LC : A STRONGLY-TIMED PROTOTYPE-BASED PROGRAMMING LANGUAGE FOR COMPUTER MUSIC

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

This paper describes LC, a new computer music programming language currently under development. LC is a strongly-timed prototype-based programming language for live computer music with lightweight concurrency and lexical closure, the design of which takes the emergence of live-coding performance on laptop computers into consideration as a significant design opportunity for a new computer music language.

1. INTRODUCTION

Programming languages designed for computer music have been playing an important role both in research and artistic creation since the early history of computer music and still attract significant interest from both researchers and artists. Both the improvement in computational speed by faster CPUs and the novel programming concepts and paradigms have been strong motivations to develop new computer music languages; faster CPUs have made real-time sound synthesis possible and novel programming concepts/paradigms imported to computer music language design have facilitated the development of computer music systems.

Besides such achievements in computer technology and programming language research, creative practices by computer music artists have also provided significant opportunities for the design and re-design of computer music languages. Among such creative practices, the emergence of *live-coding* casts an interesting question to computer music programming language design as it requires dynamic modification of a computer music program that is already being executed; what computer musicians can explore in live-coding performance can be significantly limited by the extent to which the programming language involved in live-coding can support such dynamism.

In this paper, we describe LC, a strongly-timed prototype-based computer music programming language currently under development. LC is designed as a dynamically-typed programing language with strong-typing and supports lightweight concurrency and lexical closure. Such features are considered to be beneficial to application fields, in which it is desirable to support considerable degree of dynamism, e.g. live-coding and rapid prototyping, so that further artistic explorations by computer music artists can be facilitated.

2. BACKGROUND AND RELATED WORK

2.1. Live-coding and Programming Languages

Sorensen and Gardner describe live-coding as *"a computational arts practice that involves the real-time creation of generative audio-visual software for interactive multimedia performance"* [18]. Figure 1 shows a scene of live-coding.

While live-coding is recently gaining considerable interests in computer music community, the predecessors of live-coding can already be found in 1980s [24]. Such artists as Ron Kuivila [22] and 'the Hub' [4] pioneered such aesthetics of live-coding.

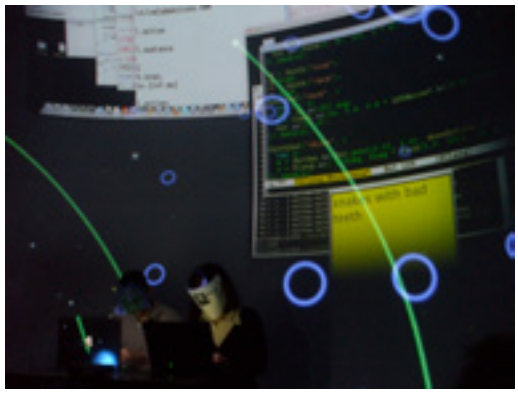

Figure 1. A live-coding performance by Wrongheaded (Matthew Yee-King and Click Nilson). Photo by Dave Griffiths used under CC BY-SA 2.0

The choice of programing environments for live-coding can significantly differ with artists, yet they mostly pick up scripting languages to facilitate live-coding activity. As Collins et al. discus, "*whilst it is perfectly possible to use a cumbersome C compiler, the preferred option for live coding is that of interpreted scripting languages, giving an immediate code and run aesthetic*" [5]; some live-coding practices involve general-purpose scripting languages such as *Lisp/Scheme*, *Perl*, *Ruby*, *Clojure*, together with software modules or external systems for sound rendering. Such examples can be found in [1, 4, 5, 22].

Yet, there also exist many programming languages designed especially for computer music and they are often used in live-coding performance [5]. Among such computer music languages and environments, *ChucK*, *Impromptu* and *SuperCollider* are widely-used for actual live-coding performance. Some artists also use graphical programming languages such as Max/MSP, PureData.

2.2. Related Programming Language Design Issues

In this section, we briefly discuss several topics in programming language design related to the later sections.

2.2.1. *Static-typing vs. dynamic-typing*

It is inevitable in programming language design to decide which type system should be adapted to a new programming language. Statically-typed languages such as Java or C/C++ *"enforce polymorphism based on the structure of the types"* [20, p.123] whereas dynamically-typed languages such as Lua or Ruby, types are evaluated at runtime.

Static-typing benefits its advantages, such as "*detection of programming mistakes"* in type errors, *"more opportunities for compiler optimization"*, *"increased runtime efficiency*" and *"better design time developer experience"* (as seen in auto-completion of method names in smart editors), while dynamic-typing is considered *"ideally suited for prototyping systems with changing or unknown requirements"* and *'indispensable for dealing with truly dynamic program behaviors such as method interception, dynamic loading, mobile code, runtime reflection, etc."* [12]

2.2.2. *Strong-typing and weak-typing*

Another issue in type system in language design is if a language is of *strongly-typed* or *weakly-typed*. A strongly-typed language "*detects when two types are compatible, throwing an error or coercing the types if they are not"* [20, p.91], whereas a weakly-typed language may perform implicit conversion or type cast.

For instance, an expression such as (*"456"* + *7*), a string value *"456"* added to a integer value *7*, will cause an error in Ruby as it is a strongly-typed language while PHP returns 463 by implicitly converting "456" to an integer value 456 since it is weakly-typed. In weakly-typed languages, such a conversion depends on language-specific rules; some other weakly-typed language may return "4567", converting 7 into a string "7".

2.2.3. *Lexical closure*

```
01:function newCounter ()
02:local i = 0
03:-- anonymous function
04:return function ()
05:       i = i + 1
06:       return i
07:     end
08:end
```

```
09:c1 = newCounter()
10:print(c1())--> 1
11:print(c1())--> 2
```

Figure 2. Lexical closure in Lua (taken from [8, p.48])

A programming language is said to have the feature of *lexical scoping*, *"when a function is written enclosed in another function, it has full access to local variables from the enclosing function"*[8, p.47] and if a language treats a function as a first-class value and a function can capture variables in such lexical context, the language has the feature of *lexical closure*. The code in Figure 2 (taken from [8, p.48]) briefly illustrates *lexical closure*. While the variable *'i'* is a local variable in *newCounter()* function, the anonymous function returned from *newCounter()* can still access to the variable *'i'*.

2.2.4. *Prototype-based programming*

Some object-oriented programing (OOP) languages are *class-based* languages. In class-based languages, the concept of class is offered, which is used as a template to create instances as a template. Thus, *"each object is an instance of a specific class"* [8, p.151]. Most class-based languages also offer the concept of *inheritance* and *overriding*, which allows a new class to reuse and replace the code of the existing class. To name a few of the many class-based languages, there exist *Smalltalk*, *Objective-C*, *C++*, *Java* and the like.

On the contrary, some languages do not offer the concept of class at all and "*each object defines its own behavior and has a shape of its own"*; yet, it is still possible to emulate classes by delegation. In such classless languages, *"each object may have a prototype, which is a regular object where the first object looks up any operation that it does not know about"* [8, p.151].

Prototype-based programming is becoming more popular than ever among programmers, possibly because of the success of *JavaScript*. Yet, the origin of prototype-based programming languages is rooted back in 1980s, when the *Self* programming language appeared in 1987 [21]. Many prototype-based programming languages have been developed since then. To name a few among these, there exist *JavaScript*, *Lua*, *Io* [6] and the like.

```
01:obj = {};
02:obj.name = "John";
03:obj.printName =
04:   function (self)
05:     print("My name is " .. self.name .. "!");
06:   end
07:obj.setName =
08:   function (self, newName)
09:      self.name = newName;
10:   end
11:
12:obj:printName();      -- My name is John!
13:obj:setName("Jane");
14:obj:printName();      -- My name is Jane!
```

Figure 3. A prototype-based programming exmaple in Lua

Above Figure 3 is an example of prototype-based programming in Lua. First, we construct a *table* object in line 01. This table object is still empty and has got no field. In line 02, we set the string "*John*" to the field *'name'*. This creates the field in the table *obj* and the value of *obj.name* is set to "*John*". The code does the same between line 04-08, but what assigned to *printName* field and *setName* field are functions; these will be the method attached to *obj* and calling these methods results as seen in line 12–14.

2.2.5. *Delegation*

Instead of inheritance in class-based languages, prototype-based languages normally offer a *delegation* mechanism. Figure 4 is an example in Lua. In this example, two instances are created (*proto* and *obj*). In line 09, the delegation from *obj* to *proto* is set up. This makes the access to a slot of *obj* be forwarded to *proto* when *obj* actually doesn't have the slot. Thus, *obj.hello()* actually calls *proto.hello()*. The access to *obj.one* returns

the value of *proto.one*. After creating a slot *obj.one* in line 17, *obj.one* returns the value retains in the slot, but *proto.one* is completely unchanged. By such a delegation mechanism, prototype-based programming languages can emulate the concept of class.

```
01:proto = {}
02:proto.one      = "one in proto!"
03:proto.hello    = function() print("Hello!") end
04:
05:obj = {}
06:obj.two = "two in obj!";
07:obj.bye = function() print("Bye!") end
08:
09:setmetatable(obj, {__index = proto})
10:
11:obj.hello()             -- “Hello!”
12:obj.bye()               -- “Bye!”
13:
14:print(obj.one)          -- “one in proto!”
15:print(obj.two)          -- “two in obj!”
16:
17:obj.one = "one in obj!"
18:print(obj.one)          -- “one in obj!”
19:print(proto.one)        -- “two in proto!”
```

Figure 4. An example of delegation in Lua

While prototype-based programing languages can emulate the behaviors of class-based languages by delegation, it can support more dynamism compared to class-based languages. For instance, the behaviors of an instance can be modified at runtime without affecting the other instances in prototype-based languages, while changing superclass of one class at runtime are mostly prohibited in class-based languages, as such a change in inheritance tree as it can violate type-safety in class-based languages.

2.2.6. *Duck-typing*

```
01:man = {};
02:man.greet = function(self) print("Hello!"); end
03:
04:dog = {};
05:dog.greet = function(self) print("Bow-wow!"); end
06:
07:f = function(obj) obj:greet() end
08:
09:f(man); -- prints out "Hello!"
10:f(dog); -- prints out "Bow-wow!"
```

Figure 5. An example of duck-typing in Lua

Dynamically typed OOP languages can support *duck-typing*, which *“allows truer polymorphic designs based on what an object can support rather than that object’s inheritance hierarchy”*[20, p.39]. Figure 5 above illustrates a simple example of duck-typing. While these two objects (*man* and *dog*) are totally independent from each other, the code runs without any error as both of them supports *greet()* method.

2.2.7. *Lightweight concurrency*

How to deal with concurrency is one of the significant issues in language design and implementation as there are significant differences in many aspects between native threads (operating systems threads) and green threads (threads provided by a virtual machine, etc.); Native threads are considered *heavyweight*. Creating a new native thread can take considerable amount of time and each native thread can require considerable amount of memory space. For instance, *Red Hat Linux* allocates at least 10MB stack for each native thread by default[1]. Furthermore, aside from the memory usage issue, the maximum number of native threads is also limited by the kernel parameters.

On the contrary, the languages with *lightweight concurrency* can provide such features as fast creation/destruction of threads and less memory usage, usually by implementing threads solely inside a virtual machine without depending on native threads. In [16], Sung et al. report that green threads shows significantly better performance in thread activation and synchronization.

Such features enable a system to manage huge number of threads. For instance, it is reported that *Erlang* could host 20 million *lightweight processes* at once during a benchmark test.[2]

2.2.8. *Strongly-timed programming*

Precise timing behaviour with predictability and repeatability is a significant issue for some applications domain such as *Cyber-Physical-Systems* [9] and computer music systems also share such a concern; for instance, microsound synthesis techniques require sample-rate accuracy in scheduling of short sound particles to render the entire sound output. A failure in scheduling sound particles on time can cause the sound output that is different from theoretically expected and the difference can be audible to human ears.

Not just at such a level of sound synthesis, even at a rhythmic level, *“a pulsation may feel not quite right when there are a few 10s of milliseconds of inaccuracy in the timing from beat to beat”* as Lyon discusses in [10]. However, many computer music programming languages still cannot guarantee such precise timing behaviours.

```
01:// synthesis patch
02:SinOsc foo => dac;
03:
04:// infinite time loop
05:while(true)
06:{
07:  // randomly choose a frequency
08:  Std.rand2f(30, 1000) => foo.freq;
08:  // advance time
09:  100:ms => now;
10:}
```

Figure 6. a ChucK program to generate a sine wave, changing its frequency of oscillation every 100 msec [23, p.44]

Wang’s *ChucK* programming language proposed the concept of *strongly-timed programming*, which is beneficial for such a issue of precise timing behaviour. As a variation of *synchronous programming* [2], it integrates the explicit control of the advance of logical synchronous time into an imperative programming language. Synchronizing the execution of the code with logical time rather than real-time, *ChucK* guarantees the precise timing behaviors in logical time.

[1]Avoiding memory leaks in POSIX thread programming http://www.ibm.com/developerworks/linux/library/l-memory-leaks/index.html

[2] http://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f?dmode=source

Figure 6 is an example of a strongly-timed program in *ChucK* (taken from [23, p.44]). After a sine wave synthesis patch is created (line 02), the main loop keeps on changing its frequency every 100ms. The core concept of strongly-timed programming can be seen in line 09, where a program explicitly advances its logical synchronous time.

2.3. LCSynth

Since the design of LC was originally begun as a control language for LCSynth, a sound synthesis language we have developed [13], this subsection briefly describes LCSynth. LCSynth is a strongly-timed sound synthesis language that integrates objects and manipulations for microsounds. Yet, it is also equipped with the traditional unit-generators [11, 14, p.1234] and the collaboration between these two different abstractions is taken into account in its design.

We also describe the extension we have made to the original version of LCSynth, such as *named parameters* and the support for *multiple inlets/outlets* for unit-generators.

2.3.1. Building a unit-generator graph

```
01:synth SinA
02:{
03:  //build a unit-generator graph.
04:  ugens {
05:    sin:~Sin(freq:440) -> ~DAC();
06:  }
07:}
```

Figure 7. A simple sine wave instrument in LCSynth

Figure 7 illustrates the definition of a simple sine wave oscillator instrument. The definition of SinA starts by *synth* keyword and a sine wave oscillator is connected to sound output between line 04-06, where a unit-generator graph is built. In line 05, '*~Sin*', a sine wave oscillator unit-generator, is given a name '*sin*' so that it can be accessed by the name in the other part of the definition.

Our version of LCSynth supports named parameters. In this example, the constructor of a ~Sin unit-generator is given 440 for the parameter *freq* for initialization.

2.3.2. Implementing control algorithm

The example (Figure 8) extends the example of sine wave oscillator instrument in Figure 7. In this example, the default outlet of a ~Sin unit-generator is connected to the left and right channels of a ~DAC unit-generator (line 05-06). As seen in this statement, both the source inlet and the destination outlets can be explicitly specified by using symbols (*\default, \ch0 \ch1*).

Line 09-21 describes the control algorithm for this instrument. The *synmain()* function is automatically called when an instance of this synth object starts making sound and execution is stopped when the synth objects is killed.

Right after the function starts, the amplitude of a sine wave oscillator output is set to 0.2 in line 12. As in this line, unit-generators inside ugens block can be referred by its given name. Its methods can be called from a *synmain()* function by using the name. The following while loop updates the frequency periodically. In line 19, the logical synchronous time is explicitly advanced by adding a duration value (period::second) to '*now*', a special variable that stands for the current logical time of the system; as a strongly-timed program is based on logical time, the frequency changes exactly every '*period*' seconds with sample-rate accuracy.

```
01:synth SinB
02:{
03:  //build a unit-generator graph.
04:  ugens {
05:    sin:~Sin() -- { \default -> [\ch0, \ch1] }
06:    -> ~DAC();
07:  }
08:  //the main function of this synth object.
09:  synmain(freq, period)
10:  {
11:    //setting the amplitude of the ~sin ugen.
12:    sin.setAmp(0.2);
13:    //main loop. the code below updates
14:    //the frequency with a random value.
15:    while(true){
16:      var freq = Rand(freq, freq * 2.0);
17:      sin.setFreq(freq);
18:      //sleep for 'period' second.
19:      now += period::second;
20:    }
21:  }
22:}
```

Figure 8. A sine wave instrument with periodic change of its frequency

2.3.3. Microsound synthesis in LCSynth

As previously described, LCSynth also provides different abstraction for sound synthesis, the focus of which is on microsound synthesis. The language design includes *Samples*, an object that can contain arbitrary number of samples, and functions/methods that can manipulate *Sample* objects. Figure 9 in the next page describes a sample code of *ascynchronous granular synthesis* [14, p] with an ADSR envelope in LCSynth.

First, a unit-generator graph is built in line 06. A ~Bridge object is connected to an ADSR envelope shaper, then to the DAC output. This ~Bridge object is used to write grains (*Samples* objects) later so to 'bridge' two different abstractions of unit-generators and Samples objects.

In *synmain*, the parameters for the ADSR unit-generator is set up and the envelope is triggered at the beginning of the code. The timing of the ADSR envelope over the entire sound is also calculated. Then in the *while* loop, samples are read from a random location of the buffer no. 0. The pitch is also altered by the argument '*rate*'. This *ReadBuf()* function returns a *Samples* object, which contains 30ms of samples as requested. In line 28, a window of the same size is generated (as a *Samples* object) and then applied to the samples read from the buffer in the following statement. The resulting windowed samples is set to the local variable *grain* and used as a single grain.

As seen in the comment (line 31- 34), if there is no necessity to apply an ADSR envelope to the entire sound, this *grain* can be directly written out to DAC as in line 30 by *WriteDAC()* function and the whole *ugens* block can be removed as there is no necessity to involve unit-generators in sound synthesis.

To apply an ADSR envelope, a ~Bridge unit-generator is used. Calling its *write()* method directly writes the grain onto its internal buffer so to let the ~Bridge unit-generator can stream each sample to the connected ADSR unit-generator. After writing a single grain to the internal buffer, the program sleeps for a random interval until when we schedule the next grain for asynchronous granular synthesis (line 37-44), also checking the timing to release the envelope.

```
01://asynchronous granular synthesis instrument
02://with an ADSR envelope.
03:synth AsyncGranular {
04:   //build a unit generator graph
05:   ugens {
06:     brg:~Bridge() -> env:~ADSR() -> ~DAC();
07:   }
08:
09:   //main function of this synth object.
10:   synmain(dur, a, d, s, r)
11:   {
12:     //set up the parameters and start it .
13:     env.setADSR(a, d, s, r);
14:     env.keyOn();
15:     //calculate the timing to release and
16:     //the timing to stop synthesis.
17:     var relTime  = now + (dur – r);
18:     var stopTime = now + dur;
19:     //loop until the stop time.
20:     while (now < stopTime){
21:       //read a 30ms samples from buffer No.0.
22:       //then apply an hanning window.
23:       var sample = ReadBuf(
24:                      bufno: 0,
25:                      offset:Rand(0,1000)::ms,
26:                      dur:   30::ms,
27:                      rate:  Rand(1,1*2));
28:       var window = GenWindow (size: 30::ms);
29:       var grain  = ApplyEnv  (src:sample,
30:                               env:window);
31:       //write to the delay line; if an ADSR
32:       //envelope is not necessary, use below
33:       //to write grains to directly DAC.
34:       //the whole ugens block can be removed.
35:       //WriteDAC(grain);
36:       brg.write(grain);
37:       //when to schedule the next grain.
38:       var wakeupTime = now + Rand(30, 50)::ms;
39:       //check if the key need to be released.
40:       if (wakeupTime >= relTime){
41:         now = relTime;//sleep until relTime
42:         env.keyOff();
43:       }
44:       //sleep until the wakeupTime.
45:       now = wakeupTime;
46:     }
47:   }
47:}
```

Figure 9. An example of asynchronous granular synthesis with an ADSR envelope in LCSynth

3. LC: A STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC LANGUAGE

We briefly describe the design of LC, a new computer music language currently underdevelopment in this section.

3.1. Necessity for a New Control Language

As seen in these examples in the previous section, LCSynth provides the counterpart entities to the users' conceptualization of microsound synthesis techniques, which are lacking in many computer music languages that only provide unit-generators.

Integration of such counterpart entities in LCSynth is beneficial to remove the *structural misfits* [3] between the representations implemented within the design of the existing computer music languages and the users' conceptualization of microsound synthesis techniques, which are considered to cause significant difficulty in facilitating programing activity.

However, LCSynth itself is not a stand-alone programming language; its target domain is purely sound synthesis domain. Hence, there is a strong necessity to develop a new computer music language that encloses LCSynth as a control language.

No matter whether such a control language is designed as a traditional score/instrument language or as a more general-purpose programming language, it is desirable to design a language that can make the best use of the features available in LCSynth, such as strongly-timed programming, named parameters, seamless access between unit-generator graph and control algorithm in synth object.

Considering such factors, we believe it is better to develop a new computer music language rather than providing LCSynth as a sound synthesis server or a software module. For instance, server-client architecture as seen in SuperCollider does not allow the control languages to work in the same logical synchronous time as in LCSynth and thus fails to provide precise timing behavior in controlling sound synthesis. Implementation as a software module can make seamless access much more difficult between LCSynth and hosting languages; at the same time, there is a concern that the specification of hosting languages may impose significant limitation in programming language, which might be crucial to our target application domain of live computer music.

Thus, it is more desirable to develop a new computer music language that can seamlessly access LCSynth, also equipped with the other features that are beneficial for creative explorations by computer musicians.

For this reason, we developed LC, which fully integrates LCSynth into its language design. LC is designed as a strongly-timed prototype-based language of dynamic and strong typing. LC also has the features of lexical closure and lightweight concurrency.

3.2. Basic Architecture

Unlike some environments like SuperCollider, LC is not a server-client system and both sound synthesis and program execution are performed in logical synchronous time as strongly-timed programs in the same virtual machine. Such a design allows a program to achieve precise timing behaviour with sample-rate accuracy in logical time. LC's VM run bytecode, which is compiled from source code by a separate compiler. Bytecode can be dynamically loaded while any other programs are running; VM immediately executes newly loaded code in the same memory space; this means that programs can share global variables, function definitions, etc.

3.3. Types

LC is a dynamically-typed language, as seen in Figure 10. LC is a strongly-typed, such an operation as *1234 + "567"* (line 04) causes an error in runtime, as the addition between integer and string is not defined. We provide an '..' operator to concatenate such an expression as a string; the expression, *(1234 .. "567")* results in a string *"1234567"*. Yet, *(1234 .. 567)* causes

a runtime error since concatenation operator is not defined for two integer values.

```
01:var a = 1234;  // dynamic-typing
02:var b = "567";
03:
04:var c = a + b; //'integer+string' causes
05:               //a runtime error(strong-typing)
06:var d = a .. b;//use '..' operator, instead.
```

Figure 10. Dynamic-typing and strong-typing in LC

3.4. Playing Sound

```
01:var synA = new SinA(freq:440);
02:
03:synA.start();
04:now += 5::second;
05:synA.kill();
06:
07:var synB = new AsyncGranular();
08:synB.start(dur: 10::second);
```

Figure 11. Instantiating and playing synth objects in LC

Since LCSynth is fully integrated into LC, the synth object definition in LCSynth can be directly written and used in a LC program. Figure 11 above briefly illustrates how a synth object can be played in LC. In the example, the synth object *SinA* is assumed defined as in Figure 7. To play a synth object, it is only required to instantiate a synth object and to call its '*start()*' method. Calling '*kill()*' method can terminate the sound. Optionally as in line 08, the duration of the sound can be specified by passing *dur* argument. After the duration, the underlying scheduler automatically kills the synth object.

3.5. Lexical closure

LC supports lexical closure. Figure 12 below shows a brief example of lexical closure in LC, which is equivalent to an example in Lua (Figure 2). As the current version of LC does not have any syntax sugar for function declaration as seen in Lua[3], an anonymous function is assigned to a variable '*newCounter*' in this example.

```
var newCounter = function(){
  var i = 0;
  return function(){
     i = i + 1;
     return i;
  };
};
```

```
var c1= newCounter();
println(c1()); // 1
println (c1()); // 2
```

Figure 12. Lexical closure in LC

3.6. Prototype-based programming

3.6.1. Table object

Prototype-based programming in LC is supported by *Table* object. This idea to use a Table originally came from Lua. The use of *Table* object in LC in this example is quite similar to the use of *object* in JavaScript.

Figure 13 shows an example of prototype-based programming in LC.

```
01:var newAccount = function() {
02:
03:  var obj = new Table();
04:
05:  var balance = 0;
06:
07:  obj.deposit  = function(amount){
08:     balance = balance + amount;
09:     return balance;
10:  };
11:
12:  obj.withdraw = function(amount){
13:    balance = balance - amount;
14:    return balance;
15:  };
16:
17:  return obj;
18:};
19:
20:var myAccount = newAccount();
21:
22:PrintLn(myAccount.deposit (999) ); //999
23:PrintLn(myAccount.withdraw(10)  ); //989
24:PrintLn(myAccount.deposit (100) ); //1089
25:PrintLn(myAccount.withdraw(999) ); //90
```

Figure 13. Prototype-based programming in LC

3.6.2. Duck-typing

```
01:var man = new Table();
02:man.greet = function() { println "Hello!"); };
03:
04:var dog = new Table();
05:dog.greet = function() { println("Bow-Wow!"); };
06:
07:var f = function(obj) { obj.greet(); };
08:
09:f(man); //prints out "Hello!"
10:f(dog); //prints out "Bow-Wow!
```

Figure 14. Duck-typing in LC

LC supports duck-typing. In Figure 14 above, line 01-05 creates an object. These two objects, *man* and *dog*, are totally independent to each other and are without any relationship between them. Yet, even when they are given to function *f()* as an argument, no error occurs and the program runs as expected, as both of them has the method '*greet()*' .

3.6.3. Delegation

```
01:var table   = new Table();
02:table.hello = function(){ println("Hello!"); };
03:
04:var proto = new Table();
05:proto.bye = function(){ println("Bye!"); };
06:
07:table.prototype = proto;
08:
09:table.hello(); // prints "Hello!"
10:table.bye();   // prints "Bye!"
```

Figure 15. delegation mechanism in LC

Delegation in LC is performed by setting a table to be delegated to *prototype* field as seen in Figure 15 above. When there occurs any access to the fields that do not exist in a table and another table is set to *prototype* field, the access is automatically delegated to the table.

3.7. Lightweight concurrency

LC's thread is implemented as a green thread inside its virtual machine and not dependent on a native thread. Thus, the memory space and the initialization time of LC's threads are much smaller than the native threads provided by the underlying operating system.

At this point, lightweight threads are scheduled deterministically by LC's VM. Such an implementation

[3] In Lua, function definition such as "*function f(a) return a end*" is equivalent to "*local f; f = function(a) return a end*".

can be also seen in *ChucK*. This can guarantee repeatable and predictable timing behaviors in logical synchronous time. This issue of repeatability and predictability in timing behaviors in computer systems is considered significantly import in some applications domain [9] and is also considered favorable for computer music applications [23].

```
01:var f = function(label, message, period, dur){
02:   var stop = now + dur::second;
03:   var timeElapsed = 0;
04:   while(now < stop){
05:     //play a Beep (synth object) for 0.5sec
06:     //and print out a message.
07:     new Beep().start(dur:0.5::second);
08:     println(label .. ":" .. message .. " -- "
09:             .. timeElapsed .. " second(s)");
10:     now += period::second;
11:     timeElapsed += period;
12:   }
13:   PrintLn(label .. ": terminated");
14:};
15:var man = f@("man", "Hello!"   , 2, 8   );
16:var dog = f@("dog", "Bow-wow!", 1, 10 );
17:man.start();
18:dog.start();
```

```
man:Hello! -- 0 second(s)
dog:Bow-wow! -- 0 second(s)
dog:Bow-wow! -- 1 second(s)
man:Hello! -- 2 second(s)
dog:Bow-wow! -- 2 second(s)
dog:Bow-wow! -- 3 second(s)
man:Hello! -- 4 second(s)
dog:Bow-wow! -- 4 second(s)
dog:Bow-wow! -- 5 second(s)
man:Hello! -- 6 second(s)
dog:Bow-wow! -- 6 second(s)
dog:Bow-wow! -- 7 second(s)
man : terminated
dog:Bow-wow! -- 8 second(s)
dog:Bow-wow! -- 9 second(s)
dog : terminated
```

Figure 16. threading in LC, an example code (above) and its output (below)

Figure 16 above is an example of threading in LC. In line 14-15 an '@'(at mark) is placed between a function name and its argument list. This creates a new thread object and initializes it with the given arguments, instead of performing a actual function call immediately. A thread can be started by calling the *start()* method as seen in line 16-17. Thus, LC can create and start threading without involving difficulty.

4. DISCUSSION

In this section, we discuss the benefits of LC's design with a focus on live-coding and rapid prototyping.

4.1.1. Strong-typing

The decision to make LC strongly-typed is largely due to that weakly-typed languages easily cause unforeseeable bugs related to implicit type-casting. For instance, PHP returns *true* for an expression such as "0x0A" == "10" (both of them are string); such a behavior of implicit type-casting can cause *hard mental operation* [7] in program comprehension.

Since it is considered that "*higher levels of software comprehension are associated with higher levels of performance on modification tasks*" [17], such difficulty in software comprehension in weakly-typed languages can also lead to difficulty in software modification, which should be desirable to avoid in live coding performance and prototype-based programming.

4.1.2. Dynamic-typing

While both advocates of static typing and dynamic typing argue its benefits, the recent study by Stuchlik shows that the development speed was significantly faster for most tasks in *Groovy* (dynamically-typed) than in *Java* (statically-typed). Yet, they found no significant difference between them for larger tasks with a higher number of type casts [19]; this justifies LC's design of dynamic-typing as its target domains are in live-coding and rapid-prototyping of a computer music system.

4.1.3. Prototype-based programming and duck-typing

Together with prototype-based programming, duck-typing allowed by such a type system enables a considerable flexibility of dynamism in programming activity, without invalidating the consistency of the code that are already running. Such a tolerance against runtime modification is also quite favorable for LC's target application domain, especially in live-coding.

4.1.4. Strongly-timed programming

The precise timing behavior supported by strongly-timed programming is quite beneficial for computer music. At a synthesis level, many computer music languages lack sample-rate accuracy in task scheduling, which is crucial in microsound synthesis. Together with LCSynth's design to integrate objects and manipulation for microsound, strongly-timed programming in LC can provide more freedom in creative exploration by computer music artists in the microsound domain.

Even at a rhythmic level, many computer music languages still suffer from imprecise timing behaviors. In *Impromptu's* a programming pattern called *temporal recursion* [18], it is required to take care of the difference between the expected ideal timing and the actual timing of when call back functions are invoked so as to keep the rhythm precise.

Although the problem in the precision of scheduling is mainly caused by its underlying framework and operating system, rather than the concept of *temporal recursion* itself, in reality programmers need to involve the same programming pattern very often in such a programming environment; As some language designers argue, the frequent use of the programming patterns may suggest *"flaws in programming language*" [15, p.155].

Such a programming pattern for task-scheduling is not necessary in a strongly-timed program, since it is based on logical synchronous time, which can be explicitly advanced by the program itself.

4.1.5. Lightweight Concurrency

In LCSynth's programming model, each instance of a synth object can run its own thread (of *synmain*). The number of threads created during program execution can be significantly large, as many notes can be created and executed simultaneously in a musical composition.

Furthermore, strongly-timed programming requires fast and precise synchronization of the threads with its internal logical synchronous time when the threads are suspended and resumed, while it is also of significant importance to meet the deadline for sound output.

Such requirements are better supported by lightweight concurrency.

5. CONCLUSION

We developed LC, a dynamic and strongly-typed language with lightweight concurrency and lexical closure, which enclose LCSynth, a sound synthesis language that integrates objects and manipulations for microsound synthesis. Such a language design can provide significant flexibility for runtime modification, which is beneficial for application domains that require considerable degree of dynamism, such as live-coding performance on laptop computers.

6. FUTURE WORK

While LC is designed as a prototype-based language, the enclosed sound synthesis language LCSynth still does not support the dynamic modification of a unit-generator graph. At the same time, its *synmain* function is also just a thread in its nature and it is desirable to re-design LCSynth so that it can be better integrated into LC more seamlessly. We are currently working on this redesign of LCSynth.

7. REFERENCES

[1] Aaron, S. et al. "A principled approach to developing new languages for live coding", *Proc. NIME,* 2011

[2] Berry, G. et al. "Synchronous Programming of Reactive Systems: An Introduction to Esterel." *Programming of Future Generation Computers*, pp.35-55, 1988

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

[4] Brown, C. and Bischoff, J. "Indigenous to the net: Early network music bands in the san francisco bay area.", http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html, accessed on Feb 6th, 2012.

[5] Collins, N. et al. "Live coding in laptop performance", *Organized Sound, Vol.8 (3),* 2003

[6] Dekorte, S. "Io: a small programming language", *Proc. OOPSLA*, 2005

[7] Green T.R.G and Blackwell. A. "Cognitive Dimensions of Information Artefacts: A Tutorial", *BCS HCI Conference*, http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/, 1998.

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

[9] Lee, E. "Computing Needs Time". *Communications of the ACM, Vol.56(5)*, 2009

[10] Lyon, E. "A Sample Accurate Triggering System for Pd and MaxMSP", *Proc. ICMC*, 2006

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

[12] Meijer, E. at Drayton, P. "Static typing where possible, dynamic typing where needed", *Proc. OOPSLA Workshop on Revival of Dynamic Languages, 2004*

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

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

[15] Seibel, P. *Coders at Work: Reflection on Craft o Programming*. Apress, 2009.

[16] Sung, M. et al. "Comparative performance evaluation of Java threads for embedded applications: Linux thread vs. Green thread", *Information Processing Letters, Vol 84(4)*, 2002

[17] Shaft, T.M. and Vessey, I. "The role of cognitive fit in the relationship between software comprehension and modification", *MIS Q. Vol.30 (1),* 2006

[18] Soresen, A. and Gardner, H. "Programming with time: Cyber-physical programming with Impromptu", *in Proc. OOPSLA*, 2010

[19] Stuchlik, A. and Hanenberg, S. "Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time", *Proc. 7th Symposium on Dynamic Languages*, 2011

[20] Tate, B.A, *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*, Pragmatic Bookshelf, 2010

[21] Ungar, D. and Smith, R.B. "SELF:The Power of Simplicity", *Proc. OOPSLA*, 1987

[22] Ward, A. et al. "Live Algorithm Programming and a Temporary Organisation for its Promotion", *READ_ME* of *Software Art and Cultures Conference,* 2004.

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

[24] Zmölnig, I.M and Eckel, G. "*LIVE CODING: AN OVERVIEW"*, *Proc. ICMC,* 2007