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

Hiroki NISHINO  
NUS Graduate School for Integrative Sciences & Engineering,  
National University of Singapore  
gf0901876@nus.edu.sg

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

Ryohei NAKATSU  
IDM Institute,  
National University of Singapore  
elena@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 the researchers and artists. 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.

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 currently under development. 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](image)

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 types”, 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 compatible”, 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 an 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

The choice of programming 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. discuss, “whilst it is perfectly possible to use a cumberose 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.

2.2.4. Prototype-based programming

Some object-oriented programming (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 was invented [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.

![Figure 2. Lexical closure in Lua](image)

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’.

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’.

2.2.5. Delegation

Instead of inheritance in class-based languages, prototype-based languages normally offer a delegation mechanism. Figure 3 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 contain the slot. Thus, obj:hello() actually calls proto:hello(). The access to obj:one returns

![Figure 3. A prototype-based programming example in Lua](image)
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@den dai.ac.jp

Ryohei NAKATSU
IDM Institute, National University of Singapore
elena@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 an opportunity for the design and re-design of computer music languages.

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 Kluijver [22] and ‘the Hub’ [4] pioneered such aesthetics of live-coding.

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 adopted to a new programming language. Statically-typed languages such as Java or C/C++ “force polymorphism based on the structure of types” [20, p.122] 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

There 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 programming (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 [38, 21] was developed. 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.

Figure 3. Prototype-based programming example 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’ of the object. 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 inheriting 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

Figure 1. A live-coding performance by Wrongheaded

(Mattandrewes used under CC BY-SA 2.0)

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 outside 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 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’.

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 outside 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 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’.

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 outside 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 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’.

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 outside 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 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’.
the value of probe.one. After creating a slot probe.one in line 17, probe.one returns the value retains in the slot, but probe.one is completely unchanged. By such a delegation mechanism, prototype-based programing languages can emulate of class.

The on the contrary, the languages with lightweight concurrency can provides 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], Sun 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-timmed programing

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 lead to 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 10ths 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.

2.2.6. Duck-typing

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.

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

2.3. LScynth

Since the design of LC was originally begun as a control language for LScynth, a sound synthesis language we have developed [13], this subsection briefly describes LScynth. LScynth is a strongly-timmed sound synthesis language that integrates objects and manipulations for microsound. Yet, it is also equipped with lightweight thread 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 LScynth, such as named parameters and the support for multiple inlets/outlets for unit-generators.

2.3.1. Building a unit-generator graph

Figure 7 illustrates the definition of a simple sine wave oscillation instrument. The definition of SinA starts by defining its syntax and a wave oscillator is connected to sound output between line 04-06, where a unit-generator graph is built. In line 05, ‘~Sin’, a 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.

Figure 8 illustrates the definition of a simple sine wave oscillation instrument. The definition of SinB starts by defining its syntax and a wave oscillator is connected to sound output between line 04-06, where a unit-generator graph is built. In line 05, ‘~Sin’, a 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 delay 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 (delay, ch0, ch1). Line 09-21 describes the control algorithm for this instrument. The syntax() function is automatically called when an instance of this synth object starts making its function 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 syntax() 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-timmed program is based on logical time, the frequency changes exactly every ‘period’ seconds with sample-rate accuracy.
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 class of class.

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], Sun 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.\footnote{Avoiding memory leaks in POSIX thread programming http://www.dms.developerworks.ibm.com/linux/03-memory-leaks/index.html}

2.2.7. Dynamically typed OOP languages can support man

2.2.8. Strongly-timed programming

Strictly speaking, duck typing is a significant is applicable for some applications domain such as Cyber-Physical-Systems\cite{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 could cause 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 heat to beat” as Lyon discusses in [10]. However, many computer music programming languages still cannot guarantee such precise timing behaviours.

2.2.9. 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 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.\footnote{http://groups.google.com/group/comp.lang.functional/msg/31b7ab2ad0/652724u83d0t-1source} Furthermore, aside from the memory usage issue, the maximum number of native threads is also limited by the kernel parameters.

On the contrary, with lightweight concurrency, it 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], Sun 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.\footnote{Avoiding memory leaks in POSIX thread programming http://www.dms.developerworks.ibm.com/linux/03-memory-leaks/index.html}

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 microsound. Yet, it is also equipped with various unit-generators\cite{11, 14, 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

LCSynth supports named parameters. In this example, the constructor of a ~Sin 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 delay 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 (\texttt{default}, \texttt{ch0}, \texttt{ch1}). Line 09-21 describes the control algorithm for this instrument. The \texttt{sin} function is automatically called when an instance of this synth object starts making its control sequence 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 \texttt{sin} function by using the name. The following while loop updates the frequency periodically. In line 19, the logical time is explicitly advanced by adding a duration value (period=second) to \texttt{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.

Figure 8. A sine wave instrument with periodic change 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 asynchronous granular synthesis\cite{14} with an 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 to ‘bridge’ two different abstractions of unit-generators and Samples objects.

In \texttt{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 \texttt{rate}. This \texttt{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 \texttt{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 \texttt{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 \texttt{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 that 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.

```lua
01: [asyncronous granular synthesis instrument]
02: [use new AsyncGranular instance]
03: synth AsyncGranular {
04: [Only a unit generator graph]
05: upgen {  
06: Bridge { ... } -> env-AsyncGR { ... } -> DAC();
07: }
08: }
09: /*main function of this synth object.
10: main(time, dx, dy, dz, d, r)
11: */
12: {  
13: //set up the parameters and start it.
14: envKey();
15: //the timing to release the envelope.
16: env.time = env + dur - r;
17: //the timing to stop synthesis.
18: var relTime = now + (dur – r);
19: //write to the delay line; if an ADSR
20: //envelope is not necessary, use below
21: if (wakeupTime >= relTime) {
22: now = wakeupTime;
23: //check if the key need to be released.
24: } else {
25: dur = 30::ms,
26: bufno: 0,
27: rate: Rand(l,l*2));
28: //WriteDAC(grain);
29: //to write grains to directly DAC.
30: //envelope to write to the delay line.
31: //to write grains to directly DAC.
32: //delay line is set.
33: //to write grains to directly DAC.
34: //to write grains to directly DAC.
35: //to write grains to directly DAC.
36: //to write grains to directly DAC.
37: //to write grains to directly DAC.
38: //to write grains to directly DAC.
39: //to write grains to directly DAC.
40: //to write grains to directly DAC.
41: //to write grains to directly DAC.
42: //to write grains to directly DAC.
43: //to write grains to directly DAC.
44: //to write grains to directly DAC.
45: //to write grains to directly DAC.
46: //to write grains to directly DAC.
47: }
48: }
```

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 computational 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 both server-client architectures 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 behavior with language 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, which is strongly-typed, such an operation as `1234 + 5672` (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 .. "5672"` results in a string `"12345672"`. Yet, `(1234 .. 5672)` causes a runtime error since concatenation operator is not defined for two integer values.

```lua
01: var newCounter = function() {  
02:  
03:  
04:  
05:  
06:  
07:  
08:  
09:  
10:  
11:  
12:  
13:  
14:  
15:  
16:  
17:  
18:  
19:  
20:  
21:  
22:  
23:  
24:  
25:  
26:  
27:  
28:  
29:  
30:  
31:  
32:  
33:  
34:  
35:  
36:  
37:  
38:  
39:  
40:  
41:  
42:  
43:  
```

```
```
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 stream each sample to the connected DAC. After writing a single grain to the internal buffer, the program sleeps for a random interval until we schedule the next grain for asynchronous granular synthesis (line 37-44), also checking the timing to release the envelope.

```javascript
01://asynchronous granular synthesis instrument 02://with an ADSR envelope. 03:syncAsyncGranular { 04:  //only a unit generator graph 05:  sync:Bridge { -> env:ADSR() -> DAC(); 06:  //main function of this sync object. 07:  sync(on:delay, off: release) { 08:    env.keyOn(); 09:    env.keyOff(); 10:    env.keyOn(); 11:    env.keyOff(); 12:    //set up the parameters and start it. 13:    env.attackTime = delay; 14:    env.sustainTime = 0.5; 15:    env.releaseTime = release; 16:    env.start(); 17:    //the timing to stop synthesis. 18:    while (now < stopTime){ 19:      offset:Rand(0,1000)::ms, 20:      rate:  Rand(l,l*2)); 21:      var window = GenWindow (size: 30::ms); 22:      //the whole ugens block can be removed. 23:      brg.write(grain); 24:      //when to schedule the next grain. 25:      if (wakeTime <= now){ 26:        now = wakeTime; 27:        sleep until waketime. 28:      } 29:      env.keyOff(); 30:    } 31:    //the whole ugens block can be removed. 32:  } 33:}; 34:  } 35:};
```

### 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 conceptual 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 language users [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 programming activity.

#### 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 a logical synchronous time as strongly-timed programs in the same virtual machine. Such a design allows a program to achieve precise timing behavior with language 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. In LC, a strongly-typed, such an operation as `(224 + 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, `(224 .. 567)` results in a string `"224567"`. Yet, `(123 .. 567)` causes a runtime error since concatenation operator is not defined for two integer values.

```javascript
01:var x = 1234; // dynamic-typing 02:var y = "567"; 03:x + y; // "1234567" 04:var f(x, y) = function(x, y) { // integer/float causes 05:return x + y + " \" \" + y; 06}(); 07}();
```

#### 3.4. Playing Sound

#### 3.5. Duck-typing

```javascript
01:var myAccount = new Accoun69; 02:03:myAccount.deposit(1000); 04:56:myAccount.getBalance(); 07:myAccount.withdraw(100); 08:9:myAccount.withdraw(100); 10:1:
```

#### 3.6. Delegation

#### 3.6.1. Table object

Prototype-based programming in LCSynth is supported by Table object. This idea to use a Table originally came from Lua. The use of Table object in LC is in this example is quite similar to the use of object in JavaScript. Figure 13 shows an example of prototype-based programming in LC.

```javascript
01:var myAccount = function() { 02:03:var obj = new Table(); 04:05:var table = new Table(); 06:07:print(table); 08:09:print(myAccount()); 01:20:PrintLn(myAccount.withdraw(999) ); //90 02:21:PrintLn(myAccount.deposit (999) ); //999 03:22:PrintLn(myAccount.deposit(10)  ); //989 04:23:PrintLn(myAccount.deposit(10)  ); //989 05:24:PrintLn(myAccount.deposit(10)  ); //989 06:25:PrintLn(myAccount.deposit(10)  ); //989 07:}
```

#### 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.

#### Figure 15. delegation mechanism in LC
can be also seen in Cloak. 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 important in some applications domain [9] and is also considered favorable for computer music applications [23].

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 dynamically-typed target domains 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 LC’s design to integrate objects and manipulation for microsound, strongly-timed programming in LC can provide more freedom in creative composition by computer music artists in the microsound domain.

Even at a rhythmical level, many computer music languages still suffer from imprecise timing behaviors. In Impromptu’s programming pattern called stopped 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 “flaw in programming language” [15, p.155]. Such a programming pattern for task-scheduling is not necessary in a strongly-timed system, 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 thread object can run its own thread (of synmain). The number of threads created during program execution can be significantly large as many different objects are created and executed simultaneously in a musical composition.

Furthermore, strongly-timed programming requires fast and precise synchronization of the threads within 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 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 strongly-typed-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 redesign LCSynth so that it can be better integrated into LC more seamlessly. We are currently working on this redesign of LCSynth.

7. REFERENCES

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 important in some applications domain [9] and is also considered favorable for computer music applications [23].

Since it is considered that "flaws in programming language" [15, p.155]. 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 dynamically-typed target domains as 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 LC'Synth's design to integrate objects and manipulation for microsound, strongly-timed programming in LC can provide more freedom in creative expression by computer music artists in the microsound domain.

Even at a rhythmic level, many computer music languages still suffer from imprecise timing behaviors. In Inprogsyn's programming paradigm called recursive 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 thread can operate on its own thread (of synmain). The number of threads created during program execution can be significantly large, as many threads are 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 logical concurrency and lexical closure, which enclose LC'Synth, 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 LC'Synth 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 redesign LC'Synth so that it can be better integrated into LC more seamlessly. We are currently working on this redesign of LC'Synth.

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