LC: A New Computer Music Programming Language with Three Core Features

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
Interactive and
Digital Media Institute,
National University of Singapore
idmnr@nus.edu.sg

ABSTRACT

This paper gives a brief overview of the three core features of LC, a new computer music programming language we prototyped: (1) prototype-based programming at both levels of compositional algorithms and sound synthesis, (2) the mostly-strongly-timed programming concept and other features with respect to time, and (3) the integration of objects and functions that can directly represent microsounds and the related manipulations for microsound synthesis. As these features correspond to issues in computer music language design raised by recent creative practices, such a language design can benefit both the research on computer music language design and the creative practices of our time, as a design exemplar.

1. INTRODUCTION

While the advance of computer technology and programming language research has largely influenced the evolution of computer music languages, issues found in creative practices have also motivated the development of new computer music languages. For instance, "the need for a simple, powerful language in which to describe a complex sequence of sound" in the early days of computer music [13, p.34] led to the invention of the unit-generator concept, which still serves as a core abstraction for digital sound synthesis. In another example, Max and some other languages for IRCAM's Music Workstation were designed with the motivation that "musicians with only a user's knowledge of computers could invent and experiment with their own techniques for synthesis and control" [18].

Therefore, the problems revealed by the creative practices can also be regarded as significant design opportunities for a new computer music programming language. In the design and development of LC, a new computer music programming language, we also took the issues raised by the creative practices of our time into account. While LC has been partly described in our previous works [14, 15, 16, 17], significant extensions have been made to its original language specification in the design process.

Copyright: © 2014 Hiroki NISHINO et al. This is an open-access article dis- tributed under the terms of the Creative Commons Attribution License 3.0 Unported*, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*

In this paper, we first address three issues in computer music language design, which were raised by the creative practices of our time: (a) the insufficient support for dynamic modification of a computer music program, (b) the insufficient support for precise timing behavior and other features with respect to time, and (c) the difficulty in microsound synthesis programming.

In the following sections, we discuss these problems and then how they correspond to the three core features of LC: (1) prototype-based programming, (2) mostly-strongly-timed programing, and (3) the integration of the objects and functions for microsound synthesis within its sound synthesis framework, together with related works and a brief discussion.

Such a discussion regarding the language design and the issues found with the creative practices can benefit further research on computer music languages and the investigation on how creative exploration by computer musicians should be supported by computer music languages.

2. THREE ISSUES IN TODAY'S COMPUTER MUSIC LANGUAGE DESIGN

2.1 The insufficient support for dynamic modification of a computer music program

Recent computer music practices suggest a significant need for more dynamic computer music programming languages today. For example, live-coding performances [6], involve the creation and modification of computer music programs on-the-fly on stage, even while the programs are being executed. In addition, *dynamic-patching* as seen in *reacTable* [12] involves the dynamic modification of a sound synthesis graph.

However, many computer music languages still exhibit certain usability difficulties when performing dynamic modification at least at one of these levels. Such difficulties can obstruct further creative musical explorations. As the degree of support for the dynamism in a programming environment depends not just on the design of a library or a framework utilized, but also on the basic language design, which can be substantially limiting. It is highly desirable to consider such an issue as one of the important criteria from the earliest stage of the language design process.

2.2 The insufficient support for precise timing behavior and other features with respect to time

The precise timing behavior of a computer music system has become a traditional issue. Even in earlier decades when a real-time interactive computer music system consisted of a computer and external synthesizer hardware, the slow processing speed of CPUs and the low bandwidth of hardware interfaces have motivated the research on the improvement of the timing precision required for better musical presentation of live computer music compositions[1]. Today, even sample-rate accurate timing behavior is considered desirable. For instance, to render the output of a microsound synthesis technique as theoretically expected, sample-rate accurate timing precision in scheduling microsounds is essential.

While some recent computer music languages provide sample-rate accurate timing behavior as in ChucK [22], LuaAV [21], their synchronous behavior can result in the temporary suspension of real-time DSP in the presence of a time-consuming task, as it blocks the audio computation until all the scheduled tasks are finished. Moreover, the features with respect to time that were seen in the computer music languages of earlier eras, such as timing constraints and time-fault tolerance, seem to not be considered in many recent computer music languages; even *Impromptu* [20], which is a good exception that is clearly designed with such considerations, still lacks some desirable features with respect to time. For example, Impromptu cannot handle the violation of execution-time constraints.

As above, the support for precise timing behavior and other features with respect to time is still an issue of significance in today's computer music language design.

2.3 The difficulty in microsound synthesis programming

Broadly speaking, usability difficulties can be caused when the abstractions applied to the software are incompatible with what a user thinks. As "the co-evolving nature of technology adoption results in new concepts emerging through use of technology", such a gap caused between the existing abstractions and emerging concepts "may introduce usability difficulties", which did not exist previously [3].

This view may correspond to the unit-generator concept and microsound synthesis, as the latter was brought into practice much later than the establishment of the former; one of the earliest well-known experiments in microsound synthesis is one by Roads in 1974 [19, p.302], long after the invention of unit-generator concepts in 1960 [7, P.26].

Indeed, several researchers have already discussed the gap between the traditional unit-generator concept and microsound synthesis. Bencina discusses such an issue in the object-oriented software design for a software granular synthesizer in [2]. In another example, the design of Brandt's Chronic computer music language is also highly motivated by problems exhibited in the traditional unit-generator concept when describing microsound synthesis techniques [4]. While its application domain focuses only on frequency-domain signal processing and analysis, Wang *et al.* describe a similar issue when discussing ChucK's unit-analyzer concept [23].

However, The former two works are not very adaptable to the design of a real-time interactive computer music language. The work by Bencina targets the stand-alone software rather than the language design. Brandt's Chronic is a non real-time computer music language, the design of which still leaves 'an open problem' for application to real time computer music languages because of its acausal behavior[2] [4, p.77]. The target domain of ChucK's unit-analyzer concept is only signal processing and analysis in the frequency-domain, and it lacks the generality to apply to various microsound synthesis techniques; The substantial necessity for further research on more appropriate abstractions that can tersely describe microsound synthesis techniques still remains.

3. THREE CORE FEATURES OF LC

3.1 Prototype-based programming at both levels of compositional algorithms and sound synthesis

In prototype-based languages, "each object defines its own behavior and has a shape of its own", whereas "each object is an instance of a specific class" in class-based languages [11, p.151]. Unlike class-based languages, slots (or fields and methods) can be added to an object dynamically after its creation. Prototype-based languages allow a significant degree of flexibility and tolerance against the dynamic modification of a computer program at runtime. The LC language adopts prototype-based programming at both levels of compositional algorithms and sound synthesis, for better support of dynamic modifications to a computer program.

At the compositional algorithm level, *Table* is provided for prototype-based programming. Figure 1 describes a simple example of prototype-based programming by *Table*. A shown, LC is a dynamically-typed language and also supports other features such as duck-typing and first-class functions.

LC also supports prototype-based programming at the sound synthesis level. Instead of *Table*, *Patch* is provided, which can be utilized to build and modify a unit-generator graph dynamically. Figure 2 (example *a*) describes an example of creating and modifying a *Patch* object. As shown in Figure 2 (example *b*), syntax sugars

[1] FORMULA well represents the research on timing precision and the time-related features in its era, even though its target application domain was still a hybrid computer music system that consists of a computer and the external MIDI synthesizer(s) [1].

[2] In Chronic, a future event can influence the result already made. As Brandt admits, this is a significant obstacle for the adoption of its programming model to a real-time computer music language [4. p.77].

are provided to make the code more readable. Additionally, a patch can be used as a subpatch (see Figure 3)

```
01: //create an object ex nihilo and initialize it.
02: var obj = new Table();
03: obj.balance = 0; //the initial balance is 0.
04: //attach the methods to the object.
05: obj.deposit = function (var self, amount){
06:   self.balance += amount;
07:   return self;
08: };
09: obj.withdraw = function (var self, amount){
10:   self.balance =+ amount;
11:   return self;
12: };
13: obj.showBalance = function (var self){
14:   println("current balance:" .. self.balance);
15:   return self;
16: };
17: //deposit and print.
18: obj.deposit(obj, 1000);
19: obj.showBalance(obj); //this prints out '1000'
20: //obj->method(a, b, c) is a syntax sugar of
21: //obj.method(obj, a, b, c).
22: obj->withdraw(750);
23: obj->showBalance(); //this prints out '250'.
```

Figure 1. An example of prototype-based programming at the level of compositional algorithms in LC.

```
Example (a)
01: //create a patch object.
02: var p = new Patch();
03:
04: //create ugens and assign them to the slots.
05: p.src = new Sin~(freq:440);
06: p.rev = new Freeverb~();
07: p.dac = new DAC~();
08:
09: //make connections.
10: p->connect(\src, \defout, \rev, \defin);
11: p->connect(\rev, \defout, \dac, \defout);
12:
13: //'compile' the patch to reflect above.
14: p->compile();
15: //play the patch and wait for 1 sec.
16: p->start();
17: now += 1::second;
18:
19: //modify the unit-generator graph
20: p.src = new Phasor~(freq:1760);
21: p->connect(\rev, \defout, \dac, \ch1);
22: p->disconnect(\rev, \defout, \dac, \defout);
23: p->compile();

Example (b)
01: //the patch statement can create and connect
02: //ugens at once and then perform compilation.
03: var p = patch {
04:   //`=>' builds a connection.
04:   src:Sin~(freq:440) => rev:Freeverb~()
05:   => dac:DAC~();
06: };
07:
08: //play the patch and wait for 1 sec.
09: p->start();
10: now += 1::second;
11:
12: //modify the unit-generator graph.
13: update_patch(p){
14:  src:Phasor~(freq:1760);
15:  //`=|' can be used for disconnection.
16:  rev =| dac;
17:  //the inlet & outlet can be given as below.
18:  rev {\defout => \ch1} dac;
19: };
```

Figure 2. An example of prototype-based programming at the level of sound synthesis in LC.

```
01: //Inlet~ and Oulet~ can be used in a subpatch.
02: var s = patch {
03:  defin:Inlet~() {\defout => \amp} Sin~(440)
04:  => defout:Outlet~();
05:};
06: //a simple tremolo effect. the above 's'
07: //is given as a subpatch (`sub:s' on line 09)
08: var p = patch {
09:      amp:Sin~(freq:5) => sub:s => dac:DAC~();
10: };
11: p->start();
```

Figure 3. An example of subpatch in LC.

3.2 Mostly-strongly-timed programming and other features with respect to time

3.2.1 Mostly-strongly-timed programming

The ideal synchronous hypothesis underlies the strongly-timed programming concept (and other similar synchronous approaches). It assumes "all computation and communications are assumed to take zero time (that is, all temporal scopes are executed instantaneously)" and "during implementation, the ideal synchronous hypothesis is interpreted to imply the system must execute fast enough for the effects of the synchronous hypothesis to hold" [5, p.360]; in a computer music language designed with such a synchronous approach, this assumption can be invalidated when the deadline for the next audio computation is missed because of a time-consuming task. This invalidation leads to the temporal suspension of audio output, which is undesirable for computer music programs. As this problem in strongly-timed programming is rooted in the underlying concept of the ideal synchronous hypothesis, the temporal suspension of audio output in the presence of a time-consuming tasks is inevitable without making any extension to the original concept.

LC proposes a new programming concept, *mostly-strongly-timed programming,* which extends *strongly-timed programing* with the explicit context switching between the synchronous/non-preemptive behavior and the asynchronous/preemptive behavior. When the current context of the thread is asynchronous/preemptive, the underlying scheduler can suspend the execution of the thread at an arbitrary timing, even without the explicit advance of time.

Thus, mostly-strongly-timed programming allows the time-consuming part of a task to be executed without suspending real-time DSP and to run in the background, while maintaining the precise timing behavior of strongly-timed programming. To switch the context explicitly, *sync* and *async* statements can be used. These statements will execute the following statement (or compound statement) in the synchronous/non-preemptive and asynchronous/preemptive contexts respectively. These two statements can be nested. Figure 4 describes a simple example of mostly-strongly-timed programming.

3.2.2 Other features with respect to time

3.2.2.1 Timing-Constraints

LC can express both start-time constraints and execution-time constraints with sample-rate accuracy. For start-time constraints, both *patch* and *Thread* objects can be given an offset to the start-time as an argument. For execution-time constraints, the *within-timeout* statement is provided. Figure 5 and Figure 6 describe these features respectively. As shown, when the code consumes more time than the given constraint by a *within* statement during the execution of its following statement (or blocked statements), it immediately jumps to the statement (or blocked statements) in the matching *timeout* block. When *timeout* is

omitted, the code simply jumps to the next statement after the *within* statement. As seen in Figure 7, execution time-constraints can be correctly nested.

3.2.2.2 Time-tagged message communication

In LC, *the message-passing model* is applied to the inter-thread communication. When a message is sent out, the delivery timing of the message can be specified. Figure 8 describes the example of message-passing in LC. As shown, when the '<-' operator is used for message passing, the delivery time or timing offset can be given. When any value of the type *time* is passed, it is interpreted as the delivery time. If the value is of the type *duration*, it is interpreted as a timing offset.

```
01: //'sync' is the default context. create a patch
02: // to make the suspension of DSP audible.
03: var p = patch {
04:   Sin~() => DAC~();
05: };
06: p->start();
07: //loading large files and extracting wavesets.
08: //as DISK I/O can be time consuming, this can
09: //temporarily suspend the real-time output.
10: LoadSndFile(0, "/large_snd_file.aiff");
11: var wavesets = ExtractWavesets(0);
12:
13: //performing it in `async'.
14: async {
15:   //as this block can be preempted without
16:   //the advance of logical time, the suspension
17:   //of the audio computation does not occur.
18:   LoadSndFile(0, "/large_snd_file.aiff");
19:   wavesets = ExtractWavesets(0);
20: }
21:
22: //sync/async can be nested freely
23: sync {
24:   //now in the synchronous context
25:   some_function_call(1, 2,3 );
26:
27:   //switch to the asynchronous context
28:   async {
29:     some_ohter_function_call(4, 5);
30:     //switch to the synchronous context again.
31:     sync {
32:       yet_anohter_function_call(4, 5);
33:     }
34:     //now back to the asynchronous context
35:     println("done.");
36:   }
37:   //now back to the synchronous context
38:   println("bye!");
39: }
```

Figure 4. An example of mostly-strongly-timed programming in LC.

```
01: //giving the start-time offset to a patch.
02: var p = patch {
03:   Sin~(880) => DAC~();
04: };
05: //the patch starts 1 second later.
06: p->start(offset: 1::second);
07:
08: //giving the start-time offset to a thread.
09: //create a first class function.
10: var f = function(var message){
11:   println("message : " .. message);
12: };
13:
14: //create a thread by LC's '@' operator.
15: var thread = f@("Hello, world!");
16: //the thread starts executing after 2 second.
17: thread->start(offset: 2::second);
```

Figure 5. An example of start-time constraints in LC.

```
01: //giving the execution-time constraints
02: within(2::second){
03:   var cnt = 0;
04:   while(true){
05:     println("count : " .. cnt);
06:     now += 0.5::second;
07:     cnt += 1;
08:   }
09:   //the below code is never reached.
10:   println("done.");
11: }
12: timeout {
13:   println("timeout!");
14: }
15: //'time out' block can be omitted.
16: within(3::second){
17:   async while(true) println("*");
18: }
```

Figure 6. An example of execution-time constraints in LC (1).

```
01: within(1::second){
02:   within(2::second){
03:     //the code jumps to the outer timeout block
04:     //exactly after 1 second.
05:     now += 3::second;
06:   }
07:   //this timeout block will never be reached.
08:   timeout {
09:     println("the inner 'timeout'.");
10:   }
11: }
12: //the code jumps to below block as expected.
13: timeout {
14:   println("the outer 'timeout'.");
15: }
```

Figure 7. An example of execution-time constraints in LC (2).

```
01: //a function to be launched as a thread.
02: var f = function() {
03:   var thread = GetCurrentThread();
04:   while(true){
05:     //receive a message in the blocking mode.
06:     var msg = thread->recv(\blocking);
07:     if (msg == \quit){
08:       break;
09:     }
10:     println("message :" .. msg);
11:   }
12:   println("quit.");
13:   return;
14: };
15:
16: //create and start a thread.
17: var thread = f@();
18: thread->start();
19:
20: //sending messages...
21: //deliver the message immediately.
22: thread <- "Hello!";
23:
24: //deliver the message at the given 'time'.
25: thread <- @now + 1::second, "1 second passed";
26:
27: //deliver the message after the given duration.
28: thread <- @2::second, "2 second passed";
29: thread <- @3::second, \quit;
```

Figure 8. An example of time-tagged inter-thread message communication in LC.

3.3 The integration of the objects and library functions that can directly represent microsounds and the related manipulations for microsound synthesis

The sound synthesis framework of LC integrates the objects and functions that can directly represent the microsounds and related manipulations for microsound synthesis. LC was first designed as a hosting language to enclose the LCSynth sound synthesis language [15, 17], yet there has been a significant degree of modifications made in the sound synthesis framework since then.

In the current version of LC, the microsound synthesis objects and functions are completely separated from the unit-generator sound synthesis framework in the current version. However, the basic programming model for microsound synthesis in LCSynth as described in [17] is still applicable to LC programs.

In LC, *Samples* is the object used to represent a single microsound. *Samples* is an immutable object, which contains the sample values within. There is no limitation for the sample size[3]. *SampleBuffer* is a mutable version of *Samples*. These two objects are mutually convertible by calling *toSampleBuffer* and *toSampleBuf* method.

```
01: //instantiate a new SampleBuffer object and
02: //fill it with sinewave of 256 samp freq * 4.
03: var sbuf = new SampleBuffer(1024);
04: for (var i = 0; i < sbuf.size; i+=1){
05:   sbuf[i] = Sin(3.14159265359 * 2 *
06:                (i * 4.0 / sbuf.size));
07: }
08:
09: //create a grain.
10: //first convert it to a Samples object.
11: var tmp = sbuf->toSamples();
12  //apply a hanning window.
13: var win = GenWindow(tmp.dur, \hanning);
14: var grn = tmp->applyEnv(win)->resample(440);
15:
16: //perform synchronous granular synthesis
17: within(5::second){
18:   while(true){
19:     PanOut(grn, 0.0); //0.0 = center.
20:     now += grn.dur / 4;
21:   }
22: }
```

Figure 9. An example of synchronous granular synthesis in LC.

```
01: //load the sound file onto Buffer No.0.
02: LoadSndFile(0, "source.aif");
03:
04: //perform sound synthesis for 2 seconds.
04: within(2::second){
05:   //these are the synthesis parameters.
06:   var pitch  = 2;
07:   var rpos   = 0::second;
08:   var grnsize= 512;
09:   var grndur = grnsize::samp;
10:   var win    = GenWindow(grndur, \hanning);
11:   var rdur   = grndur * pitch;
12:
13:   //perform pitch-shifting.
14:   while(true){
15:     //read the sound fragment.
16:     var snd  = ReadBuf(0, rdur, offset: rpos);
17:
18:     //resample and apply an envelope.
19:     var tmp = snd->resample(grnsize);
20:     var grn = tmp->applyEnv(win);
21:
22:     //output the grain. advance the read pos.
23:     PanOut(grn);
24:     rpos += grn.dur / 2;
25:
26:     //wait until the next timing.
27:     now  += grn.dur / 2;
28:   }
29: }
```

Figure 10. An example of pitch-shifting by granulation in LC.

Figure 9 and Figure 10 describe simple examples of *synchronous granular synthesis*[4], and *pitch-shifting by granulation* [19, p.127] in LC, respectively. As seen on line 05 in Figure 9, each sample within *Samples* and *SampleBuffer* is directly accessible by the '[]' operator.

Figure 11 shows a pictorial representation of waveset harmonic distortion. As shown, each waveset [5] is resampled to produce the harmonics of the original waveset and then overlap-added to the original after being weighted. Figure 12 shows a simple example only with the second harmonics, not weighted.

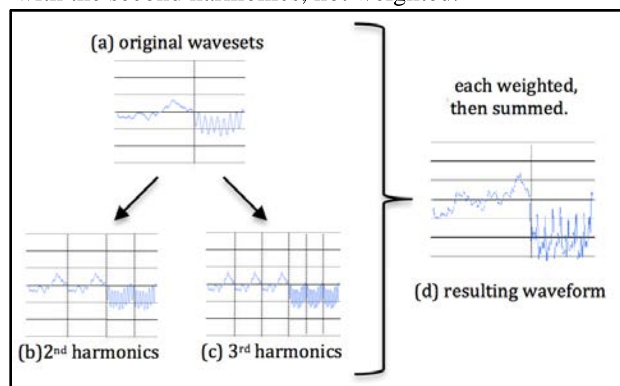

Figure 11. A pictorial representation of waveset harmonic distortion technique.

```
01: //load the sound file and extract wavesets.
02: LoadSndFile(0, "/sound/sample1.aif");
03: var wvsets = ExtractWavesets(0);
04:
05: //perform a simple waveset harmonic distortion.
06: for (var i =0; i < wvsets.size; i+= 1){
07:   //resample the waveset at the given index
08:   //so to create the 2nd harmonics.
09:   var orig = wvsets[i];
10:   var octup= orig->resample(orig.size / 2);
11:
12:   //schedule the original.
13:   WriteDAC(orig);
14:   //schedule two 2nd harmonics. give the offset
15:   //to schedule another right after the 1st one.
16:   WriteDAC(octup);
17:   WriteDAC(octup, offset:octup.dur);
18:
19:   //sleep until the next timing.
20:   now += orig.dur;
21: }
```

Figure 12. An example of waveset harmonic distortion in LC.

Figure 13 describes almost the same example of waveset harmonic distortion, but with the triangle envelope applied to the entire output. As shown, a *Samples* object can be written directly into the input of a unit-generator (lines 16 to 22) and the output of a unit-generator can be taken out as a *Samples* object (line 24). Figure 14 shows another example of waveset harmonic distortion. This example also applies reverberation together with envelop-shaping. As shown, a patch can be used in the same manner as the Figure 13 example. Furthermore, as seen lines 31to 46 in the Figure 14 example, if a patch is active, the patch automatically reads the given input and outputs the processed sound to the DAC output.

Thus, the collaboration between the unit-generator concept and LC's microsound synthesis abstraction can be performed quite easily.

[3] However, an out-of-memory exception is thrown if the memory allocation failed when creating a *Samples* or *SampleBuffer* object.

[4] In synchronous granular synthesis, the sound "results from one or more stream of grain" and "the grains follow each other at regular intervals" [19, p.93].

[5] A waveset is defined as "the distance from a zero-crossing to a 3rd zero-crossing" [25, Appendix II p.50]. In Figure 11 (left), each waveset is separated by grey lines.

To add more, FFT/IFFT can be also performed within the same microsound synthesis framework. Figure 15 describes a simple cross-synthesis example in LC.

```
01: //load the sound file and extract wavesets.
02: LoadSndFile(0, "/sound/sample1.aif");
03: var wvsets = ExtractWavesets(0);
04:
05: //create an triangle envelope ugen. trigger it.
06: var env = new TriEnv~(2::second);
07: env->trigger();
08:
09: //perform a simple waveset harmonic distortion.
10: for (var i =0; i < wvsets.size; i+= 1){
11:   //resample the waveset at the given index
12:   //so to create the 2nd harmonic.
13:   var orig = wvsets[i];
14:   var octup= orig->resample(orig.size / 2);
15:
16:   //write the original to the ugen input.
17:   env->write(orig);
18:   //write two 2nd harmonics. give the offset
19:   //to schedule another right after the 1st one.
20:   env->write(octup);
21:   env->write(octup, offset:octup.dur);
22:
23:  //read the output of the ugen. send it to dac.
24:   var out = env->pread(orig.dur);
25:   WriteDAC(out);
26:   //sleep until the next timing.
27:   now += wvsets[i].dur;
28: }
```

Figure 13. An example of waveset harmonic distortion in LC, with the triangle envelope applied to the entire output.

```
01: //load the sound file and extract wavesets.
02: LoadSndFile(0, "sample2sec.aif");
03: var wvsets = ExtractWavesets(0)
04: //create a patch and trigger the envelope
05: var pat = patch {
06:   defin:TriEnv~(2::second) => Freeverb~()
07:  defout::Outlet~();
08: };
09: pat.defin->trigger();
10:
11: //perform a simple waveset harmonic distortion.
12: for (var i =0; i < wvsets.size; i+= 1){
13:   //resample the waveset at the given index
14:   //so to create the 2nd harmonics.
15:   var orig = wvsets[i];
16:   var octup= orig->resample(orig.size / 2);
17:
18:   //write to the patch's default input.
19:   pat->write(orig);
20:   pat->write(octup);
21:   pat->write(octup, offset:octup.dur);
22:
23:  //read the output of the ugen. send it to dac.
24:   var out = pat ->pread(orig.dur);
25:   WriteDAC(out);
26:   //sleep until the next scheduling timing.
27:   now += wvsets[i].dur;
28: }
29:
30: //swap the outlet with DAC and play the patch.
31: update_patch(pat) {
32:   defout:DAC~();
33: };
34: pat.defin->trigger();
35: pat->start();
36:
37: //perform a simple waveset harmonic distortion.
38: for (var i = 0; i < wvsets.size; i+= 1){
39:   var orig = wvsets[i];
40:   var octup= orig->resample(orig.size / 2);
41:
42:   pat->write(orig);
43:   pat->write(octup);
44:   pat->write(octup, offset:octup.dur);
45:
46:   //as the patch is active, there is no need to
47:   //read the patch and send it to dac.
48:   now += wvsets[i].dur;
49: }
```

Figure 14. An example of waveset harmonic distortion in LC, with the triangle envelope and reverberation applied.

```
01: //load the sound files onto the buffers.
02: LoadSndFile(0, "/sound/sound1.wav");
03: LoadSndFile(1, "/sound/sound2.wav");
04: //the duration of each FFT/IFFT window and
05: //the number of the overlapping windows.
06: var dur =1024::samp;
07: var ovlp= 4;
08:
09: //process 800 frames.
10: for (var i=0; i < 800; i += 1){
11:   //first, extract snd fragments from the buffers.
12:   var src1 = ReadBuf(0, dur, offset:i* dur / ovlp);
13:   var src2 = ReadBuf(1, dur, offset:i* dur / ovlp);
14:
15:   //perform FFT. PFFT applies a window and returns
16:   //an array of Samples objects [magnitude, phase].
17:   var pfft1 = PFFT(src1, \hanning);
18:   var pfft2 = PFFT(src2, \hanning);
19:
20:   //cross synthesis
21:   var ppved = pfft1[0]->mul(pfft2[0]);
22:
23:   //perform IFFT and writes to the sound output.
24:   var pifft = PIFFT(ppved, pfft1[1], \hanning);
25:   //wait until the next timing.
26:   now += src1.dur / ovlp;
27: }
```

Figure 15. An example of cross-synthesis in LC.

4. DISCUSSION

4.1 Prototype-based programming in LC

As briefly mentioned in Section 2.1, while there exists the need for a more dynamic computer music language, the existing computer music languages exhibit certain problems at least at either the level of sound synthesis or the level of compositional algorithms.

For instance, as ChucK is a statically-typed class-based language, ChucK is not suitable for dynamic modification at runtime. Assume a variable *src* is assigned a *SinOsc* unit-generator; one cannot simply assign a *Phasor* unit-generator to *src* for replacement, since the types of these two objects differ. Using a common parent class *Ugen* for the type of *src* would hinder access to the fields or methods that exist in *SinOsc* or *Phasor*, but not in *Ugen*. Furthermore, it shows a certain degree of viscosity[6] in the modification of a synthesis graph, as it is required to disconnect the connections to the unit-generator to be replaced first, and then rebuild the connections to a new unit-generator. This is because ChucK builds the connections between the instances of the unit-generators rather than the variables.

SuperCollider [24] seems fairly dynamic in its basic language concept, yet its *Just-in-Time programming library* [24, chapter 7] exhibits a different kind of *viscosity* against the dynamic modification of a synthesis graph. In Just-in-Time programming, while there isn't the necessity for reconnection as in ChucK, the modification of a synthesis graph is allowed only at the point where a *proxy object* is utilized. When a modification where a proxy object is not used needs to be made, it can require a considerable degree of recoding. Figure 15 briefly illustrates a typical viscosity problem in Just-in-Time programing; even only to make *c* and *d* in the synthesis graph (on lines

[6] Viscosity is defined as "resistance to change: the cost of making small changes" and it "becomes a problem in opportunistic planning when the user/planner changes the plan" [3].

07 and 08) replaceable, almost the whole code must be rewritten as on lines 17 through 26.

```
01: p = ProxySpace.push // if needed
02:
03: ~a = Lag.ar(LFClipNoise.ar(2 ! 2, 0.5, 0.5), 0.2);
04: (
05: ~b = {
06:   var c,d;
07:   c = Dust.ar(20 ! 2);
08:   d = Decay2.ar(c, 0.01, 0.02, SinOsc.ar(11300));
09:   d + BPF.ar(c * 5, ~a.ar * 3000 + 1000,0.1)
10: };
11: );
12:
13: ~b.play;
14:
15: // the refactored code from above
16:
17: (
18: ~a={
19:  var a;
20:  a = Lag.ar(LFClipNoise.ar(2 ! 2, 0.5, 0.5), 0.2);
21:  BPF.ar(~c.ar * 5, a * 3000 + 1000, 0.1);
22: }
23: );
24: ~c = {Dust.ar(20 ! 2)};
25: ~d = {Decay2.ar(~c.ar,0.01,0.02),SinOsc.ar(11300)};
26: ~b = ~a + ~b;
27:
28: ~b.play;
```

Figure 16. Refactoring a synthesis graph at runtime in SuperCollider [24, p.212].

Impromptu also supports a considerable degree of dynamic modification at the compositional algorithm level, as it is an internal domain-specific language[7] built on LISP, which is highly dynamic. At the sound synthesis level, it depends on Apple's Audio Unit framework and the dynamic modification of the connections between Audio Units is also supported. However, the replacement of *audio units* must involve the removal of existing connections and requires reconnections as in ChucK.

On the contrary, LC adopts the concept of prototype-based programming at both the compositional algorithms and sound synthesis levels. As the connections in a synthesis graph in LC's patch are made between the slots and not between the instances of unit-generators, the replacement of unit-generators can be performed simply by an assignment. The modification of a synthesis graph can be performed quite simply as shown in Figure 2.

4.2 Mostly-strongly-timed programming and other features with respect to time in LC

As already discussed in Section 2.2, in computer music languages designed with the synchronous approach, a time-consuming task can easily lead to the temporary suspension of real-time DSP, as seen in ChucK, LuaAV and the like. However, if the sound synthesis thread (or process) is separated from a thread (or process) that performs compositional algorithms, the synchronization between them will be imprecise and sample-rate accurate timing behavior will be unrealizable in today's computer systems; thus, such languages as SuperCollider or Impromptu fail to provide the sample-rate accurate timing behavior. LC's mostly-strongly-timed programming provides one solution for this problem by extending strongly-timed programing with the explicit switching between synchronous and asynchronous contexts as described in the previous section.

Many computer music languages lack certain desirable features with respect to time. While the designers of Impromptu clearly take such features into consideration and provide the capability for timing constraints, Impromptu does not provide the feature of time-fault tolerance and cannot handle the violation of execution-time constraints. Impromptu's framework to handle execution-time constraints has another significant problem in that it cannot describe the nested execution-time constraints. Moreover, as Impromptu performs sound synthesis in a different thread than threads for compositional algorithms, the timing behavior of Impromptu is not very precise in comparison with other languages designed with the synchronous approach.

On the contrary, LC provides the sample-rate accuracy in timing behavior. Both constraints on start-time and execution-time are performed with the sample-rate accuracy. Start-time constraints will be never violated due to LC's synchronous behavior. By the *within-timeout* statement, LC can handle the violation of execution-timing constraints. Execution-time constraints can be correctly nested.

4.3 The Integration of the objects and library functions/methods for microsound synthesis in LC

As discussed in Section 2.3, LC is not the first language with objects that can directly represent microsounds. The previous works by Bencina (the software design for granular synthesizers), Brandt (Chronic computer music language), and Wang (ChucK's unit-analyzer concept) also discuss the necessity for more appropriate abstractions for microsound synthesis, emphasizing the difference between microsound synthesis techniques and other conventional synthesis techniques that can fit within the unit-generator concept.

Bencina states "granular synthesis differs from many other audio synthesis techniques in that it straddles the boundary between algorithmic event scheduling and polyphonic event synthesis" [2, p.56]. Brandt attributes the difficulty in microsound synthesis programming in unit-generator languages partly to the inaccessibility to the lower-level details, which the unit-generator concept abstracts away[8][3]. Wang *et al.* also state that "the high-level abstractions in the system should expose essential low-level parameters while doing away with syntactic overhead, thereby providing a highly flexible and open framework that can be easily used for a variety of tasks" when discussing the design of ChucK's unit-analyzer concept [23].

---

7 "An internal DSL is a DSL represented within the syntax of a general-purpose language" [9, p.15] and morphs "the host language into a DSL itself – the Lisp tradition is the best example of this" [8].

8 Brandt discusses that "if a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language" in a unit-generator language, in [3, p.4].

LC's microsound synthesis framework is also designed with a similar approach. As shown in the examples in Section 3.3, in LC's programming model, microsound synthesis is described straightforwardly as an algorithmic scheduling of microsound objects. Each sample within a microsound object is directly accessible, while the utility methods are also offered to manipulate samples at once.

The significant difference between LC and these previous works is that LC provides a programming model for real-time interactive computer music languages with more generality; the works by Bencina and by Brandt do not target the design of real-time computer music languages, and Wang's unit-analyzer concept targets only frequency-domain signal processing and analysis. In addition, LC's microsound synthesis framework is also highly independent from the unit-generator concept.

5. CONCLUSIONS

In this paper, we discussed the three issues in today's computer music practices and described how each feature of LC corresponds to them, with code examples and a comparison with other languages. As LC's language design is motivated to contribute to the solutions to the issues discovered in recent creative practices, it can benefit both further research on computer music languages and creative practices, as one design exemplar.

6. FUTURE WORK

As the current version of LC is just a proof-of-concept version, we are currently planning to implement a more efficient version. We are also currently working to provide more detailed publications on each of LC's features.

7. REFERENCES

[1] D. P. Anderson and R. Kiuvila. *"Formula: A programming language for expressive computer music", Computer Vol.24 (7),* 1991

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

[3] A. Blandford and T. Green, *"From tasks to conceptual structures: misfit analysis"*, in *Proc. IHM-HCI Vol. (2)*, 2001

[4] E. Brandt, *Temporal Type Constructors for Computer Music Programming,* Ph.D. thesis, Carnegie Melon University, 2008

[5] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real Time Java and Real Time Posix.* Addison Wesley. 2001

[6] N. Collins *et al.*, *"Live coding in laptop performance", Organised Sound, Vol.8 (3), Cambridge University Press,* 2003.

[7] R. T. Dean, *Oxford handbook of computer music.* Oxford University Press USA. 2009

[8] M. Fowler, *Language workbenches: The killer-app for domain specific languages,* http://www.martinfowler.com/articles/languageWorkbench.html. [Online; accessed on 22/Mar/2014].

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

[10] T. R. Green and A. Blackwell, "*Cognitive dimensions of information artefacts: a tutorial*", in *BCH HCI conference,* 1998.

[11] R. Ierusalimschy, *Programming in Lua, Second Edition.* LUA.ORG, 2006

[12] M. Kaltenbrunner *et al., "Dynamic patches for live musical performance"*, in *Proc. Intl. Conf. New Interfaces for Musical Expression*, 2004.

[13] M. V. Mathews, *et al.*, *The Technology of Computer Music*. MIT press, 1969

[14] H. Nishino and N. Osaka, *"LCSynth: A Strongly-timed Synthesis Language that Integrates Object and Manipulations for Microsounds"* in *Proc. Sound and Music Computing Conference,* 2012.

[15] H. Nishino *et al.*, *"LC: A Strongly-timed Prototype-based Programming Language for Computer Music"*, in *Proc. ICMC,* 2013

[16] H. Nishino *et al., "Unit-generators Considered Harmful (for Microsound Synthesis): A Novel Programming Model for Microsound Synthesis in LCSynth",* in *Proc. ICMC,* 2013

[17] H. Nishino, *"Mostly-strongly-timed Programming"*, in *Proc. ACM SPLASH/OOPSLA,* 2012

[18] M. Puckette, *"FTS: A real-time monitor for multiprocessor music synthesis". Computer music journal*, Vol.15(3), 1991

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

[20] A. Sorensen *et al., "Programming with time: Cyber-physical programming with Impromptu"*, in *Proc. ACM SPLASH/OOPLSA,* 2010

[21] G. Wakefield *et al., "LuaAV: Extensibility and heterogeneity for audiovisual computing", in Proc. the Linux Audio Conference,* 2010.

[22] G. Wang, *The chuck audio programming language. A strongly-timed and on-the-fly environ/mentality.* Ph.D. thesis, Princeton University, 2008.

[23] G. Wang *et al., "Combining analysis and synthesis in the chuck programming language"*, in *Proc. ICMC,* 2007.

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

[25] T. Wishart, *Audible Design.* Orpheus the Pantomime. 1994