LCSynth: A Strongly-Timed Synthesis Language that Integrates Objects and Manipulations for Microsounds

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

Naotoshi OSAKA
Department of Information
Systems and Multimedia Design
Tokyo Denki University
osaka@im.dendai.ac.jp

ABSTRACT

In this paper, we describe LCSynth, a new sound synthesis language currently under development, which integrates objects and manipulation for microsounds in its language design. Such an integration of objects and manipulations for microsounds into sound synthesis framework can facilitate creative exploration in microsound synthesis techniques, which have been considered relatively difficult in the existing sound synthesis frameworks and computer music languages that depend solely on traditional abstraction of unit-generators.

1. INTRODUCTION

The concept of the unit-generator was developed in very early phase of computer music history and has played a significant role in computer music sound synthesis. Even long after its birth, this unit-generator concept is still frequently seen as a core abstraction for sound synthesis in many recent computer music languages.

While the unit-generator concept is considered as a suitable abstraction for many sound synthesis techniques, some researchers and practitioners in computer music have been arguing the difficulty in implementing microsound synthesis techniques [1, 7, 13, 22]; today microsound synthesis techniques are often implemented by modeling each microsound as a note as described in [18, p.91] or enclosed inside a unit-generator as Geiger discuses in [13]. However, as described in the following section, these approaches may involve some problems in implementation and also can significantly limit the exploration in related synthesis techniques.

It is our position that there is a problem in microsound synthesis techniques rooted in the abstraction applied to sound synthesis framework design based solely on the unit-generators. Microsound synthesis techniques are brought into computer music practices long after the invention of the unit-generator concept and hence the unit-generator concept itself could never consider the requirements for microsound synthesis techniques in its birth; It is necessary to reconsider the abstraction of sound synthesis frameworks so to facilitate the implementation of microsound synthesis techniques.

In this paper, we describe LCSynth, a new sound synthesis programming language that integrates objects and manipulations for microsounds. LCSynth is designed as *a strongly-timed language* [22] so to precisely perform microsound synthesis techniques. The collaboration between the unit-generators and microsound objects is also considered as a part of its design. Such an integration of microsound entities and related manipulations provide simpler programming model for the implementation of microsound synthesis techniques and can be beneficial for creative exploration in computer music creation.

2. RELATED WORK

2.1 The Unit-Generator Concept

A unit-generator is *"a software module that emits audio or control signals (envelopes) or modifies these signals"* [17, pp.1234] and is considered to perform *"conceptually similar functions to standard electronic equipment used for electronic sound synthesis"*[13]. The concept was first introduced into computer music by Music-III developed in 1960 [17, p787]. Since then, the unit-generator concept has served as a core abstraction in computer music. Even those computer music languages that are widely used today, such as Max/MSP [26], PureData [16], SuperCollider [24] and Chuck [22], all depend on the unit-generator concept.

2.2 Microsound Synthesis

Dennis Gabor, a British physicist, around mid-1940s, first proposed this concept of microsound. Roads briefly summarizes Gabor's concept of microsound as following: *"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"* [18, p.57].

Many different sound synthesis techniques that involve such short sound particles (or *microsounds*) have been developed since, (not limited to the use of Gaussian particles as in Gabor's original concept), such as granular synthesis [19], FOF synthesis [20], FOG synthesis [8] and waveset synthesis [25]. Generally speaking, in practice, microsound synthesis involves *overlap-add* of microsounds with various waveforms and durations.

For instance, *waveset synthesis* manipulates sound by the unit of *waveset*, a short sound particle defined as "*distance from a zero-crossing to a 3^{rd} zero-crossing"* [25, p.50]; e.g. Waveset transposition is a technique which "*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"*. Waveset shuffling *"permutes collections of wavesets. A simple shuffle of successive wavesets starting with (a, b, c, d) becomes (b,a,d,c); shuffling by pairs of wavesets makes (d, c, a, b).* [18, p.207]. Figure 1 (taken from [25, p.50]) below shows the pictorial representations of these two techniques.

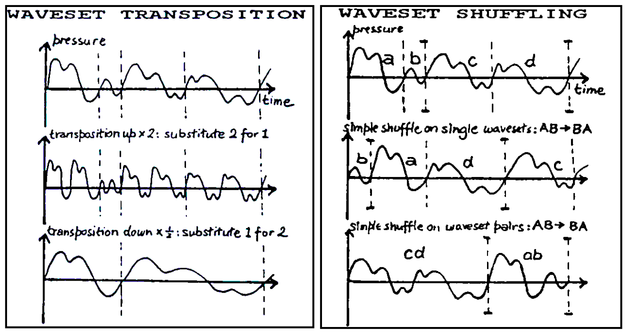
WAVESET TRANSPOSITION
pressure
time
transposition up x 2: substitute 2 for 1
transposition down x ½: substitute 1 for 2
WAVESET SHUFFLING
pressure
a
b
c
d
time
simple shuffle on single wavesets: AB→BA
b
a
d
c
simple shuffle on waveset pairs: AB→BA
cd
ab

Figure 1. The pictorial representations of two waveset synthesis techniques: waveset transposition (left) and waveset shuffling (right) [25, p.50]

As another example, granular synthesis techniques involve short sound particles (*grains*) such that "*a single grain serves as a building block for sound objects"* [18, p.87]. Generally speaking, granular synthesis depends on '*overlap-add'* of such grains[1]; the sum of the grains that overlap each other reconstitutes the entire output signal. Even when all the waveforms of the grains are identical, the overlap-add of the grains can result in different waveforms depending on how they overlap each other.

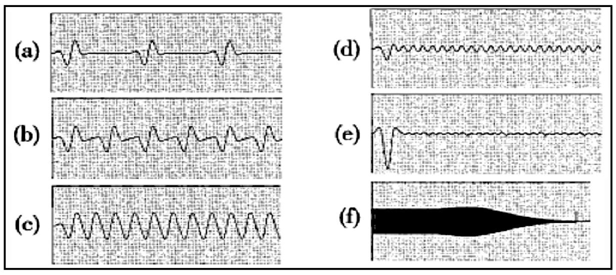

Figure 2. Influence of grain density on pitch (a) 50 grains/sec (b) 100 grains/sec (c) 200 grains/sec (d) 400 grains/sec (e) 500 grains/sec (f) Plot of a granular stream sweeping from the infrasonic frequency of 10 grains/sec to the audio frequency of 500 grains/sec over 30 seconds (taken from [18, p.95]).

[1]Some waveset synthesis techniques can also involve such 'overlap-add'. In, waveset harmonic distortions, each waveset can be overlapped and added by the other wavesets of various different pitches, which derive from the original waveset.

Figure 2 shows how the density of grains can influence the resulting pitch in *synchronous granular synthesis*, in which *"the grains follow each other at regular intervals"* [18, p.93]. As seen in Figure 2, even when all the grains are totally identical, a change in the interval between grains results in a different pitch.

As above, microsound synthesis techniques can involve *overlap-add* of microsounds with various waveforms and durations to constitute the entire output signal. The intervals between microsounds can be both periodic and aperiodic and each microsound must be scheduled with sample rate accuracy; otherwise, the resulting signal output would differ from what is expected. Such a characteristic of microsound synthesis differs considerably from "*the orthodox method of analysis*", which *" starts with the assumption that the signal s is a function s(t) of time t."* [12], before Gabror's discovery of microsounds; this orthodox method may correspond more to the unit-generator concept, which is modeled after analog sound synthesis by electronic equipment.

2.3 Implementing Microsound Synthesis Techniques

In this subsection, we briefly describe two different approaches to implement microsound synthesis techniques in the unit-generator based computer music languages.

2.3.1 Developing a Unit-Generator Object

One of the most common approaches for microsound synthesis is to enclose microsound synthesis techniques in unit-generator objects. Many computer music languages provide some framework to write a unit-generator as a external library written in general purpose programming languages such as C or C++, which can be compiled into native code. Instantiation and manipulations of microsounds are wrapped within such a unit-generator. For instance, a granular synthesis unit-generator may instantiate grains and overlap-add them just within itself and simply output the resulting signal.

2.3.2 Implementing microsound synthesis techniques solely in a computer music language

The other approach is to implement microsound synthesis techniques only within a computer music language. Visual Programming Languages (VPLs) such as Max/MSP or PureData are widely used in computer music community. While the developers of computer music VPLs normally provide ready-made unit-generators for some microsound synthesis techniques, implementing microsound synthesis techniques solely in such VPLs without built-in unit-generators can involve considerable difficulty. Richard Dudas, an American composer, invented one of the well-known techniques to implement granular synthesis in Max/MSP and other visual computer music languages alike. Instead of dynamic instantiation of sound objects, the implementation by Dudas uses the number of grain players that are serially connected.

Figure 3 is a screenshot of a part of his granular synthesis patch in Max/MSP. Each grain player sub-patch (*rgrain2~*) starts playing a single grain when it receives *bang* message. First, the *bang* message from the *metro* object is sent to the first grain player sub-patch. This grain player sub-patch immediately starts playing a grain with given parameters if it is not already playing a grain. If this grain player is already playing a grain, it simply delegates the *bang* message to the next grain players cascaded to it. By such delegation mechanism, Dudas implemented granular synthesis solely within Max/MSP.

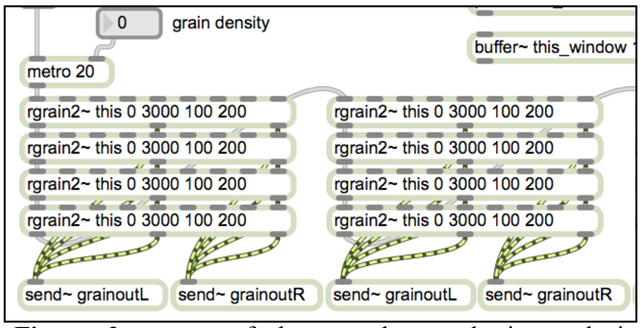

Figure 3. a part of the granular synthesis patch in Max/MSP by Richard Dudas (Courtesy of the Composer)

In those languages based on score/instrument abstractions as in CSound [5], it is common to model each microsound as a note-level sound object. Roads describe such an approach in [18, p.91]. In this approach, overlap-add of microsounds is processed as notes that are played with overlapping in their life times.

```
01:(
02:   //define an instrument(synth) for as a grain.
03:   SynthDef(\grain, {
04:     arg bufnum;
05:     var samp, env, out;
06:     samp      = PlayBuf.ar(1, bufnum, 1);
07:     env       = EnvGen .ar(Env.triangle(0.05,1),
08:                           doneAction:2    );
09:     out       = samp * env;
10:     //output to the channel no.0.
11:     Out.ar(0, out);
12:   }).add;
13:)
14://read a sound file onto the buffer (no.0).
15: Buffer.read(Server.local, "test.wav" ,bufnum:0);
16: //perform granular synthesis
17:(
18:  //create a function that plays grains.
19:  var func = {
20:    //generate grains with 25 msec interval.
21:    inf.do {
22:      //play one grain.
23:      Synth.new(\grain, [\bufnum, 0]);
24:      //wait 25 msec before the next grain
25:      (0.025).wait;
26:    };
27:  };
28:  //play the function above.
29:  func.fork;
30:)
```

Figure 4. An example of simple synchronous granular synthesis in SuperCollider

Figure 4 above is a simple synchronous granular synthesis program, which takes this approach to model. A single grain of 50msec duration is defined as a *synth* (a note-level object in SuperCollider) in line 03-12 and the loop between line 21 and 26 continuously plays this *synth* as a note with 25msec interval. However, due to the server-client architecture and timing inaccuracy in SuperCollider, this code can't render the precise synchronous granular synthesis[2]. As described in the previous section, such a lack of timing accuracy is a significant problem in implementing microsound synthesis techniques.

```
-----------------File:grain.ck-----------------
01:// the patch
02:SndBuf buf => Envelope env => dac;
03://load the file onto the buffer
04:"test.wav" => buf.read;
05:
06:fun void grain( int duration) {
07:  //set up the duration for the grain envelope.
08:  duration::ms => env.duration;
09:  0.5 => env.gain;
10:  //main loop for this grain.
11:  while (true){
12:    0 => buf.pos; //read from the head of 'buf'.
13:    //trigger the envelope.
14:    env.keyOn();
15:    (duration * 0.5)::ms => now;
16:    env.keyOff();
17:    (duration * 0.5)::ms => now;
18:  }
19:}
20://play a grain of 50ms that repeats over and over.
21:spork ~grain(50);
22://time loop
23:while (true){
24: 1::ms => now;
25:}
---------------File:playgrain.ck---------------
01://start the first shread that play grain.
02:Machine.add( "grain.ck" );
03://start the second one after 25 msec
04:25::ms => now;
05:Machine.add( "grain.ck" );
```

Figure 5. An example of simple synchronous granular synthesis in ChucK

Wang's ChucK, *a strongly-timed* programming language for computer music, provides an appropriate functionality for such an issue, by integrating *logical synchronous time* and its explicit control within a language design [22]. In ChucK, a program is allowed to be *"self-aware in the sense that it always knows its position in time and can control its own progress over time"*[3] [22, p. 41].

In ChucK, the major approach used in the community for the implementation of granular synthesis is to play grains by multiple *shreds* (non-preemptive threads in ChucK). Figure 5 shows a simplified version of the granular synthesis example found on CCRMA website[4] based on this approach. In this example, line 06-10 in the file '*grain.ck*' (Figure 5 above) plays one grain of 50msec at once, continuously without an interval. This code is executed as a separate *shred* in the other program '*playgrain.ck*' (Figure 5 below) in line 02. After 25msec, the same 'grain.ck' is another *shred* in line 05.

Thus, the overlap-add of 50msec grains with 25msec interval can be achieved. Because ChucK's performs the above in logical synchronous time, this code in Figure 5 can result in sample-rate accuracy in timing and output precise synchronous granular synthesis sound unlike the pervious examples in Max/MSP and SuperCollider.

[2] Figure 3 of Max/MSP implementation also has the problem of timing inaccuracy. As described in Max 5 tutorial, '*The basic unit of time for scheduling events in Max is the millisecond*' (http://cycling74.com/docs/max5/tutorials/msp-tut/msphowmspworks.html)

[3] see [22, 23] for the detail of *strongly-timed* programming concept.

[4] CCRMA-Wiki MultiGrain Granular Synthesis in Chuck (https://ccrma.stanford.edu/wiki/MultiGrain_Granular_Synthesis_in_Chuck)

Such programming patterns in ChucK seem largely due to both its language design and library design. ChucK lacks an abstraction of note-level objects as '*synth*' in SuperCollider and normally unit-generator graphs is written in a top level of the code or enclosed in a function or a class; thus, a ChucK shred can be used to substitute a note-level object in the other languages. While it is still possible to assign one shred per grain and to model each grain as a note by terminating each shred after playing a single grain, this programming pattern seems to be generally avoided in ChucK; since Chuck's *SndBuf* object cannot share the buffer each other object and require to load the sound file for each object and this can lead to a large overhead in performance efficiency and the waste of memory space.

2.3.3 Pros and Cons of the Two Approaches

Both of the approaches described above have their own merits and demerits. There is a significant benefit in performance efficiency in implementing microsound synthesis techniques as unit-generators that run as native code. At the same time, by fully wrapping a synthesis algorithm within a unit generator object, the use of such a microsound synthesis unit-generator is made easier to use together with the other unit-generator objects such as a reverberator or an envelope shaper.

Yet, from the perspective of end-user programming, such an approach is less helpful when the intentions of end-user programming are in *exploratory design* and *exploratory understanding* [3]. Since what users allowed with such built-in objects is only to use them with the predefined interfaces and functionality, there are almost no opportunity for users to experiment by modifying the enclosed algorithms for creative exploration; for such an exploration, users need to modify the original source code of unit-generators or write their own. This requires a certain level of expertise in programming, which may be too difficult for most end-users.

The later approach to implement microsound synthesis techniques within a computer music language may suffer from the significant performance inefficiency compared to the unit-generators executed in native code. On the other hand, this approach seems to support exploratory design and understanding better in that the users can fully explore the given code; users can easily investigate the code written in the computer music language they already learned and modify them for better understanding or creative exploration of the synthesis algorithms.

Yet, in practice, there are still some problems that hinder programming activity by end-users. For instance, if a microsound synthesis technique is enclosed as a unit-generator, the output from such a microsound unit-generator can be easily applied to an envelope just by connecting it to an envelope unit-generator. Yet, when the programming patterns as seen in the example of Figure 4 and Figure 5 are used to implement microsound synthesis techniques solely in computer music languages, a end-user programmer must develop or acquire another programming pattern how to apply such an entire envelope to the output from each *synth* or *shred*; this requires more understanding of an underlying synthesis framework, which is not related to the concept of microsound synthesis techniques at all.

As described in the previous section, microsound synthesis is normally conceptualized as manipulations and overlap-add of short sound particles. Yet, the example in Max/MSP uses a delegation mechanism for overlap-add of microsounds and the example in SuperCollider uses a *synth*, a note-level object, and the one in ChucK uses a *shred,* a non-preemptive thread, instead. Such a gap may also cause difficulty in comprehension and modification of a program, since as described later, programming activity is considered as a mapping between knowledge in application domain and computing domain.

3. RECONSIDERING SOUND SYNTHESIS FRAMEWORKS

In this section, we discuss our motivation and perspective why and how microsound synthesis techniques should be considered in sound synthesis framework design in a computer music language. Some part of this discussion can be described also in our previous publication [15].

3.1 Abstraction in Computer Music Language Design

It has been frequently argued that while abstractions play a significant role in software design, various serious usability problems can be caused when abstractions applied to software design are incompatible with the users' conceptualization. As Blackwell discusses "*even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centered design* [2]. Blandford and her colleagues also provide a valuable perspective for this abstraction issue in their framework called CASSM (Concept-based Analysis of Surface and Structural Misfits"), "*the purpose of which is in the identification of misfits between the way the user thinks and the representation implemented within the systems* " [4]. In CASSM, such incompatible abstractions or gaps between the user's conceptualization and the system's representations are called *'conceptual misfits'* and such conceptual misfits can lead to significant usability problems in software design.

Generally speaking, domain-specific languages are normally built on certain software frameworks or libraries. As Fowler describes in [11, p.29], "*the most common way to build in abstraction is by implementing a library or framework"* and *"in this view a DSL is a front-end to a library providing a different style of manipulation to the command-query API"*; A computer music language can also be considered as such a 'front-end' to an underlying library or framework. Hence, since programmers are considered to "*use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping*" [9, p22], it is highly desirable to consider *conceptual misfits* between expert knowledge and the representation within synthesis frameworks in computer music languages.

3.2 Conceptual Misfits in Microsound Synthesis

Then, what kind of conceptual misfits can be found in computer music languages, regarding microsound synthesis? Roads describes nine musical time-scales as a "*comprehensive view"* to the "*temporal hierarchy of structure in music compositions*", which are *infinite, supra, macro, meso, sound-object, micro, sample, subsample, infinitesimal* [18, p.3]. Table 1 below describes the time-scales below *meso* time-scale as Roads explains, which are within the scope of sound synthesis frameworks.

Sound object	*"A basic unit of musical structure, generalizing the traditional concept of note to include complex and mutating sound events on a time scale ranging from a fraction of a second to several seconds."*
Micro	*"Sound particles on a time scale that extends down to the threshold of auditory perception (measured in thousandths of a second or milliseconds)."*
Sample	*"The atomic level of digital audio systems: individual binary samples or numerical amplitude values, one following another at a fixed time interval. The period between samples in measured in millionths of a second (microseconds)."*
Subsample	*"Fluctuations on a time scale too brief to be properly recorded or perceived, measured in billionths of a second (nanoseconds) or less"*
Infinitesimal	*"The ideal time span of mathematical durations such as the infinitely brief delta functions"*

Table 1. The time-scales of *sound-object, micro, sample, subsample and infinitesimal* as Roads describes [18, p.3].

Roads' description of nine musical time-scales is considered expert knowledge as it covers the whole range of different time-scales in computer music. The question is whether or not any misfit can be found between these musical time-scales and the representations within sound synthesis frameworks. Table 2 describes the typical representations found within many sound synthesis frameworks.

note/ synth/ patch		These correspond to *sound object* time-scale in Roads' classification. While a *"note"* in CSound score file and a *"synth"* in SuperCollider are clearly close to '*the traditional concept of note*' as in Table 1, a 'patch' in visual computer music languages such as Max/MSP or PureData may also be considered to include the upper level time-scales such as *meso* or *macro,* since it normally include musical control algorithms together within synthesis algorithms.
unit-generator	**audio vector**	In many computer music systems, unit-generators perform DSP by fixed size vectors of samples called *audio vector* [7, p.467]. Normally, audio vectors are not directly visible to users.
	sample	Within unit-generators, each sample for signal output is computed by iterating input audio-vectors, (typically in a *for* loop).

Table 2. The typical representations implemented within software sound synthesis frameworks.

There can be found several conceptual misfits between such typical representations within sound synthesis frameworks and Roads' nine musical time scales. Obviously, there are no counterpart entities in time-scales of *microsound, subsample, and infinitesimal* in Table 2. While computer music may rarely involve the later two, we consider the lack of counterpart to such entities such as *short sound particles* in microsound time-scale is a significant problem that is causing 'incompatible abstraction' in computer music languages; the approach of implementing unit-generators maps microsound entities incompatibly to unit-generators, enclosing everything in sample time-scale and therefore users have much less opportunity to manipulate objects in microsound time-scale. The approach of implementing microsounds as sound objects also involves such incompatible mapping, which may hinder users from comprehending the code.

The presence of *audio vector* [6, p.467], which doesn't exist in Roads' musical time-scale, results in so-called *control rate,* which makes it difficult to schedule microsound with sample rate accuracy. At the same time, since the notion of *audio vector* is normally implicit, one may encounter difficulty in understanding why a microsound synthesis program one wrote produces inaccurate output, which differs from what is expected.

From such a perspective as above, we argue that the integrations of objects and manipulations for microsounds should be seriously considered as a part of sound synthesis framework design for computer music languages.

4. LCSYNTH: A DESIGN OVERVIEW

In this section, we briefly describe the design of our new synthesis language that integrates objects and manipulations for microsounds. By this integration, we aim to provide a much simpler programming model for microsound synthesis techniques. While we are not able to describe full language specifications, both because of page limit and of the current 'under-development' status of the language, this section provides an overview how the design for LCSynth integrates microsounds.

4.1 The Integration of Microsounds

```
01://define an instrument with the name 'granular'
02:synth granular {
03:  //granular synthesis algorithm.
04:  synmain(bufnum) {
05:    //----------initialization--------
06:    //first take out 30msec from the buffer.
07:    var sample = readBuf (bufnum  , 30::msec);
08:    //generate an envelope.
09:    var env    = genEnv  (\hanning, 30::msec);
10:    //applying the envelope to the fragment.
11:    var grain  = applyEnv(sample , env);
12:
13:    //---perform granular synthesis---
14:    //the loop to generate 50 grains.
15:    for (var i = 0; i < 50; i++){
16:      //write out grain directly to the dac.
17:      dac.out(grain);
18:      //20 msec interval.
19:      now += 20::msec;
20:    }
21:  }
22:}
//from the other control language under development.
//load a sound file onto the buffer.
loadBuf(0, "sounds/test.wav" );
//play the instrument defined above.
play granular(0);
```

Figure 6. A simple synchronous granular synthesis object in LCSynth

Figure 6 above shows an example of simple synchronous granular synthesis in the current version of LCSynth. Line 02-22 define a sound-object named *granular*. Line

04-22, the method, *synmain,* defines an entry point called when this sound object is instantiated. The names and numbers of the parameters can be defined by users as they wish. In this case, it defines only one parameter *buf-num*, which is used to specify the buffer no. to retrieve a grain from.

In line 07, the built-in function *readbuf* is called to retrieve 30ms samples from the buffer specified by *bufnum*. This built-in function returns the array of floating point values used as a microsound. The reference to the array is set to the variable *samp*. In line 09, the *genEnv* function generates an envelope to apply to the samples retrieved by the buffer. '*\hanning'* is a symbol, a string which bound to an internal unique ID. Giving '*\hanning'* and 30msec as parameters to genEnv, an envelope with hanning window shape is generated. The reference to this envelope is set to the variable *env.* In line 11, a grain is created by applying the envelope to the samples retrieved from the buffer.

The algorithm for synchronous granular synthesis is described between line 13 to line 19. The for-loop starts from line 15 generates 50 grains. First, in line 17, the *grain* object prepared in line 11 is written out to *dac*. This *dac* object simply write the given *grain* object to the internal sound I/O buffer, performing overlap-add with the existing signals already written to the buffer. Line 19 explicitly advances the time 20msec. Since LCSynth is a strongly-timed programming language, this is performed in logical synchronous time. Thus scheduling will be exactly 20msec in sample-rate accuracy.

Currently LCSynth is a small language designed only for synthesis. For prototyping and testing, we are also developing a simple language that hosts LCSynth inside. The rest of the code in Figure 5 describes an instantiation of the defined sound object in this simple host language. First, it load the sound file to the buffer no. 0 by *loadbuf* built-in function and use *play* to instantiate and play the sound object.

4.2 Using Unit-Generators

LCSynth still has unit-generators. Yet, as suggested by Roads' musical time-scale, the unit-generators and audio graphs are separated from microsound objects and manipulations in LCSynth, as they work in *sample time-scale*. Figure 6 above describes two simple examples how the unit-generators are implemented in LCSynth. In Example (A) in Figure 6, line 02 to line 07 defines a sound object, with a name *'sineoscA'*. The audio graph is defined inside the block, which starts from keyword *ugen* as in line 04 to line 06. In Line 05, first '*s:~Sine(440)*' instantiate a sine wave unit-generator with the argument of 440, which set the frequency of this sine wave unit-generator on creation. The '*s:'* is a label given to this instance and can be used to refer to this unit-generator later in the code. '~DAC(0)' also instantiates a DAC output unit-generator, with an argument of 0, which is the channel no of DAC, In stereo setting, this is the number for left channel. In '~DAC(0)', there is no label given to this unit-generator object. To connect these unit-generators form an audio-graph, '->' operator is used. This operator performs similar function as '=>' operator in ChucK and connect the output from the unit-generator on left hand side to the input of the unit-generator on right hand side.

```
-----------------Example (A)-----------------
01://define an instrument with the name 'sinoscA'
02:synth sineoscA {
03:  //define an audio graph for this synth object.
04:  ugen {
05:    s:~Sin(440) -> ~DAC(0);
06:  }
07:}

-----------------Example (B)-----------------
01://define an instrument with the name 'sinoscA'
02:synth sineoscB {
03:  //define an audio graph for this synth object.
04:  ugen {
05:    s:~Sin(440) -> ~DAC(0);
06:  }
07:  //control algorithm within this sound object.
08:  synmain(period){
09:    //update sin wave frequency every 50msec.
10:    while(true){
11:      //set rand val between 220 and 1760 to freq
12:      s.setFreq(rand(220, 1760));
13:      now += period;
14:    }
15:  }
16:}
```

Figure 7. Two examples of the use of unit-generators in LCSynth.

To control unit-generators, *synmain* function can be used to enclose control algorithms inside a sound object. Figure 7, Example (B) describes such an example. The audio graph constructed between line 04-06 is the same as Example (A). When this sound object is instantiated, *synmain* is called with an argument, which is set to the parameter *period*. The 'while' loop between 10 -14 updates the frequency of the sin wave unit-generator once per duration specified by *period*, by calling *setFreq* method of sine wave unit-generator with its given label *s*, specifying the new frequency generated by *rand*() function.

4.3 Collaboration between Microsounds and Unit-Generators

As discussed in the previous section, it is often the case that the entire output of microsound synthesis techniques needs to be applied other signal processing. For instance, while each grain in granular synthesis may be applied its own envelope, one may want to apply an ADSR envelope to the entire sound output that consist of these grains. Thus, it is highly desirable to provide simpler means for such collaboration between microsound time-scale and sample time-scale.

In LCSynth, this collaboration is achieved by using delay unit-generators. We describe such an example in Figure 8. The audio graph is constructed in line 04 – 08. A delay line with its internal buffer for 1 sec is given a label *dly* and connected to an ADSR envelope shaper. The output of this envelope shaer is connected to a dac output. Line 11-30 describes a granular synthesis algorithm. This time, we first generate an envelope in line 16. In the main loop from line 20, a random position between 0 to 2000msec is set to *pos*. This *pos* is used as a offset position to retrieve samples from the buffer later on. In line, 23 *readBuf* function retrieve 30msec samples from the given offset *pos* from the buffer no specified by *bufnum*. Line 24 applies an envelope to these samples to generate a grain.

```
01://define an instrument with the name 'envgran'
02:synth envgran {
03:  //create an audio graph.
04:  ugen {
05:    //connect a delay to an ADSR envelope shaper.
06:    //then to DAC.
07:    dly:~Delay(1::sec) -> ~ADSR(0.24, 1, 0.5)
08:                       -> ~DAC(0);
09:  }
10:  //granular synthesis algorithm.
11:  synmain(bufnum) {
12:    //----------initialization--------
13:    //first take out 30msec from the buffer.
15:    //generate an envelope.
16:    var env  = genEnv(\hanning, 30::msec);
17:
18:    //---perform granular synthesis---
19:    //the loop to generate 50 grains.
20:    for (var i = 0; i < 50; i++){
21:      //generate a grain.
22:      var pos   = rand(0, 2000)::msec
23:      var sample= readBuf (bufnum, 30::msec, pos);
24:      var grain = applyEnv(sample, env);
25:      //write one grain into the delay line.
26:      dly.write(grain);
27:      //wait for 20 msec.
28:      now += 20::msec;
29:    }
30:  }
31:}
```

Figure 8. An example of the collaboration between microsounds and unit-generators in LCSynth

In line 26, instead of writing a grain directly to dac output as in Figure 6, the grain is written to the delay line *dly*. This also performs overlap-add with the existing samples in delay line. Line 28 advances the logical synchronous time 20 msec. This causes samples to be computed for 20 msec by the audio graph. The grains written to the delay line are output to the envelope shaper sample-by-sample. LCSynth is implemented with incremental garbage collection. The objects instantiated within *for* loop are automatically freed by the garbage collector.

Thus, LCSynth provides simple means for the collaboration between two different time-scales of sample level and microsound level in its language design.

4.4 DISCUSSION

As we described above, LCSynth integerates objects and manipulations for microsounds in its language design.It separates *sample time-scale* where unit-generators work and *microsound time-scale* where microsound synthesis techniques are performed.

The programming model may seem slightly similar to how ChucK's *Unit Analyzers* work in that both largely depend on strongly-timed programming. Yet, while the *Unit Analyzer* concept is invented for "*seamlessly combining analysis and synthesis"* in ChucK [10], our aim of this new language design is to provide a clear programming model that maps between Roads' musical time-scales and language design by providing counterpart entities related to each musical time-scale in a sound synthesis language; *Sound object time-scale* is mapped to *synth* sound objects, *microsound time-scale* to the *synmain* function and *sample time-scale* to the audio graph made in *ugen* block. The objects and manipulations for microsound synthesis are provided in its language design to avoid incompatible mapping that hinder programming activity as seen in the existing languages. We believe this organization well corresponds to Roads' musical time-scales and provide better mapping between expert knowledge and language design in computer music; Such an issue how microsound synthesis techniques should be designed and implemented has been one of the major concerns in sound synthesis framework design [1, 7, 13, 21].

Moreover, LCSynth can perform many other DSP techniques by the same programming model; For instance, to perform waveset transposition, each waveset simply needs to be rescaled in its duration and be written to the dac output or a delay line. The delay line in LCSynth can be used not only to write signals but also to retrieve signals. This makes it possible to perform FFT/IFFT on the retrieved samples in the same programming model; such a simple programming model is also a benefit of LCSynth's language design, as short time FFT/IFFT is also considered to belong to microsound time-scale.

Such a simple programming model applicable to many different related techniques would be beneficial for musical explorations by computer musicians. Additionally garbage collection in LCSynth also significantly facilitates computer music programming; users do not have to worry about explicit memory management.

5. PROJECT STATUS AND EVALUATION

LCSynth is still a small language designed for sound synthesis and currently being hosted by simple APIs in C++ and a small test script under development. We are particularly interested in developing a new computer music language to enclose LCSynth as its synthesis module. The language design process involved heuristic evaluations by external experts based on the framework provided by HCI researchers such as Cognitive Dimensions of Notations [3]. Speaking generally, the evaluations by external experts were very positive to our language design to integrate objects and manipulations for microsounds. More details on the evaluations are described in our previous publication [15], together with other additional language design proposals still under progress.

The efficiency in run-time performance is also a significant concern in computer music languages. While LCSynth compute signals sample-by-sample, it compiles the audio graphs of unit-generators into small bytecode and performs DSP by bytecode This software design resulted in a significant improvement in performance efficiency and we haven't found any significant performance inefficiency at this point. The current version can run more than 128 instances of a simple sine wave oscillator connected to DAC at once without glitch on MacbookAir (Intel Core i7/1.8GHz); This is likely to be fast enough for most computer music practices. We are planning to investigate the detailed performance evaluation in the future after several iterations of prototyping.

6. CONCLUSION

We developed LCSynth, a new strongly-timed synthesis language that integrates objects and manipulations for microsounds. As implied by conceptual misfits between nine musical time-scales by Roads and the representation implemented within the traditional unit-generator based frameworks, we reconsidered the abstraction in the design of our new synthesis language and integrated objects and manipulations for microsounds in our new language. Such a new abstraction can provide a simpler programming model for microsound synthesis techniques and would be desirable for creative musical exploration.

7. ACKNOWLEDGEMENT

The authors thank Prof.Richard Dudas at Hanyan University for providing his granular synthesis Max/MSP patches and the related information.

8. REFERENCES

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

[2] Blackwell, A.F., Church, C. and Green, T.R.G. The Abstraction is 'an Enemy': Alternative Perspectives to Computational Thinking. In *Proc. PPIG08*, 2008

[3] Blackwell, A.F. and Green, T.R.G. Notational Systems – the Cognitive Dimensions of Notation Framework, *HCI Models, Theories and Frameworks: Toward a Multidisciplinary Science,* Morgan Kaufmann, 2003

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

[5] Boulanger, R. et al. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing and Programming*. The MIT Press, 2000

[6] Boulanger, R. and Lazzarini, V. *The Audio Programming Book*, The MIT Press, 2010

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

[8] Clarke, J. M. et al. VOCEL: New implementations of the FOF synthesis method. In *Proc ICMC88*, 1988.

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

[10] Fiebrink R., Wang, G. and Cook, P. Support for MIR Prototyping and Real-Time Applications in the ChucK Programming Language. In *Proc. ISMIR'08,* 2008

[11] Fowler, M. *Domain-Specific Languages*, Addison-Wesley, 2010

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

[13] Geiger, G. Abstractions in Computer Music Software Systems. *DEA Thesis, Universitat Pompeu Fabra*. 2005

[14] Mathews, M.V. et al. The Technology of Computer Music. The MIT Press, 1969

[15] Nishino, H. How Can a DSL for Expert End-User be Designed for Better Usability?: A Case Study in Computer Music. In *Proc. SIGCHI'12,* 2012

[16] Puckette, M. The Theory and Technique of Electronic Music. World Scientific Press, Singapore, 2007.

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

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

[19] Roads, C. Introduction to Granular Synthesis. In *Computer Music Journal, Vol.12(2),* 1988

[20] Rodet, X. Time-Domain Formant-Wave-Function Synthesis. In *Computer Music Journal Vol.8(3),* 1984

[21] Wakefield, G.D. and Simith, W. Using Lua for Multimedia Composition, In *Proc. ICMC07*, 2007

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

[23] Wang, G. and Cook, P. ChucK: A Programming Language for On-the-fly, Real-time Audio Synthesis and Multimedia, In *Proc. ACM Multimedia'04,* 2004

[24] Wilson, S. et al. *The SuperCollider Book*, The MIT Press, 2011

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

[26] Zicarelli, D. An Extensible Real-Time Signal Processing Environment for Max. In *Proc. ICMC98*, 1998