flute, move
    // the features to the desired times in the morphed sound:
    double flut_times[] = { 0.4, 1., 2.2, 2.6 };
    double clar_times[] = { 0.2, 1., 2., 3. };
    double desired_times[] = { 0.1, 1., 2.5, 3. };

    Dilator dliate_clar( clar_times, clar_times + 4, desired_times );
    dliate_clar.dilate( clar.begin(), clar.end() );

    Dilator dilate_flut( flut_times, flut_times + 4, desired_times );
    dilate_flut.dilate( flut.begin(), flut.end() );

    // morph the clarinet smoothly into the flute from
    // 1 to 2 seconds:
    BreakpointEnvelope morphenv;
    morphenv.insertBreakpoint( 1, 0 );
    morphenv.insertBreakpoint( 2, 1 );
    Morpher m( morphenv );
    m.morph( clar.begin(), clar.end(), flut.begin(), flut.end() );

    // synthesize the morphed sound:
    const double Srate = 44100.0;
    std::vector< double > v( long( ( 3.1 /* seconds */ * Srate ) ) );
    Synthesizer synth( Srate, v );
    synth.synthesize(  m.partials().begin(), m.partials().end() );
    AiffFile::Export( "morph.aiff", Srate, 1, 16, v.begin(), v.end() );
  }
  catch( std::exception & ex )
  {
    std::cerr << "Caught exception: " << ex.what() << std::endl;
    return 1;
  }

  return 0;
}
```

Figure 4. Morphing a clarinet tone with a flute tone using the Loris Python API.

```
import loris

# import raw (undistilled) clarinet partials:
clar = loris.importSdif("clarinet.sdif")

# compute a frequency reference envelope based on the
# clarinet's fundamental partial (about 415 Hz), and
# distill the clarinet partials (in place):
clar_env = loris.createFreqReference( clar, 350, 500, 20 )
loris.channelize( clar, clar_env, 1 )
loris.distill( clar )

# shift the pitch of the clarinet partials down by six half
# steps to match the pitch of the flute tone:
loris.shiftPitch( clar, loris.BreakpointEnvelopeWithValue( -600 ) )

# import raw (undistilled) flute partials:
flut = loris.importSdif("flute.sdif")

# compute a frequency reference envelope based on the
# flute's fundamental partial (about 291 Hz), and
# distill the flute partials (in place):
flut_env = loris.createFreqReference( flut, 250, 500, 20 )
loris.channelize( flut, flut_env, 1 )
loris.distill( flut )

# align temporal features in the clarinet and flute, move
# the features to the desired times in the morphed sound:
flut_times = [ 0.4, 1., 2.2, 2.6 ]
clar_times = [ 0.2, 1., 2., 3. ]
desired_times = [ 0.1, 1., 2.5, 3. ]

# note: feature times are passed to the dilate function
# as strings, because no common representation of
# collections (lists, tuples, etc) is available:
loris.dilate( flut, str(flut_times), str(desired_times) )
loris.dilate( clar, str(clar_times), str(desired_times) )

# morph the clarinet smoothly into the flute from 1 to 2 seconds:
morphenv = loris.BreakpointEnvelope()
morphenv.insertBreakpoint( 1, 0 )
morphenv.insertBreakpoint( 2, 1 )
morphed = loris.morph( clar, flut, morphenv, morphenv, morphenv )

# synthesize the morphed sound:
Srate = 44100.
v = loris.synthesize( morphed, Srate )
loris.exportAiff( 'morph.aiff', v, Srate, 1, 16 )
```

Figure 5. Morphing a cello tone with a flute tone one octave higher using the Loris Python API.

```
import loris

# import distilled flute and cello partials:
flut = loris.importSdif("flute.distilled.sdif")
cell = loris.importSdif("cello.distilled.sdif")

# align temporal features in the clarinet and flute, move
# the features to the desired times in the morphed sound:
flut_times = [ 0.4, 1.0, 2.2, 2.6 ]
cell_times = [ 0.15, 0.8, 3.0, 4.4 ]
desired_times = [ 0.15, 1., 2.5, 3.0 ]

# note: feature times are passed to the dilate function
# as strings, because no common representation of
# collections (lists, tuples, etc) is available:
loris.dilate( flut, str(flut_times), str(desired_times) )
loris.dilate( cell, str(cell_times), str(desired_times) )

# raise the pitch of the flute partials by one halfstep,
# to be one octave higher than the cello tone:
loris.shiftPitch( flut, loris.BreakpointEnvelopeWithValue( 100 ) )

# double the labels of all the flute partials:
it = flut.begin()
while not it.equals(flut.end()):
       part = it.partial()
       part.setLabel( part.label() * 2)
       it = it.next()

# morph the cello smoothly into the flute from
# 1 to 2 seconds:
morphenv = loris.BreakpointEnvelope()
morphenv.insertBreakpoint( 1, 0 )
morphenv.insertBreakpoint( 2, 1 )
morphed = loris.morph( cell, flut, morphenv, morphenv, morphenv )

# synthesize the morphed sound:
Srate = 44100.
v = loris.synthesize( morphed, Srate )
loris.exportAiff( 'cellute.aiff', v, Srate, 1, 16 )
```

Applications

Loris provides only programmatic access to the reassigned bandwidth-enhanced sound model and to the morphing and manipulation functionality described in the "Sound Morphing in Loris" section of this article, and is therefore usable only with difficulty by non-programmers. Recently, however, several software tools have been developed that allow non-programmers to access the powerful sound modeling, morphing, and manipulation features in Loris.

Fossa

To bridge the gap between sound designers and computer programmers, a graphical control application called *Fossa* is under development and distributed as part of the Loris project. Fossa includes both a graphical representation of reassigned bandwidth-enhanced analysis data and the ability to audition sounds rendered from such data, allowing the user to see and hear the results of different manipulations. Reassigned bandwidth-enhanced partials can be imported from Sound Description Interchange Format (SDIF) files (Wright et al. 1999), which are supported by Loris. Alternatively, Fossa can perform reassigned bandwidth-enhanced analysis of AIFF files according to user-specified analysis parameters.

Imported partials are displayed in amplitude, frequency, and noise plots. The parameter envelopes for a collection of imported partials are plotted against time in distinct amplitude, frequency, and noisiness graphs. Several sounds can be imported into Fossa at once, and manipulations applied to the displayed partial collection (selected from a pop-up menu) are immediately reflected in the parameter envelope plots. Manipulations available in Fossa include parameter scaling operations (such as pitch shifting) as well as channelization and distillation in preparation for morphing. This interactive graphical representation makes it possible to visualize the effects of complex operations like distillation and to evaluate the suitability of various channelization strategies. Fossa's frequency and amplitude envelope displays are shown in Figures 6 and 7, respectively.

Fossa provides a graphical interface for interactive construction and application of morphing control functions. Independent breakpoint envelopes for morphing frequency, amplitude, and noisiness can be assembled and manipulated in the click-and-drag editor shown in Figure 8 and applied using the Loris morphing algorithms. In the figure, a clarinet sound is morphed into a cello sound over approximately 4 sec. The frequencies are interpolated linearly over the duration of the morph, whereas the amplitudes are morphed more quickly and the noisiness more slowly over the first half of the morph. The morphed sound is added to the list of imported sounds for inspection or further manipulation. Finally, morphed or otherwise manipulated partials can be exported to an SDIF data file, or they can be rendered for audition or export to an AIFF file.

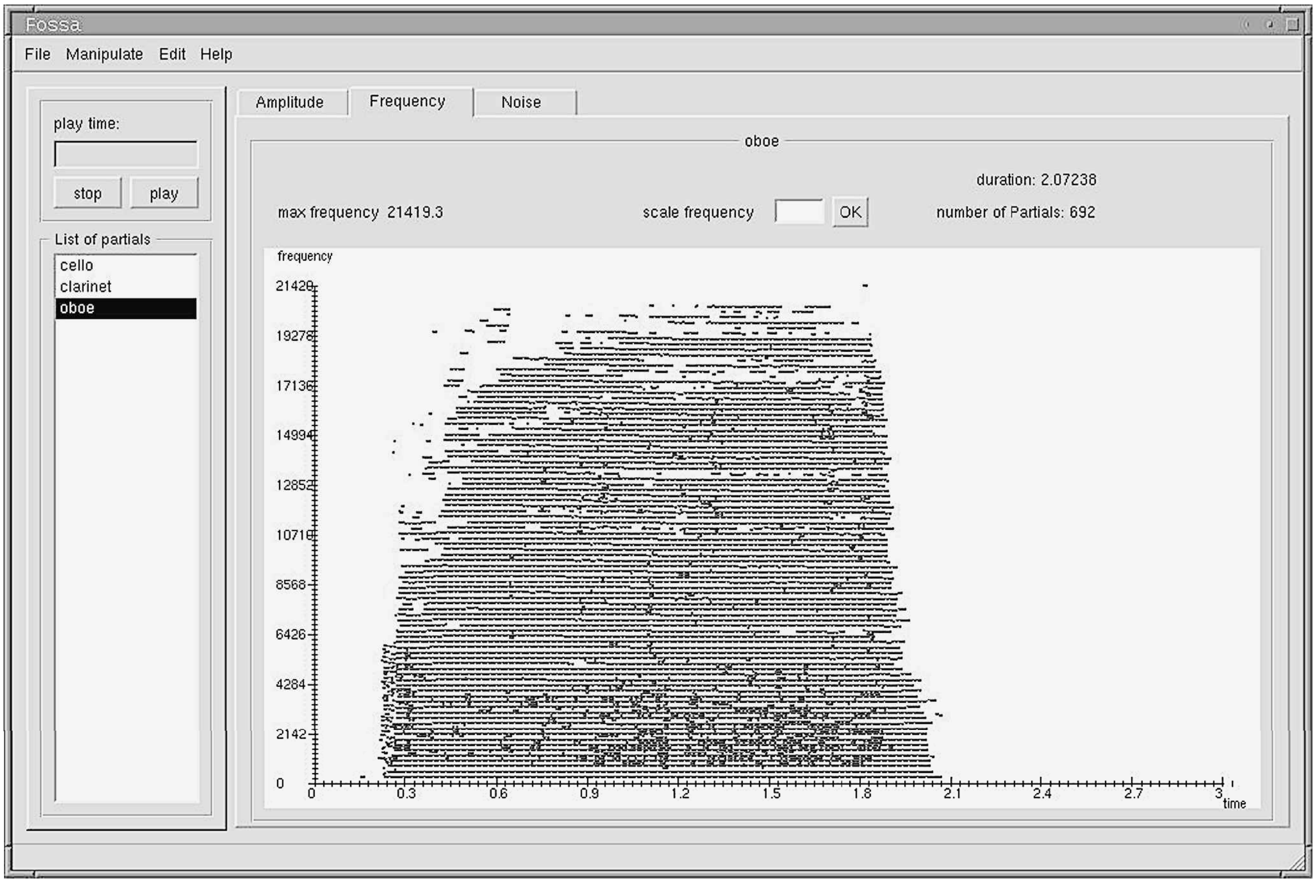
Fossa was developed by Susanne Lefvert as a master's thesis for Lulea University of Technology in collaboration with University of Chicago, under the guidance of Mike O'Donnell. Fossa is distributed as free software under the GNU General Public License (GPL), and distributions of the beta release for UNIX and Linux operating systems are available at the Loris Web site.

Real-Time Synthesis in Kyma

Together with Kurt Hebel of Symbolic Sound Corporation, we have implemented a stream-based real-time bandwidth-enhanced synthesizer using the Kyma Sound Design Workstation (Hebel and Scaletti 1994).

Many real-time synthesis systems allow sound designers to manipulate streams of samples. In our real-time bandwidth-enhanced implementation, we work with streams of data that are not time-domain samples. Rather, our *Envelope Parameter Streams* encode frequency, amplitude, and band-

Figure 6. Parameter envelope display in Fossa showing frequency envelopes for partials in a reassigned bandwidth-enhanced analysis of an oboe tone.



width envelope parameters for each bandwidth-enhanced partial (Haken, Tellman, and Wolfe 1998; Haken, Fitz, and Christensen forthcoming).

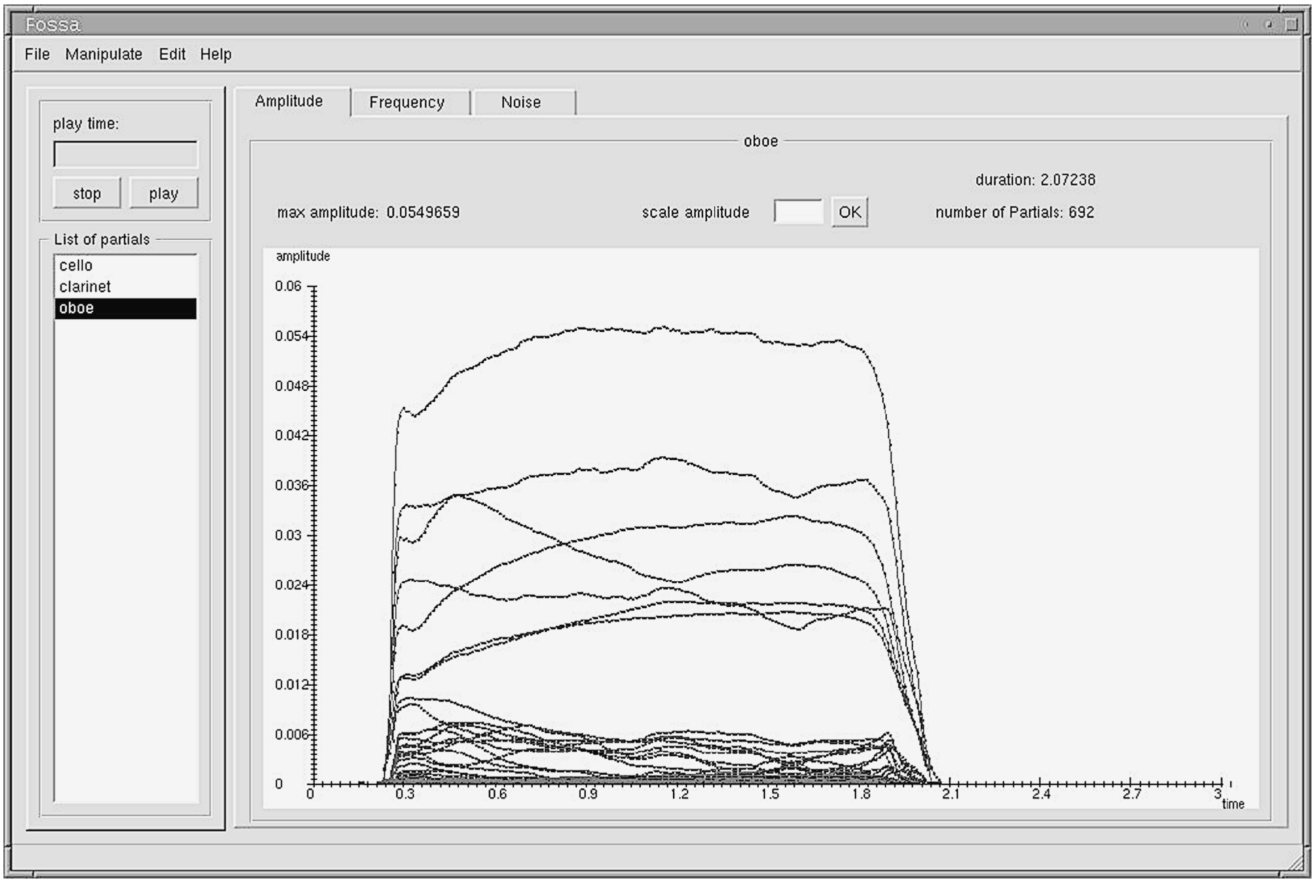
The Kyma data flow graph shown in the top half of Figure 9 produces a polyphonic real-time timbre morph between sounds analyzed in Loris. The data-flow graph depicts six interconnected Kyma Sound Objects and a speaker output. The parameters for one of the Sound Objects (the *MultiSpectrum-InRam*) are shown in the bottom half of Figure 9. The parameters for the other Sound Objects are not shown in this figure. Real-time data flow is from left to right.

The *MultiSpectrumInRAM* Sound Object reads amplitude, frequency, and bandwidth envelopes from reassigned bandwidth-enhanced analysis data files prepared by Loris. Weighted averages of envelopes are used generate a morphed envelope parameter stream.

The envelope parameter streams feeds into the *Oscillators* object, a bank of bandwidth-enhanced sinusoidal oscillators. Each oscillator synthesizes a single bandwidth-enhanced partial in the morphed sound, and all oscillators are evaluated each sample time. The *Noise* and *LowPassFilter* sounds provide the *Oscillators* object with band-limited noise values required for synthesis. The output of *Oscillators* is the time-domain sum of all its oscillators.

The *timbreControlSpace* object is a script that provides some of the parameters used by *Multi-*

Figure 7. Parameter envelope display in Fossa showing amplitude envelopes for partials in a reassigned bandwidth enhanced analysis of an oboe tone.

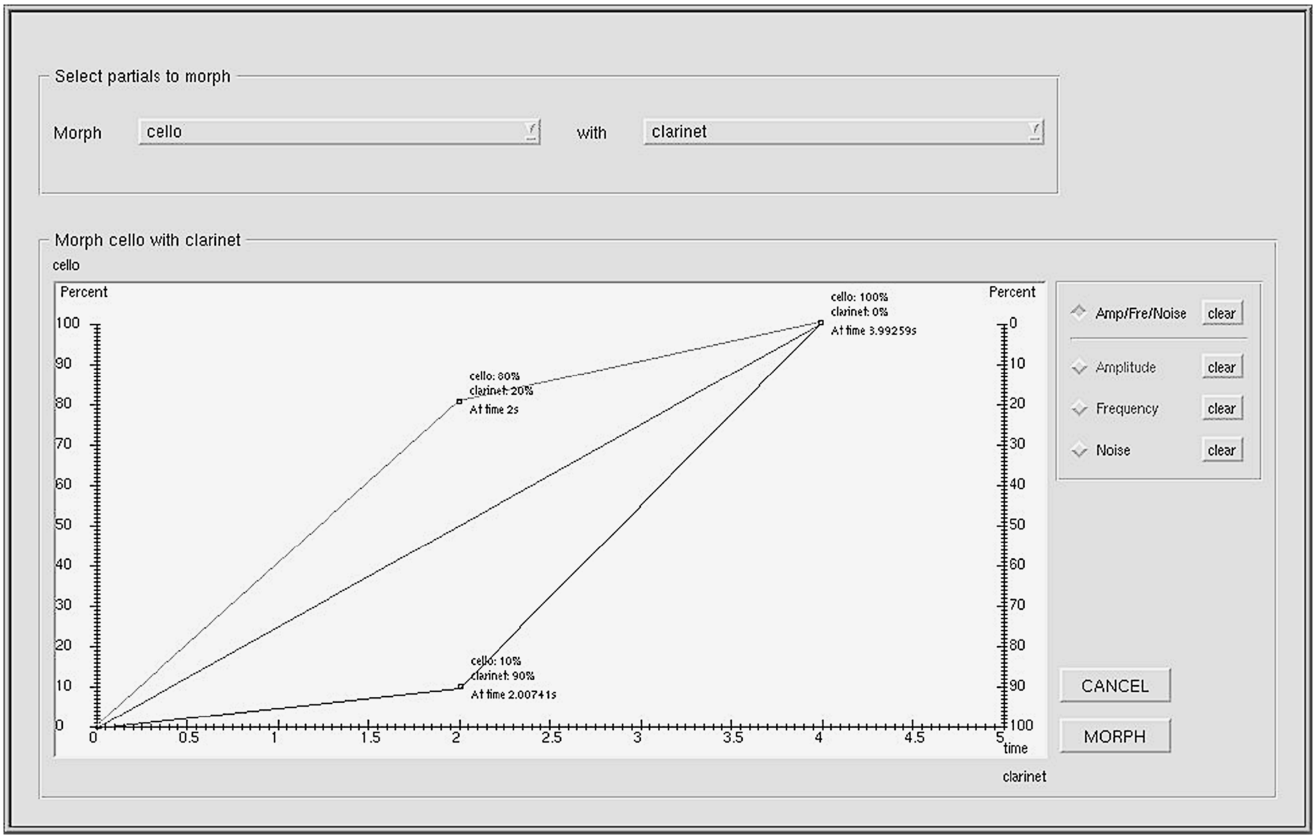


SpectrumInRam. The *MIDIPolyphony* object produces multiple copies the subgraph to its left. In this example, *MIDIPolyphony* parameters (not shown) are set to produce ten copies of the subgraph, resulting in ten-voice polyphony. The *MIDIPolyphony* adds the time-domain samples from its ten duplicated subgraphs, and that sum is sent to the speaker.

The parameters for *MultiSpectrumInRAM* are shown in the bottom half of Figure 9. File names are listed for the *Analyses* parameter. These are source timbres analyzed in Loris. This example has 24 file names, with timbres derived from recordings of cello, violin, oboe, and bassoon at various pitches and dynamic levels. The source timbres have been time-aligned (to align manually specified temporal events) and frequency-aligned (to align fundamental frequencies). The 24 source timbres are indexed numerically by the expressions in the *Indices* parameter. This example has eight expressions in the *Indices* parameter, which continually select eight of the total 24 source timbres for an eight-way morph. The eight expressions in the *AmpWeights* parameter specify the morphing envelope for sine and noise amplitudes of the partials in the eight source timbres in the eight-way morph. The eight expressions in the *PitchWeights* parameter (in this case identical to the *AmpWeights* expressions) specify the morphing envelope for frequencies of the partials in the eight-way morph.

The *FirstPartial* and *NbrPartials* parameters

Figure 8. *Morphing function editor in Fossa, showing individual frequency, amplitude, and noisiness morphing functions for a morph between a cello and a clarinet.*



Select partials to morph

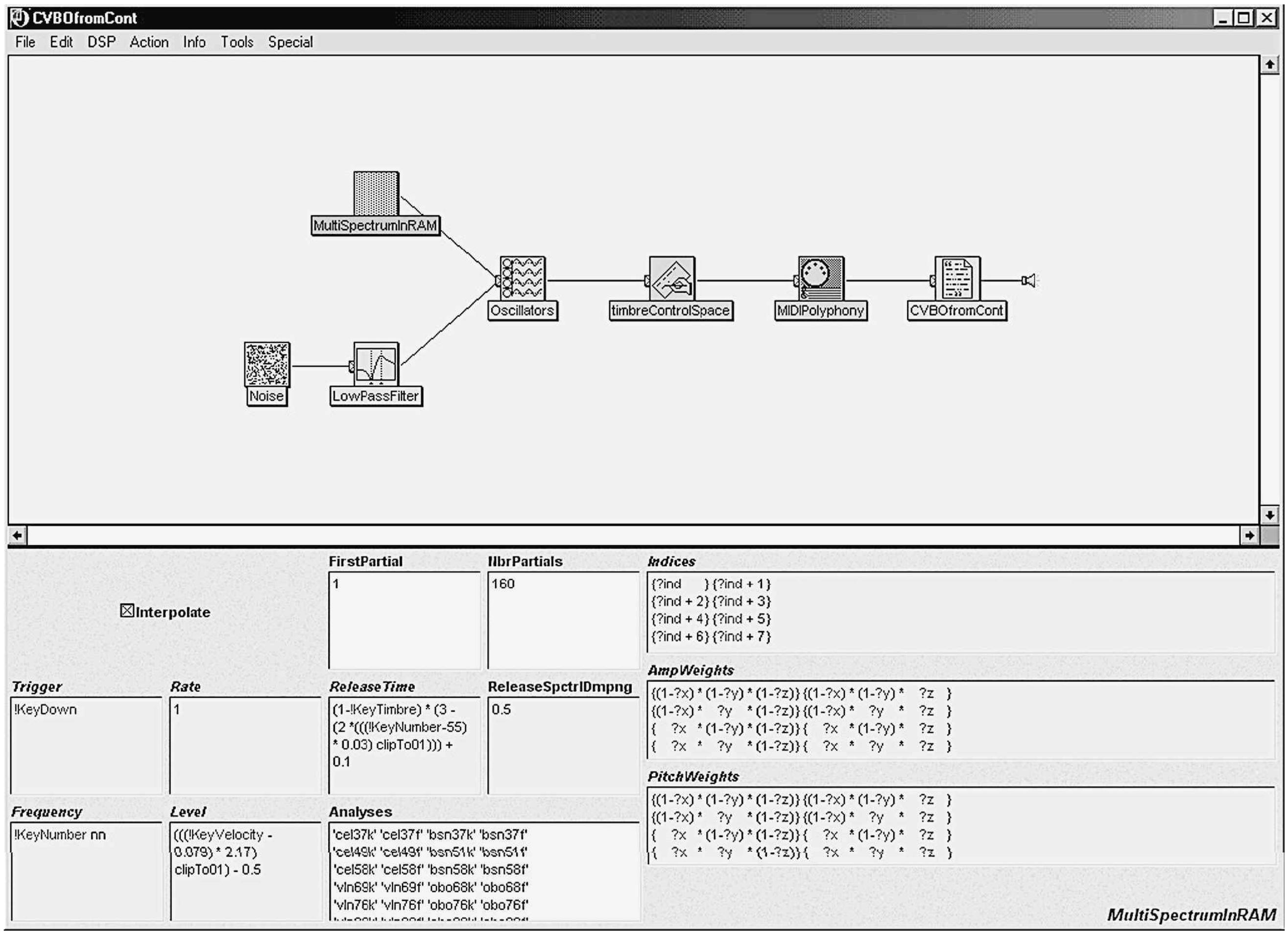
Morph cello with clarinet

specify that 160 partials should be used from each source timbre. Setting the *Rate* parameter to 1 indicates the rate of timbral evolution (the step rate through the time-aligned envelopes) matches the source timbres. The *Trigger* parameter specifies a gate signal to start and stop notes. The *ReleaseTime* parameter specifies the release, and the *ReleaseSpctrlDmping* specifies faster damping of higher partials during the release. *Frequency* specifies the frequency of the synthesis, and the *Level* parameter specifies an overall attenuation for the morphed amplitudes.

Several of the parameters are functions of *KeyDown*, *KeyNumber*, *KeyTimbre*, and *KeyVelocity*. These correspond to the real-time MIDI-derived signals *gate*, *fractional continuous note number* (i.e., MIDI note number plus pitch bend), *continuous controller value* (0–127), and *fractional continuous volume* (i.e., MIDI velocity scaled coninuously by channel volume), respectively.

The *ReleaseTime* object includes an expression that results in a long release for the cello timbres, a medium release time for the violin timbres, and a short release time for the bassoon and oboe timbres. The *Level* parameter provides loud and quiet variations as a function of *KeyVelocity*.

The *ind*, *x*, *y*, and *z* values are provided by the script contained in *timbreControlSpace*, shown in Figure 10. The *ind* parameter determines which eight of the 24 source timbres are to be morphed based on the MIDI pitch (*KeyNumber*) and the pitch of the source timbres. The *x* parameter controls morphing based on the current fractional *KeyNumber*, and the *y* parameter controls morphing based on the *KeyTimbre*. Finally, the *z* parameter controls morphing based on *KeyVelocity*.



☒Interpolate

FirstPartial: 1

NbrPartials: 160

Indices:
```
{?ind    } {?ind + 1}
{?ind + 2} {?ind + 3}
{?ind + 4} {?ind + 5}
{?ind + 6} {?ind + 7}
```

Trigger: !KeyDown

Rate: 1

ReleaseTime: (1-!KeyTimbre) * (3 - (2 *(((!KeyNumber-55) * 0.03) clipTo01))) + 0.1

ReleaseSpctrlDmpng: 0.5

AmpWeights:
```
{(1-?x) * (1-?y) * (1-?z)} {(1-?x) * (1-?y) *   ?z  }
{(1-?x) *   ?y   * (1-?z)} {(1-?x) *   ?y   *   ?z  }
{   ?x  * (1-?y) * (1-?z)} {   ?x  * (1-?y) *   ?z  }
{   ?x  *   ?y   * (1-?z)} {   ?x  *   ?y   *   ?z  }
```

PitchWeights:
```
{(1-?x) * (1-?y) * (1-?z)} {(1-?x) * (1-?y) *   ?z  }
{(1-?x) *   ?y   * (1-?z)} {(1-?x) *   ?y   *   ?z  }
{   ?x  * (1-?y) * (1-?z)} {   ?x  * (1-?y) *   ?z  }
{   ?x  *   ?y   * (1-?z)} {   ?x  *   ?y   *   ?z  }
```

Frequency: !KeyNumber nn

Level: (((!KeyVelocity - 0.079) * 2.17) clipTo01) - 0.5

Analyses:
```
'cel37k' 'cel37f' 'bsn37k' 'bsn37f'
'cel49k' 'cel49f' 'bsn51k' 'bsn51f'
'cel58k' 'cel58f' 'bsn58k' 'bsn58f'
'vln69k' 'vln69f' 'obo68k' 'obo68f'
'vln76k' 'vln76f' 'obo76k' 'obo76f'
```

MultiSpectrumInRAM

Figure 9. Data flow graph consisting of six Sound Objects (top) and parameters for one of the Sound Objects (bottom) in the Symbolic Sound Kyma environment. This data flow graph produces a polyphonic timbre morph between sounds using reassigned bandwidth-enhanced analysis data prepared in Loris.

The dataflow graph in Figure 9 and the script in Figure 10 are configured for use with a *Continuum Fingerboard*. The Continuum Fingerboard is a new MIDI controller that allows continuous control over each note in a performance (Haken, Tellman, and Wolfe 1998). It resembles a traditional keyboard in that it is approximately the same size and is played with ten fingers. Like keyboards supporting polyphonic aftertouch, it continually measures each finger's pressure. The Continuum Fingerboard also resembles a fretless string instrument in that it imposes no discrete finger positions; any pitch may be played, and smooth glissandi are possible. It tracks in three dimensions the position for each finger pressing on the playing surface. These continuous three-dimensional outputs (mapped to *KeyNumber*, *KeyVelocity*, and *KeyTimbre* in Kyma) are a convenient source of control parameters for real-time manipulations.

Using Loris with Csound

A set of Csound unit generators supporting modified synthesis and morphing of reassigned bandwidth-enhanced model data is under development. Csound is a flexible and extensible orchestra/score system in the style of "Music N" languages

Figure 10. Smalltalk script associated with the timbreControlSpace *Sound Object in Figure 9.*

```
| slicePitches sliceWidths w nnToSlice slice lownn scalenn zvalue |

"Pitches of the source timbres."
slicePitches := #( 0 37 49 58 68 76 88 109.52 ).

"Compute width of each slice in the timbre space."
sliceWidths := (1 to: (slicePitches size - 1)) collect:
            [ :i | (slicePitches at: (i+1)) - (slicePitches at: i)].

"Make notenumber-to-slice lookup table."
w := WriteStream on: (Array new).
(1 to: (sliceWidths size)) do:
            [ :i | w next: (sliceWidths at: i) put: (i-1) ].
nnToSlice := w contents.

"Slice and pitch for the current cube in timbre control space."
slice := !KeyNumber of: nnToSlice.      "slice number, left side of cube"
lownn := slice of: slicePitches.        "pitch, left side of cube"

"Get pitch scale for inside of cube."
scalenn := slice of: (sliceWidths collect: [ :tot | tot inverse]).

"Velocity values are MIDI 10 to 127, make z value full range 0 to 1."
"0.079 = 10/127  1.085 = 127/117"
zvalue := ((!KeyVelocity - 0.079) * 1.085) clipTo01.

"Provide ?ind, ?x, ?y, and ?z for the subgraph."
Oscillators
       start: 0 s
       duration: 9999 s
       "Index into analyses list; lower left corner of the cube."
       ind: ((slice * 4) + 1)
       "x-weighting: relative pitch position in the current cube"
       x: ((!KeyNumber - lownn) * scalenn)
       y: !KeyTimbre                     " y weighting: front/back "
       z: ((2 * zvalue) - 0.5) clipTo01. " z weighting: pressure "
```

and is one of the most popular and widely distributed software synthesis applications (Boulanger 2000). Csound supports a wide variety of synthesis techniques, including analysis-based techniques, such as the phase vocoder (Dolson 1986) and linear predictive coding.

Csound unit generators for importing and manipulating reassigned bandwidth-enhanced analysis data will provide Csound's huge user community with the sound morphing and manipulation capabilities of Loris. These tools will enable Csound users to integrate high-fidelity sound morphing and transformation into their own compositions and sound designs, and they will further allow Loris users to avail themselves of the rich set of control structures and sound design tools available in Csound.

Bandwidth-enhanced additive synthesis is performed by the Csound unit generators called *lorisread* and *lorisplay.* The *lorisplay* unit generator renders a stored set of reassigned bandwidth-enhanced partials using the bandwidth-enhanced sinusoidal oscillator implemented in the Loris library. The frequency, amplitude, bandwidth (noisiness), and phase envelopes for the partials are imported from analysis data stored in Sound Description Interchange Format (SDIF) data files (Wright et al. 1999) using *lorisread*. The *lorisread* unit generator stores the partial data in a labeled location in memory for access by other Csound unit generators.

The syntax for the Loris Csound modules is shown in Figure 11. First, *lorisread* is initialized with several parameters: the name of the SDIF con-

Figure 11. The syntax of the Csound unit generators supporting bandwidth-enhanced additive synthesis and sound morphing based on the Loris library.

```
lorisread  ktimpnt, ifilcod, istoreidx, kfreqenv, kampenv,
           kbwenv[, ifadetime] ar

lorisplay  ireadidx, kfreqenv, kampenv, kbwenv

lorismorph isrcidx, itgtidx, istoreidx, kfreqmorphenv,
           kampmorphenv, kbwmorphenv
```

trol file (*ifilcod*); an arbitrary integer index (*istoreidx*) used to identify the memory location of the partial data for reference by other generators; and, optionally, the partial fade time (*ifadetime*), used to control the turn-on and turn-off rate for partials having non-zero initial and final amplitudes. Because the bandwidth-enhanced oscillator parameters are updated at the global control rate (*krate*) specified in the Csound orchestra, the value of the fade time is approximate and represents the minimum time over which partials fade in and out. If unspecified, the fade time defaults to 0.

The *ktimpnt* parameter is a control-rate time index into the reassigned bandwidth-enhanced analysis data. It is an absolute time index, in seconds, and functions identically to the time index parameter used by other Csound resynthesis unit generators, such as the phase vocoder *pvoc* and the LPC resynthesis modules *lpread* and *lpreson*, which allow the sound to be rendered forwards or backwards, or at varying speeds.

Both *lorisread* and *lorisplay* apply control-rate scaling to the frequency, amplitude, and bandwidth envelopes of the reassigned bandwidth-enhanced partials. The parameter *kfreqenv* is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and 0.5 transposes down an octave. Similarly, *kampenv* and *kbwenv* scale the partial amplitudes and bandwidth, and a value of 1 leaves those parameters unmodified.

The *lorismorph* unit generator performs sound morphing using two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions.

The syntax for *lorismorph* is shown in Figure 11. Like *lorisplay*, *lorismorph* uses integer indices to refer to its source and target partial sets. A third index specifies the location of the morphed partial set. The morphed partials can be rendered using *lorisplay* or subjected to further morphing. Because each set of partials has its own time index (the *ktimpnt* parameter of *lorisread*), feature-alignment can be performed in the Csound orchestra as previously described.

The sound morph is described by three control-rate morphing envelopes. First, *kfreqmorphenv* describes the interpolation of partial frequency values in the two source sounds. When *kfreqmorphenv* is 0, partial frequencies are obtained from the partials in *ifilename0*. When *kfreqmorphenv* is 1, partial frequencies are obtained from the partials in *ifilename1*. When *kfreqmorphenv* is between 0 and 1, the partial frequencies are interpolated between the two sources. Interpolation of partial amplitudes and bandwidth (noisiness) coefficients are similarly described by *kampmorphenv* and *kbwmorphenv*.

The use of *lorismorph* is illustrated by the Csound orchestra example in Figure 12. The instrument in Figure 12 performs a sound morph between a flutter-tongued trombone tone and a cat's meow using reassigned bandwidth-enhanced partials stored in "trombone.sdif" and "meow.sdif." The data in these SDIF files have been channelized and distilled to establish correspondences between partials as previously described.

The two sets of partials are imported using *lorisread*, which stores the partials in memory locations labeled 1 and 2. Each of the original sounds has

Figure 12. A Csound sound morphing example.

```
; Morph the partials in trombone.sdif into the
; partials in meow.sdif. The start and end times
; for the morph are specified by parameters p4
; and p5, respectively. The morph occurs over the
; second of four pitches in each of the sounds,
; from .75 to 1.2 seconds in the flutter-tongued
; trombone tone, and from 1.7 to 2.2 seconds in
; the cat's meow. Different morphing functions are
; used for the frequency and amplitude envelopes,
; so that the partial amplitudes make a faster
; transition from trombone to cat than the frequencies.
; (The bandwidth envelopes use the same morphing
; function as the amplitudes.)
;
instr 1
   ionset  =      p4
   imorph  =      p5 - p4
   irelease =     p3 - p5

   im0   =     p4
   im1   =     p5
   itmorph =   im1-im0

   kttbn  linseg   0, ionset, .75, imorph, 1.2, irelease, 2.4
   ktmeow  linseg   0, ionset, 1.7, imorph, 2.2, irelease, 3.4

   kmfreq  linseg   0, ionset, 0,  75*imorph, .25, .25*imorph, 1,
       irelease, 1
   kmamp  linseg   0, ionset, 0, .75*imorph, .9, .25*imorph, 1,
       irelease, 1

   lorisread  kttbn, "trombone.sdif", 1, 1, 1, 1, .001
   lorisread  ktmeow, "meow.sdif", 2, 1, 1, 1, .001
   lorismorph 1, 2, 3, kmfreq, kmamp, kmamp
   asig lorisplay  3, 1, 1, 1
   out asig
endin
```

four notes, and the morph is performed over the second note in each sound (from 0.75 to 1.2 sec in the flutter-tongued trombone tone, and from 1.7 to 2.2 sec in the cat's meow). The different time index functions, *kttbn* and *ktmeow*, align those segments of the source and target partial sets with the specified morph start, morph end, and overall duration parameters. Two different morphing functions are used, so that the partial amplitudes and bandwidth coefficients morph quickly from the trombone values to the cat's-meow values, and the frequencies make a more gradual transition. The morphed partials are stored in a memory location labeled 3 and rendered by the subsequent *lorisplay* instruction.

They could also have been used as a source for another morph in a three-way morphing instrument.

The *lorisread*, *lorisplay*, and *lorismorph* unit generators are not currently part of the standard Csound source or binary distributions, but Csound is designed to be extensible by incorporation of user-defined unit generators. Following several steps outlined in Csound documentation (see, for example, Boulanger 2000) and described in detail in a document distributed with Loris, these and other custom unit generators can be added to the Csound source code and built into an enhanced Csound application.

Conclusion

The reassigned bandwidth-enhanced analyzer implemented in the Loris software library supports high-fidelity, robust modeling of a wide variety of sounds. The analyzer includes a small set of non-interacting parameters that are easily tuned to arrive at an optimal configuration for a particular sound. Loris provides manipulation and transformation operations on the reassigned bandwidth-enhanced model data needed to implement sound morphing. Previously accessible only through programmatic interfaces (C/C++ and various scripting languages), a variety of new software tools have been presented that make the sound modeling and morphing capabilities of Loris available to composers, sound designers, and other non-programmers.

References

Allen, J. B., and L. R. Rabiner. 1977. "A Unified Approach to Short-Time Fourier Analysis and Synthesis." *Proceedings of the IEEE* 65(11):1558–1564.

Auger, F., and P. Flandrin. 1995. "Improving the Readability of Time-Frequency and Time-Scale Representations by the Reassignment Method." *IEEE Transactions on Signal Processing* 43(5):1068–1089.

Bargar, R., A. Betts, I. Choi, and K. Fitz. 2000. "Models and Deformations in Procedural Synchronous Sound for Animation." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 205–208.

Beazley, D. M. 1998. "SWIG and Automated C/C++ Scripting Extensions." *Dr. Dobbs Journal* 282:30–36.

Boulanger, R., ed. 2000. *The Csound Book*. Cambridge, Massachusetts: MIT Press.

de Cheveign, A., and H. Kawahara. 2001. "Comparative Evaluation of F0 Estimation Algorithms." *Proceedings of Eurospeech 2001*. Bonn, Germany: International Speech Communication Association.

Dodge, C., and T. A. Jerse. 1997. *Computer Music: Synthesis, Composition, and Performance*, 2nd ed. New York: Shirmer Books.

Dolson, M. 1986. "The Phase Vocoder: A Tutorial." *Computer Music Journal* 10(4):14–27.

Fitz, K., and L. Haken. 1996. "Sinusoidal Modeling and Manipulation Using Lemur." *Computer Music Journal* 20(4):44–59.

Fitz, K., L. Haken, and P. Christensen. 2000a. "A New Algorithm for Bandwidth Association in Bandwidth-Enhanced Additive Sound Modeling." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 384–387.

Fitz, K., L. Haken, and P. Christensen. 2000b. "Transient Preservation Under Transformation in an Additive Sound Model." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 392–395.

Haken, L., K. Fitz, and P. Christensen. Forthcoming. "Beyond Traditional Sampling Synthesis: Real-Time Timbre Morphing Using Additive Synthesis." In J. W. Beauchamp, ed. *Sound of Music: Analysis, Synthesis, and Perception*. Berlin: Springer-Verlag.

Haken, L., E. Tellman, and P. Wolfe. 1998. "An Indiscrete Music Keyboard." *Computer Music Journal* 221:30–48.

Hebel, K., and C. Scaletti. 1994. "A Framework for the Design, Development, and Delivery of Real-Time Software-Based Sound Synthesis and Processing Algorithms." *Audio Engineering Society Preprint* A-3(3874).

Hess, W. 1983. *Pitch Determination of Speech Signals*. Berlin: Springer-Verlag.

Kent, J., W. Carlson, and R. Parent. 1992. "Shape Transformation for Polyhedral Objects." *Proceedings for the 19th Annual Conference on Computer Graphics and Interactive Techniques.* New York: ACM Press, pp. 47–54.

Lazarus, F., and A. Verroust. 1998. "Three-Dimensional

Metamorphosis: A Survey.'' *The Visual Computer* 14(8/9): 387–389.

Masri, P., A. Bateman, and N. Canagarajah. 1997. ''A Review of Time-Frequency Respresentations with Applications to Sound/Music Analysis-Resynthesis.'' *Organised Sound* 2(3):193–205.

McAulay, R. J., and T. F. Quatieri. 1986. ''Speech Analysis/Synthesis Based on a Sinusoidal Representation.'' *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34(4):744–754.

Serra, X., and J. O. Smith. 1990. ''Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic Plus Stochastic Decomposition.'' *Computer Music Journal* 14(4):12–24.

Tellman, E., L. Haken, and B. Holloway. 1995. ''Timbre Morphing of Sounds with Unequal Numbers of Features.'' *Journal of the Audio Engineering Society* 43(9):678–689.

Wolberg, G. 1998. ''Image Morphing: A Survey.'' *The Visual Computer* 14(8/9): 360–372.

Wright, M., et al. 1999. ''Audio Applications of the Sound Description Interchange Format Standard.'' *Audio Engineering Society 107th Convention* preprint #5032. New York: Audio Engineering Society.