UNIT-GENERATORS CONSIDERED HARMFUL (FOR MICROSOUND SYNTHESIS): A NOVEL PROGRAMMING MODEL FOR MICROSOUND SYNTHESIS IN LCSYNTH

Hiroki NISHINO
NUS Graduate School for Integrative Sciences & Engineering, National University of Singapore
g0901876@nus.edu.sg

Naotoshi OSAKA
Dept. of Information Systems & Multimedia Design, Tokyo Denki University
Osaka@dendai.ac.jp

Ryohei NAKATSU
IDM Institute, National University of Singapore
elenr@nus.edu.sg

ABSTRACT

In this paper, we describe a novel programming model for microsound synthesis techniques in LCSynth, a strongly-timed sound synthesis language, with concrete examples in granular synthesis and waveset synthesis. Instead of encapsulating microsound synthesis techniques inside unit-generators, LCSynth provides objects and manipulations for microsound synthesis.

We discuss the benefits of such a language design for creative explorations in the domain of microsounds and why the traditional unit-generator concept may not be very appropriate for this domain. Such a discussion is beneficial to the further development of new sound synthesis frameworks and computer music languages.

1. INTRODUCTION

The unit-generator concept is one of the most important domain-specific core abstractions developed in the history of computer music. While there exist many varieties in how the concept is actually implemented and integrated into the language design, many computer music languages are built upon this concept even today.

The unit-generator concept is still quite beneficial for various sound synthesis techniques. Its encapsulation of signal processing algorithms with the common interfaces makes it able to compose more complex signal processing modules as the interconnected unit-generators. Most implementations, except very few, autonomously perform signal processing without explicit scheduling of the timing when to compute the output; such features significantly reduce the amount of effort that users must make in implementing various sound synthesis algorithms.

However, it should be reconsidered if the traditional unit-generator concept is still appropriate for microsound synthesis techniques; some previous works discussed the related issues such as better software design or appropriate extensions to sound synthesis frameworks, which would be more suitable for microsound synthesis techniques [1,4].

In our previous publication [11], we argued the difficulty in programming microsound synthesis techniques, from the perspective of *structural misfit* [3] between the sound synthesis frameworks built on the unit-generator concept and the conceptualization of microsound synthesis by users. We discussed this conceptual gap as a significant cause of the difficulty involved in microsound synthesis programming in the unit-generator languages; this argument led to the design and the development of LCSynth, a strongly-timed sound synthesis language that integrates objects and manipulation for microsounds, based on the assumption that the removal of such a conceptual gap between the software frameworks and the user's conceptualization can result in better language design that can facilitate microsound synthesis programming.

In this paper, we first describe a novel programming model for microsound synthesis in LCSynth, by giving concrete examples for two microsound synthesis techniques (granular synthesis and waveset synthesis). Then, we discuss how such a design concept of LCSynth can facilitate creative explorations by computer musicians in microsound synthesis and why the traditional abstraction of the unit-generator concept may not be truly appropriate for this problem domain; the features provided by unit-generators, the encapsulation of signal processing algorithms and the autonomous timing behavior without explicit scheduling, can turn into significant obstacles when exploring microsound synthesis techniques.

Such a discussion can be beneficial for both the further research in computer music software engineering and for the future development of more usable computer music programming languages.

2. RELATED WORK

2.1. Unit-Generator and Microsound Synthesis

2.1.1. Unit-generator

A unit-generator is *"a software module that emits audio or control signals (envelopes) or modifies these signals"* [13, p.787] and the concept first appeared in MUSIC-III language in 1960 [9]. In his book written in 1969, Mathews, the inventor of the unit-generator concept, describes that unit-generators perform "*conceptually similar functions to standard electronic equipment used for electronic sound synthesis"*[10, p.15]; thus, the original unit-generator concept is strongly associated with the electronic sound synthesis by electronic equipment.

Once the graph of unit-generators is built and the sound synthesis starts, the unit-generators autonomously compute the required amount of samples for sound output when necessary; users do not have to explicitly specify the timing when to compute sound output from the unit-generators. Only few computer music environments, such as Chuck [14], LCSynth [11], Marsyas [5], allow such explicit timing control.

2.1.2. *Microsound synthesis*

Dennis Gabor, a British physicist, in mid-1940s, first proposed the concept that originated the microsound synthesis techniques. Roads briefly describes the concept as *"in Gabor's conception, any sound can be decomposed into a family of functions obtained by time and frequency shift of a single Gaussian particle. Another way of saying this is that any sound can be decomposed into an appropriate combination of thousands of elementary grains*" [12, p.57].

While Gabor's theory is more strongly associated with time-frequency analysis rather than with computer music sound synthesis, microsound synthesis techniques we have today is derived from his concept that the entire sounds can be composed of short sound particles. Generally speaking, microsound synthesis techniques are realized by the overlap-add of such short sound particles with various waveforms, durations and intervals.

Yet, it should be noted here that this concept of microsound is brought to computer music practices much later than the emergence of the unit-generator concept (in 1960). For instance, one of the earliest experiments in microsound synthesis was realized by Roads on a mainframe computer in 1974 [12, p.302]; the unit-generator concept could never have taken microsound synthesis into consideration when it was invented.

2.1.3. *The conceptual gap between the unit-generator concept and microsound synthesis techniques*

A significant difference can be found between the concept of the unit-generator and one of microsound synthesis. As Mathews mentioned, the former is modeled after electronic sound synthesis and is quite similar the '*orthodox method'* that *"starts with the assumption that the signal s is a function s(t) of time t",* which Gabor contrasted to his new theory [7]. Such a concept significantly differs from the concept of microsound synthesis, in which the entire sound consists of many short sound particles that overlap-add.

In our previous publication [11], we discussed such a gap as a source of difficulty in programming microsound synthesis techniques as seen in the existing sound synthesis framework. By comparing the nine musical time-scales in computer music by Roads [12, p.3] and the representation within the unit-generator based sound synthesis framework, we assessed that the difficulty in programming microsound synthesis techniques is due to the lack of the counterpart entities to *microsound time-scale* in the existing sound synthesis frameworks; generally speaking, it is considered that such an abstraction in software design incompatible with the user's conceptualization can cause significant usability problems [2]. Blandford calls such a gap as *a conceptual misfit* in her framework for the analysis of usability problems called *CASSM (Concept-based Analysis of Surface and Structural Misfits)* [3].

2.2. LCSynth

Based on the assessment as above, we developed a new sound synthesis language: LCSynth. As we argued in [11], the design of LCSynth aims to remove a *conceptual misfit* between the user's conceptualization of microsound synthesis techniques and the traditional unit-generator languages, by directly integrating objects and manipulations for microsounds.

LCSynth is a strongly-timed programming language. As seen in *ChucK* [14], strongly-timed programming is a variation of synchronous programming and is based on logical synchronous time rather than the passage of real-time. A strongly-timed program explicitly advances a logical synchronous time and thus allows the precise timing behaviours in logical time, which is significantly desirable for computer music applications.

LCSynth also has traditional unit-generators. Figure 1 describes a simple sine wave oscillator instrument in LCSynth. In LCSynth, a sound synthesis module is defined as a *synth* object. The line 01-08 defines such a *synth* object, giving the name, *SinA*. In line 02-04, a unit-generator graph is defined inside the *ugens* block. In this example, the output from *~Sin* (a sine wave oscillator) is connected to the input of *~DAC* (sound output). The *~Sin* object is given a name of *sin* so that it can be referred in the other part of the code. In line 05-07, the *synmain* function for this *synth* object is defined. This function is immediately called when an instance of this synth object start playing. In this example, the *synmain* function has one argument *freq* with its default value 440. This value is used to set the frequency of *sin* by calling 'setFreq' method of ~Sin (line 06).

```
01:synth SinA {
02:  ugens {
03:    sin:~Sin() -> ~DAC();
04:  }
05:  synmain(freq = 440){
06:    sin.setFreq(freq);
07:  }
08:}
```

Figure 1. A simple sine wave instrument example in LCSynth

As previously described, LCSynth integrates objects and manipulations for microsound synthesis. A *Samples* object in LCSynth is a counterpart entity for a short sound particle in the concept of microsound synthesis and contains arbitrary number of samples within. A *Samples* object is immutable and the samples within a *Samples* object cannot be changed after its creation. Instead, LCSynth also provides *SampleBuffer* objects as a mutable object. A *SampleBuffer* can be converted to *Samples*[1] and vice versa.

[1] The relationship between *Samples* and *SampleBuffer* in LCSynth is similar to the relationship between *String* and *StringBuffer* in Java.

```
01:synth SinA {
02:  synmain(freq = 440){
03:    //create a sine wave table.
04:    var buf = new SampleBuffer(512);
05:    for (var i = 0; i < buf.size; i +=1){
06:      buf[i] = Sin(6.28318530718 / buf.size * i);
07:    }
08:    //convert it to a Samples object.
09:    var tmp = buf.toSamples();
10:    //resample to the given frequency.
11:    var sin = tmp.resample(44100.0 / freq);
12:
13:    while(true){
14:      //write the samples directly to DAC.
15:      WriteDAC(sin);
16:      //wait until the next DAC output timing.
17:      now += (sin.size)::samp;
18:    }
19:  }
20:}
```

Figure 2. An example code for a simple sine wave instrument without unit-generators in LCSynth

Figure 2 above is another version of a simple sine wave oscillator instrument in LCSynth. Instead of unit-generators, it involves *Samples* and *SampleBuffer* objects in sound synthesis, together with the strongly-timed programming feature of LCSynth. In line 04-07, a sine wave table of 512 samples is created, using a *SampleBuffer* object. This table is converted to *Samples* object (line 09) and then resampled to fit its size to the given frequency[2] (line 11). In the main loop (line 13-18), the samples within *sin* is first written out to DAC output by *WirteDAC()* function (line 16) and then the program sleeps until the next output timing[3] (line 17). Since LCSynth is strongly-timed, this sleep time is in logical synchronous time and thus it assures sample-rate accuracy; thus, the program behaves as a sine wave oscillator instrument.

Additionally, LCSynth also provides the means to stream the samples within *Samples* objects to the unit-generators. Figure 3 briefly illustrates such an example. A *~Bridge* unit-generator has its own internal buffer, which can be used to write the samples from *Samples* object. *~Bridge* object streams the written samples to the other unit-generator. In line 20, the samples are written directly into the internal buffer of *brg*, an instance of ~Bridge() unit-generator, by *write()* method and then immediately routed to ~ADSR object. This *write()* method overlap-add the written samples as in *WriteDAC()* function.

We will not particularly emphasize this feature in the later sections, as our focus is on the programming model for microsound synthesis in LCSynth. Yet, such a collaboration between the unit-generators and any synthesis algorithm is always possible via a *~Bridge* object.

[2] *Since Samples is a immutable object, it returns a new Samples object, which is resampled to the given size.*

[3] *'now' is a special variable that stands for the current system logical time. A thread can sleep by adding a duration value (such as 5::samp or 10::second) to 'now' as seen in Figure 2, line 17.*

```
01:synth SinB {
02:  ugens {
03:    brg:~Bridge () -> env:~ADSR() -> ~DAC();
04:  }
05:  synmain(freq = 440, dur){
05:    //create a sine wave table.
06:    var buf = new SampleBuffer(1024);
07:    for (var i = 0; i < buf.size; i +=1){
08:      buf[i] = Sin(6.28318530718 / buf.size * i);
09:    }
10:    var tmp = buf.toSamples();
11:    var sin = tmp.resample(44100.0/freq);
12:
13:    //trigger the ADSR envelope.
14:    env.keyOn();
15:    //calculate the trigger-off and stop timing.
16:    var keyOffTiming = now + 1::second;
17:    var stopTime     = now + dur::second;
18:
19:    while(now < stopTime){
20:      brg.write(sin);
21:      //calculate the next DAC output timing.
22:      var nextTiming = now + (sin.size)::samp;
23:      //check if we reach the key off timing.
24:      if (nextTiming >= keyOffTiming){
25:        //wait until the key off timing.
26:        now = keyOffTiming;
27:        //release the envelope.
28:        env.keyOff();
29:        //not to call keyOff() again.
30:        keyOffTiming = stopTime;
31:      }
32:      //wait until the next DAC output timing.
33:      now = nextTiming;
34:    }
35:  }
36:}
```

Figure 3. An example of the combination of unit-generators and *Samples* objects in LCSynth

2.3. LCSound

```
01:orchestra {
03:  //create a sine wave table.
04:  var buf = new SampleBuffer(512);
05:  for (var i = 0; i < buf.size; i +=1){
06:    buf[i] = Sin(6.28318530718 / buf.size * i);
07:  }
08:  //set the sine wave table to a global variable.
09:  global gSin = buf.toSamples();
10:
11:  synth SinA {
12:    synmain(freq = 440){
13:      //below is to access a global variable.
14:      global gSin;
15:      //resample to the given frequency.
16:      var sin = gSin.resample(44100.0 / freq);
17:      while(true){
18:        //write the samples directly to DAC.
19:        WriteDAC(sin);
20:        //wait until the next output timing.
21:        now += (sin.size)::samp;
22:      }
23:    }
24:  }
25:}
26:score
27:{
28:  0.0:  SinA()             for 8.00,
29:        SinA(freq:880)     for 4.00;
30:  2.0:  SinA(freq:1760)    for 5.00;
31:}
```

Figure 4. An example code of LCSound

LCSynth itself is a synthesis language, which focuses on the descrpitions of sound synthesis algorithms. LCSound is a simple score/instrument type of computer music language that encloses LCSynth within, which is developed as a test-bed for LCSynth.

We briefly describe an example in LCSound in Figure 4. The file is separated into two sections, *orchestra* and *score*. The *orchestra* section (line01-10) is the section where the definitions of *synth* objects in LCSynth are described. As seen line 03-09, the *orchestra* section can also contain the code fragments. In this example, a sine wave table is created and set to a global variable *gSin* so that it can be shared by the instances of *SinA*, which is defined between line 11-23.

The defined *synth* objects can be used in *Score* section (line 26-31). First, the start time in second is placed before ':' and then the name of *synth* object follows with the argument(s) given to its *synmain* function. As seen in Figure 4, LCSound supports the named parameters and the default value. The duration (in second) to play the synth object can be specified after '*for*'. The notes that shares the same start time can be juxtaposed after ',' (line 28-29). As above, LCSound is a typical score/instrument language, which is similar to CSound. The current version of LCSound can run both in real-time and in non real-time.

3. PROGRAMMING MICROSOUND SYNTHESIS IN LCSYNTH

3.1. Granular synthesis

3.1.1. Synchronous Granular Synthesis

Synchronous granular synthesis is a kind of granular synthesis, in which "*sounds results from one or more streams of grains. Within each stream, one grain follows another, with a delay period between the grains. Synchronous means that the grains follow each other at regular intervals"* [12, p.93].

Figure 5 below describes an example code of synchronous granular synthesis in LCSynth. In this example, we use just one cycle of sine wave as a grain and set it to a wave table (line 01- 05). The *synmain* function of this example takes 3 arguments, *dur* for the duration of the entire sound, *interval* for the regular intervals between grains and gsize for the size of the grain in samples. Line 11 generates a grain by resampling the wave table. In the following main loop (line 12-15), first a single grain is written out for DAC output. Then the program sleeps for the interval (with sample-rate accuracy), and the next grain is then written out for DAC output. The overlap-add is performed automatically and thus synchronous granular synthesis can be performed by such a simple code.

```
01:var tmp = new SampleBuffer(256);
02:for (var i = 0; i < tmp.size; i +=1){
03:  tmp[i] = Sin(6.28318530718 / tmp.size * i);
04:}
05:global gTable = tmp.toSamples();
06:
07:synth SyncGran {
08:  synmain(dur, interval, gsize){
09:    global gTable;
10:    var stopTime = now + dur::second;
11:    var grain = gTable.resample(gsize);
12:    while(now < stopTime){
13:      WriteDAC(grain);
14:      now += interval::samp;
15:    }
16:  }
17:}
```

Figure 5. A synchronous granular synthesis example in LCSynth

3.1.2. Quasi-synchronous granular synthesis

Quasi-synchronous granular synthesis is another variation of granular synthesis, in which "*the grains follow each other at unequal intervals, where a random deviation factor determines the irregularity"* [12, p.93].

Figure 6 below is an example of quasi-synchronous granular synthesis in LCSynth. As seen in this example, the modification from the previous example of synchronous granular synthesis is quite simple; only line 08 is modified so that the intervals can be randomized between the values of *interval1* to *interval2*.

```
01:synth QuasiSyncGran {
02:  synmain(dur, interval1, interval2, gsize){
03:    global gTable;
04:    var stopTime = now + dur::second;
05:    var grain = gTable.resample(gsize);
06:    while(now < stopTime){
07:      WriteDAC(grain);
08:      now += Rand(interval1, interval2)::samp;
09:    }
10:  }
11:}
```

Figure 6. A quasi-synchronous granular synthesis example in LCSynth

3.1.3. Time-stretching and Pitch-shifting

While there exist phase-vocoding techniques for time-stretching and pitch-shifting, microsound synthesis techniques called *granular sampling* [8] are also known as simple and computationally efficient techniques for time-stretching and pitch-shifting.

```
01:LoadSndFile(0, "test.aif");
02:
03:synth TimeStrech {
04:  synmain(ratio, dur){
05:    var stopTime = now + dur::second;
06:    var pos = 0::second;
07:    var window = GenWindow(100::ms, \hanning);
08:    while(now < stopTime){
09:      var read = ReadBuf(bufno :0, dur: 100::ms,
10:                         offset:pos);
11:      var grain = read.applyEnv(window);
12:      WriteDAC(grain);
13:      now += 50::ms;
14:      pos += 50::ms / ratio;
15:    }
16:  }
17:}
```

Figure 7. A time-stretching example in LCSynth

Figure 7 above is a simple granular sampling example for time-stretch without pitch change. In the following explanation, assume a *ratio* parameter of two is assumed (line 04) to time-stretch the samples to twice the length of the original. The main loop (line 08-16) first read 100ms of samples from the buffer and applies a hanning window to create a single grain and then write it out to DAC. Then, the program sleeps for 50ms before the next grain (line 13)[4]. Yet, the reading position is advanced to the next position, but only by 50ms / 2 = 25ms (line 14). As the reading position advances only at half the speed of the advance of the logical time, the sound can be time-stretched to twice as long as the original sound.

On the contrary, pitch-shifting without changing of the duration of the sound is achieved by resampling of the original sound data by the unit of grains. Figure 8 shows

[4] *As the duration of grain is 100ms, this causes grains to overlap-add with the interval of every 50ms*

a simple example of this technique. This time, we change the number of samples to read from the buffer and then resample it to the desired grain size so that we can shift the pitch of the original sound; for instance, if 882 samples of the sound data is read as a grain and then resample to 441 samples, the pitch of the grain is shifted an octave higher than the original in this example, since the sampling rate stays the same.

```
synth PitchShift {
  synmain(pitch, dur){
    var stopTime = now + dur::second;
    var pos = 0::second;
    var window = GenWindow(441::samp);
    while(now < stopTime){
      var size = 441:: samp * pitch;
      var read = ReadBuf(bufno :0  , dur: size,
                           offset:pos);
      read = read.resample(read.size / pitch);
      var grain = read.applyEnv(window);
      WriteDAC(grain);
      now += 50::ms;
      pos += 50::ms;
    }
  }
}
```

Figure 8. A pitch-shifting example in LCSynth

3.2. Waveset Synthesis

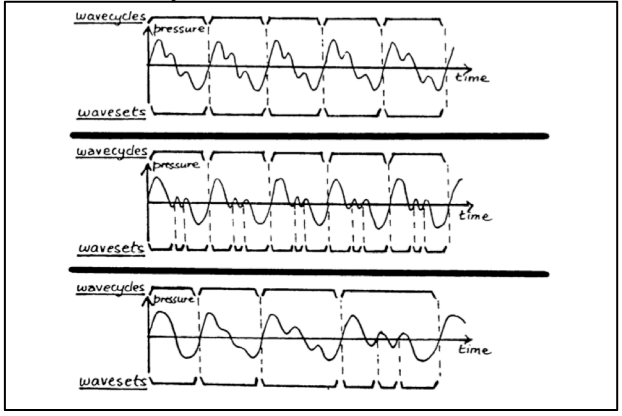

Figure 9. Wavecycles and wavesets (taken from [17 , p.50])

Waveset synthesis techniques also belong to the family of microsound synthesis techniques. Waveset synthesis techniques are realized as various kinds of manipulations of short sound particles called *wavesets*. According to Wishart, a waveset is defined as a *"distance from zero-crossing to a 3rd zero-crossing"* whereas *wavecycle* is defined as "*wavelength of sound, where clearly pitched"* [17, p.50]. Figure 9 below taken from [17, p.50] briefly illustrates *wavesets* and *wavecycles*. As seen in this figure, a wavecycle can contain more than one wavesets, while the waveform and the length of a waveset can vary significantly. While there are a number of different waveset synthesis techniques[5], we take 4 of these for the following implementation examples in LCSynth.

3.2.1. *Waveset inversion*

Waveset inversion is a technique that inverts a waveset in time domain as seen in Figure 10 (right), and "*usually produces an "edge" to the spectral characteristics of the sound"* [16, p.42]. Figure 11 below describes an example of waveset inversion in LCSynth. First, a sound file is loaded onto the buffer no.0 by *LoadSndFile()* function and then the wavesets are obtained from the buffer by *ExtractWaveSets()* function. This function returns an array of the wavesets extracted from the buffer.

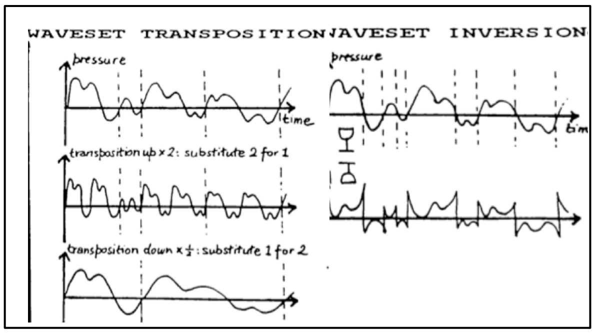
WAVESET TRANSPOSITION
WAVESET INVERSION
pressure
time
transposition up x 2: substitute 2 for 1
transposition down x ½: substitute 1 for 2
pressure
time

Figure 10. the pictorial representations of waveset transposition (left) and waveset inversion (right) [17, p.50]

While it is possible to implement the algorithm to invert a waveset by accessing each sample within a *Samples* object and by using a *SampleBuffer* to generate an inverted waveset, *Samples* has *invertWS()*, a method that returns the inverted waveset as seen in Figure 10. Since *Samples* is immutable, *invertWS()* returns a new instance of *Samples*. In line 07-08, *WriteDAC()* method is used to output an inverted waveset to DAC and then the program sleeps for exactly the size of the samples in the logical synchronous time. This procedure is repeated until all the wavesets are inverted and output.

```
01:LoadSndFile(0, test.aif");
02:global wavesets = ExtractWaveSets(0);
01:synth WSInvert {
02:  synmain(){
03:    global wavesets;
04:    for (var i = 0;i < wavesets.size; i += 1){
05:      var orig     = wavesets[i];
06:      var inverted = orig.invertWS();
07:      WriteDAC(inverted);
08:      now += inverted.size::samp;
09:    }
10:  }
11:}
```

Figure 11. A waveset inversion example in LCSynth

3.2.2. *Waveset transposition*

Waveset transposition is a technique that "*substitutes N copies of a waveset in the place of M wavesets, for example 2 in the space of 1 or 1 in the space of 4, for doubling and quartering of frequency respectively"*. [17, p.50]. Figure 10 (left) illustrates this waveset synthesis technique.

Figure 12 in the next page shows two examples of waveset transposition, one that substitutes 2 wavesets for 1 (octave up) and one that substitutes 1 waveset for 2 (octave down) as seen Figure 10(left). To change the size of a waveset, the *resample()* method can be used as in line 09 and 23. For waveset transposition to substitute 2 wavesets for 1, it is enough to output the waveset resampled to the half size of the original twice as seen in line 10-13 and for waveset transposition to substitute 1

[5] For instance, 17 different techniques are available in Wishart's Composer Desktop Project software in [11, p207].

for 2, the size of a waveset is doubled by resampling and then skip the next one after DAC output (line 23-26).

```
01:LoadSndFile(0, "test.aif");
02:global wavesets = ExtractWaveSets(0);
03://waveset transposition (octave up)
04:synth WSTransA {
05:  synmain(){
06:    global wavesets;
07:    for (var i = 0; i < wavesets.size; i += 1){
08:      var orig = wavesets[i];
09:      var octup = orig.resample(orig.size / 2);
10:      WriteDAC(octup);
11:      now += octup.size::samp;
12:      WriteDAC(octup);
13:      now += octup.size::samp;
14:    }
15:  }
16:}
17://waveset transposition (octave down)
18:synth WSTransB {
19:  synmain(){
20:    global wavesets;
21:    for (var i = 0;i < wavesets.size - 1; i += 2){
22:      var orig = wavesets[i];
23:      var octdown = orig.resample(orig.size * 2);
24:      WriteDAC(octdown);
25:      now += octdown.size::samp;
26:    }
27:  }
28:}
```

Figure 12. A waveset transposition example in LCSynth

3.2.3. *Waveset substitution*

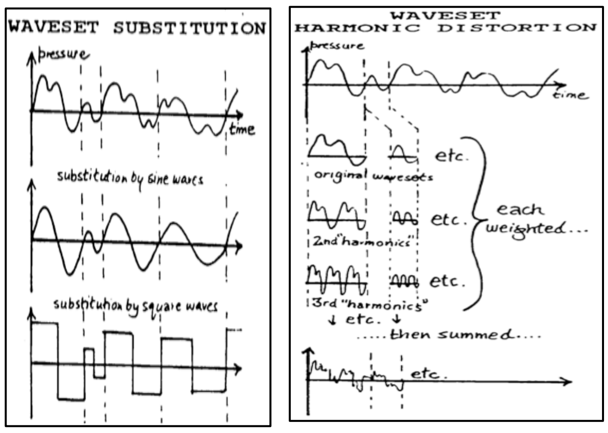

Figure 13. waveset substation and waveset harmonic distortions (taken from [17, p.50])

Waveset substitution *"replaces wavesets by a stipulated waveform of the sample amplitude, frequency and time span as the original waveset*"[12, p.207]. Figure 13 (left) illustrates the waveset synthesis technique pictorially. In this example, the original wavesets are substituted with sine waves and square waves. The code for waveset substitution in LCSynth is shown in Figure 14. Suppose wavesets are already extracted from a buffer and set to a global variable *wavesets* before this code. First, a wave table of square wave is generated in line 01-07. This wave table is set to a global variable *square* in line 08. In the main loop (line 14-20), a waveset is taken from the array of wavesets one by one. Then, a square wave is resampled to the same size as the original waveset and amplified to the same amplitude (line 16-17). This new waveset is used for substitution and written to DAC output, followed by sleeping until the timing to schedule the next waveset (line 18-19); thus, a waveset substitution is performed.

```
01:var sqbuf = new SampleBuffer(256);
02:for(var i = 0; i < sqbuf.size / 2; i += 1){
03:  sqbuf[i] = 1.0;
04:}
05:for(var i = sqbuf.size/2; i < sqbuf.size; i += 1){
06:  sqbuf[i] = -1.0;
07:}
08:global square = sqbuf.toSamples();
09:
10:synth WSSubst {
11:  synmain(){
12:    global wavesets;
13:    global square;
14:    for(var i = 0;i < wavesets.size; i += 1){
15:      var orig = wavesets[i];
16:      var tmp  = square.resample(orig.size);
17:      var out  = tmp.amplify(orig.maxAmp());
18:      WriteDAC(out);
19:      now += out.size::samp;
20:    }
21:  }
22:}
```

Figure 14. a waveset substitution example in LCSynth

3.2.4. *Waveset harmonic distortion*

```
01:synth WSHarmDist {
02:  synmain(weight1, weight2, wieght3){
03:    global wavesets;
04:    for(var i = 0;i < wavesets.size; i += 1){
05:      //the original, 2nd and 3rd harmonics.
06:      var orig = wavesets[i];
07:      var oct1 = orig.resample(orig.size / 2);
08:      var oct2 = orig.resample(orig.size / 3);
09:      //weight each of them.
10:      orig = orig.amplify(weight1);
11:      oct1 = oct1.amplify(weight2);
12:      oct2 = oct2.amplify(wieght3);
13:      //write out the original.
14:      WriteDAC(orig);
15:      //write out the 2nd harmonics.
16:      WriteDAC(oct1);
17:      WriteDAC(oct1, offset: oct1.size::samp);
18:      //write out the 3rd harmonics.
19:      WriteDAC(oct2);
20:      WriteDAC(oct2, offset: oct2.size::samp);
21:      WriteDAC(oct2, offset: oct2.size::samp * 2);
22:      //wait until the next scheduling timing.
23:      now += orig.size::samp;
24:    }
25:  }
26:}
```

Figure 15. A waveset harmonic distortion example in LCSynth

Waveset harmonic distortion *"superimposes N harmonics on the waveset fundamental with a scaling M relative to the previous harmonic"* [12, p.207]. Figure 13 (right) illustrates this technique pictorially.

We describe an example of waveset harmonic distortion in LCSynth in Figure 15. In line 06-08, wavesets for the 2nd and 3rd harmonics are generated from the original by resampling. These three wavesets are then weighted by the given arguments (line 10-12). First, the original waveset is written out to DAC (line 14) and the waveset of 2nd harmonics is written out exactly at the same timing. Then, the 2nd harmonics is written again, but this time it is given an offset of its own sample size (line17). The results in this waveset is written at the timing right after the 1st one. The same procedure is applied to the 3rd harmonics (line 19-21).

In line 23, the program wait until the next timing schedule by sleeping for the duration of the original wavesets; thus waveset synthesis can be realized.

4. DISCUSSION

4.1. Unit-Generators Considered Harmful for Microsound Synthesis

In a sound synthesis framework built upon the unit-generator concept, DSP algorithms are encapsulated inside the unit-generators with common interfaces that allow the composition of the unit-generators to describe more complex sound synthesis algorithms. Many sound synthesis frameworks, except a very few, provide autonomous behaviours for the timing of when the computations of sound output must be performed, so that users do not have to explicitly schedule the timing.

While these features are quite beneficial for many sound synthesis algorithms, such an abstraction also revokes the access to the details required for the experiments in the level of microsound time-scale from users. The samples that are routed between unit-generators are normally hidden or hardly accessible from the users and no direct counterpart entities for short sound particles, e.g. grains or wavesets, are provided. Thus, in such a framework, even a very simple microsound synthesis technique may involve a certain degree of complexity, regardless of its conceptual simplicity.

For instance, it is very common in many granular synthesis techniques to apply an envelope made of some window function to a fragment of samples taken from some sound data and to overlap-add them to constitute the processed sound. Such kind of task can also be seen in the time-stretching example (Figure 7) and the pitch-shifting example (Figure 8). Yet, to perform this simple task only within the unit-generator-based synthesis framework, each single grain needs to be modeled as a note-level object and overlap-add of grains is realized by simultaneously playing such note-level objects. Roads describes such a typical strategy in [12, p.91].

While the lack of precise timing control is due to the implementation rather than the unit-generator concept itself, autonomous behavior in sample computation timing makes it harder to schedule grains with precise timing and results in inaccurate sound output in many existing synthesis framework. Such an inaccuracy in timing can be clearly audible to human perception; as such frameworks revoke the access to the explicit timing control, it can often involve cumbersome programming patterns to compensate imprecise timing behavior. For instance in SuperCollider [15], it frequently involves a programming pattern to schedule a synth object ahead of time, with offset to the actual timing and to use *OffsetOut* unit-generator for better accuracy in microsound synthesis.

Furthermore, even though it is also possible to enclose the whole algorithm of a microsound synthesis technique within a unit-generator to achieve accurate sound output, such an encapsulation makes it impossible to modify or extend the encapsulated algorithms by users for further creative experiments.

As above, while the encapsulation of DSP algorithms and the autonomous behaviour are important features of the unit-generator-based sound synthesis frameworks, such features turn into significant obstacles when users want to explore in the domain of microsounds; the entities of microsounds and the synthesis algorithms are hidden from inside within the unit-generators by encapsulation. Autonomous behaviour in computation timing also makes it hard to control the behaviour of the unit-generators with sample-rate accuracy. However, users need to access these hidden details to modify and to extend the synthesis algorithm of microsound synthesis techniques for creative exploration; thus, the difficulty in implementing microsound synthesis techniques in the unit-generator based sound synthesis frameworks is indeed caused by the features of the unit-generators themselves, even though hiding such details is beneficial for many other sound synthesis techniques.

4.2. The Benefits of LCSynth's Language Design and Its Programming Model for Microsound Synthesis

On the contrary, the design of LCSynth is designed to avoid hiding such details and makes them accessible to users. It provides the entities of microsounds as *Samples* object and *SampleBuffer* object with many related functions that manipulate these objects. At the same time, strongly-timed programming supports the precise timing behavior required for accurate sound rendering results. LCSynth also takes it into consideration in its design to automatically perform overlap-add operation to *Samples* objects and to schedule them in the future timings.

As programmers "*use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping*"[6, p22] in programming activity, it is quite beneficial to provide objects and functions in a computer music language, which can be considered as the direct counterparts for the entities in the user's conceptualization of microsound synthesis; mapping between the program code (the computing domain) and the conceptualization of synthesis algorithm (the application domain) can be made a lot easier without involving incompatible mapping.

Moreover, since LCSynth doesn't have to encapsulate microsound synthesis algorithms as seen in unit-generators, the modification and the extension of sound synthesis algorithm beyond what existing programs offer can be significantly facilitated, by providing more opportunities for further creative exploration.

For instance, as seen in the examples of granular synthesis, to modify synchronous granular synthesis algorithm to quasi-synchronous granular synthesis algorithm can be easily achieved just by using *Rand()* function and such a strategy can easily be reused if a user want to randomize the pitch and the intervals in waveset synthesis. In another instance, to extend some granular synthesis algorithm so that it does harmonizes the sounds of octave upper, the programming model used in *waveset harmonic distortions* can be applied in the almost same form.

Thus, LCSynth's programming model for microsound synthesis allows users to easily reuse *programming schema* [6, p.23] in different synthesis techniques, once users have obtained such programming schema. This benefit for creative exploration in the level of microsound time-scale can be difficult to achieve with unit-generators, as this type of explorations involve the manipulations of the entities that are normally abstracted away inside unit-generators.

As discussed above, while it is still equipped the unit-generators, LCSynth also contains significantly different abstraction in its sound synthesis framework, which allows a novel programming model that facilitate the experiments in the level of microsound time-scale.

4.3. Two Different Abstractions

We described the difference between LCSynth's abstraction for microsound and that of the unit-generator concept. As seen in Figure 2, which uses *Samples* objects for wavetable synthesis, it is likely to possible to implement other synthesis techniques other than microsound synthesis techniques by introducing more objects and functions into LCSynth. However, it may still be beneficial to use the unit-generators in many cases, since each of these two abstractions deal with different problem domains.

The unit-generator concept aims to let each unit-generator enclose a simple signal processing algorithm and work as a basic building block, allowing constitution of more complex synthesis algorithms by interconnecting unit-generators. On the contrary, LCSynth's abstraction is rather focus on providing objects and manipulations as building blocks to compose microsound synthesis algorithms.

From such a perspective, the abstraction of the unit-generator concept and that of microsounds in LCSynth deal with different levels of sound synthesis, and it would be fair to provide some features to let these two abstractions collaborate with each other as seen in the example in Figure 3; it is beneficial for users to provide a framework design in which these two different abstractions can coexist.

5. CONCLUSION

We described a novel programming model for microsound synthesis in LCSynth with concrete examples of granular synthesis and waveset synthesis. Instead of encapsulating DSP algorithms within unit-generators, LCSynth provides objects and manipulations for microsound synthesis. We also discussed the benefit of this programming model and why the traditional unit-generator concept may not be truly appropriate for the problem domain of microsound synthesis.

Such a novel programming model and the detailed discussion on the difference between LCSynth's programming model and the unit-generator concept can be beneficial for the further development of new sound synthesis frameworks and computer music languages.

6. REFERENCES

[1] Bencina, R. Implementing Real-Time Granular Synthesis. *In Audio Anecdotes III*, A.K Peters, 2006.

[2] Blackwell, A.F et al. The Abstraction is 'an Enemy': Alternative Perspectives to Computational Thinking. *Proc. PPIG08*, 2008.

[3] Blandford, A. et al. Evaluating System Utility and Conceptual Fit Using CASSM. *Intl. Journal of Human-Computer Studies Vol.66*, pp.393-400, 2008.

[4] Brandt, E. *Temporal Type Constructors for Computer Music Programming*. Ph.D Thesis, Carnegie Melon University, 2002.

[5] Burroughs, N. et al. Flexible Scheduling for DataFlow Audio Processing". *Proc. ICMC'05*, 2005.

[6] Détienne, F. *Software Design - Cognitive Aspects*. Springer Verlag, 2001.

[7] Gabor, D. Lectures on Communication Theory, *Technical Report 238, Research Laboratory of Electronics*. Massachusetts Institution of Technology, 1952.

[8] Lippe, C. Real-time granular sampling using the ircam signal processing workstation. *Contemporary Music Review, Vol.10,* 1994.

[9] Mathews, M. V. An acoustic compiler for music and psychological stimuli. *Bell System Technical Journal, Vol.40*, 1961

[10] Matthews, M.V. et al. *The Technology of Computer Music*. The MIT Press, 1969.

[11] Nishino, H and Osaka, N. LCSynth: A Strongly-Timed Synthesis Language that Integrates Objects and Manipulations for Microsounds", *Proc. Sound and Music Computing*, 2012 .

[12] Roads, C. *Microsound*, The MIT Press, 2004.

[13] Roads, C. *The Computer Music Tutorial*, The MIT Press, 1996.

[14] Wang, G. *The Chuck Audio Programming Language: A Strongly-Timed And On-the-Fly Environ/Mentality*, Ph.D Thesis, Princeton University, 2008.

[15] Wilson, Scott et al. *The SuperCollider Book*. The MIT Press, 2011.

[16] Wishart, T. *Audible Design*. Orpheus Books LTD. 1994.

[17] Wishart, T. *Audible Design*: *Appendix 2,* Orpheus Books LTD. 1994.