Performing STFT and ISTFT in the Microsound Synthesis Framework of the LC Computer Music Programming Language

HIROKI NISHINO¹,a) RYOHEI NAKATSU²,b)

Received: July 31, 2015, Accepted: February 8, 2016

Abstract: This paper describes how the short-term Fourier transform (STFT) and inverse short-term Fourier transform (ISTFT) are integrated within the sound synthesis framework of LC, a new computer music programming language, which the authors prototyped, and discusses its benefits for computer music programming. In addition to the traditional unit-generator-based sound synthesis framework, LC provides a framework for microsound synthesis, which is highly independent from the unit-generator concept, and STFT and ISTFT can be also performed within the same framework. While the unit-analyzer concept in the ChucK audio programming language shows a certain degree of similarity to LC’s programming model for STFT and ISTFT, in that both languages allow direct access to low-level spectral data from user programs, due to the dependence on the unit-generator-based sound synthesis framework, a ChucK program that utilizes unit analyzers can exhibit unnecessary complexity in its implementation, when the hop sizes differ among the STFT frames in the program. On the other hand, thanks to the high independence from the unit-generator concept, LC’s microsound synthesis framework can provide a simpler and terser programming model and avoid such unnecessary complications. As other unit-generator languages can also exhibit similar problems as seen in ChucK’s unit analyzers, depending on its sound synthesis framework design, such a language design of LC would be beneficial, not just as a design exemplar for next generation computer music languages, but also to reconsider the design of existing unit-generator languages on such issues regarding how STFT should be integrated in a unit-generator language and whether unit-generators should fully synchronize the audio computation with the advance of global system time.

Keywords: computer music, unit generator, unit analyzer, spectral sound processing, programming language

1. Introduction

The short-term Fourier transform (STFT) is already an essential feature that a computer music language is expected to support. Various STFT-based sound synthesis and processing techniques are involved in today’s creative practices in computer music [3], [18]. Yet, in unit-generator languages, extra care is often required in programming when performing STFT. For instance, in a programming environment that does not provide any useful features to facilitate the implementation of the overlap-add method for FFT frames, users have to make extra effort to write their own code to realize overlapping FFT frames. Furthermore, in many computer music programming languages and environments, spectral processing and feature extraction techniques that utilize STFT are normally encapsulated within low-level built-in modules written in other languages (e.g., C or C++), since the algorithms are often hard to describe just by combining unit-generators.

Some computer music languages try to overcome such deficits by allowing direct access to low-level data within FFT frames. Nyquist [6] is one of the earliest examples of a computer music language of this kind. Such an approach is beneficial in that users can describe desired spectral processing and analysis algorithms within the language itself, without writing built-in modules in another language. While Nyquist is basically designed for non-real-time sound synthesis and analysis, ChucK [18] is an example of a real-time, interactive computer music language that allows direct access to low-level frame data through its unit analyzers [19].

A new interactive real-time computer music language we prototyped, LC [9], [10], is another example designed with a similar idea. LC’s microsound synthesis framework is highly independent of the traditional unit-generator concept and significantly focuses on microsound synthesis (as its name indicates). Unlike most existing computer music languages, LC can perform STFT and ISTFT within this microsound synthesis framework, without involving unit generators and even without the advance of logical time. In addition, LC provides the unit-generator-based sound synthesis framework and the seamless collaboration between these two frameworks with significant different abstractions. Such language design of LC provides simple and terse programming models for spectral processing as well, whereas ChucK and other unit-generator languages may involve a considerable degree of complexity in the resulting code in certain situations, as we describe in later sections.

Generally speaking, as ChucK’s unit analyzers still highly de-
pend on the unit-generator-based sound synthesis framework, the implementation can exhibit a considerable degree of complication, even for simple STFT-related techniques when the hop sizes differ among STFT and ISTFT frames. This problem is largely due to the design of ChucK’s sound synthesis framework, in which all unit generators (and unit analyzers) synchronize the audio computation with the advance of the system’s global logical time. Hence, the same kind of problems may easily occur in other unit-generator languages with similar software framework design; thus, the language design of LC can benefit not just by offering a design exemplar for next generation computer music languages but also by reconsidering the design of unit-generator-based sound synthesis frameworks.

2. Related Work

2.1 STFT in Unit-generator Languages

As previously mentioned, to perform STFT-related techniques in a unit-generator language, extra effort is often required to realize the overlap-add method. Figure 1 describes a typical example in Max to perform STFT [4]. The example involves two fft~ objects to overlap-add two FFT frames*1 and the windowing is realized by multiplying the incoming audio signal by the output signals of one cycle~ object*2 for each FFT frame, respectively. When more FFT frames must be overlapped, more objects must be utilized, and the arguments must be also set up according to the number of overlapping frames. Thus, the implementation can be more complicated when increasing the number of overlapping FFT frames.

Since such tasks (i.e., windowing of incoming samples, buffering of the samples for FFT, and overlap-adding of FFT frames) are normally involved when STFT is performed, many computer music languages provide special objects that encapsulate these tasks to allow users to focus on the implementation of desired algorithms. For instance, Max provides the pfft~ object and SuperCollider [20] offers the FFT/IFFT unit generators, as described in Ref. [19].

2.2 Accessibility of Low-level Audio Data

While such a strategy to provide certain objects to encapsulate common tasks would be quite useful to facilitate the implementation of STFT-related techniques, the actual spectral processing and analysis tasks must be performed after obtaining FFT frame data. Generally, however, “the analysis functionality of these systems relies on pre-made, black-box objects (e.g., coded and imported from C/C++) to meet the needs for common specific analysis tasks” in many unit-generator languages, as Wang et al. discussed in Ref. [19].

Brandt indicated that the unit-generator concept significantly depends on black box abstraction, and considered it a significant problem in computer music language design, since “if a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language” [1, p.4]. As many algorithms for spectral processing and analysis can be complicated and often hardly realizable through just the combination of unit generators, the black-box abstraction nature of the traditional unit-generator concept can be problematic, when a desired operation is not provided as a unit generator, as Brandt discussed.

Thus, to make a computer music language more expressive in spectral processing and analysis, it is desirable to allow direct access to low-level data in each frequency bin and to provide a means to describe a desired algorithm within the language itself. Nyquist[6] is a precursor example (and possibly the first language) that is designed with such a concept. Brandt's Chronic computer music language [1] is another good example, which provides users with accessibility to low-level audio data and the expressiveness to describe desired algorithms. While these two works are of significant interest as design exemplars, these two languages are, however, non-real-time sound synthesis languages and are not designed for real-time interactive computer music.

2.3 ChucK Audio Programming Language and Its Unit-analyzer Concept

The unit-analyzer concept in the ChucK audio programming language [19] provides a good design exemplar for a real-time interactive computer music language with accessibility to low-level spectral data and expressiveness of desired spectral processing algorithms. Yet, before getting into the details of the unit-analyzer concept, we briefly describe ChucK’s strongly-timed programming concept, which realizes the sample-rate accuracy in timing behavior. This programming concept is important in order to understand the code examples of the unit analyzers given in the following sections.

2.3.1 Strongly-timed Programming

Broadly speaking, the strongly-timed programming concept is a variation of synchronous programming, which is based on the ideal synchronous hypothesis. In the ideal synchronous hy-
pothesis, “all the computation and communication is assumed to take zero time (that is, all temporal scopes are executed instantaneously)” and “during implementation, the ideal synchronous hypothesis is interpreted to imply [that] the system must execute fast enough for the effect of the synchronous hypothesis to hold” [2, p.360].

While many other synchronous programming languages are designed for reactive-systems*, ChucK adopts the concept of synchronous programming into an imperative programming language for interactive systems, with a significant focus on precise timing behavior. In strongly-timed programming, a user program can explicitly control the advance of its logical synchronous time, otherwise the logical time will not be advanced at all. In ChucK, the audio computation is performed only when the user program explicitly advances the logical time. Figure 2 describes a simple strongly-timed program example, as Wang described in Ref. [18, p.43]*4.

Such an issue of timing precision can also be important in spectral processing and analysis algorithms. If the audio computation is performed ‘out-of-sync’ with a spectral processing/analysis algorithm, the result of the computation will be different from what users expect.

2.3.2 Unit-analyzer Concept
To support more expressiveness in spectral processing and analysis, ChucK introduces the concept of the unit analyzer, “which carries with it a set of operations and a connection model that resemble but are distinct from those of a unit-generator” [19]. The unit-analyzer concept resembles the unit-generator concept in that it follows the similar connection model, and a unit analyzer (UAna, plural UAnae) “defines a set of control parameters and can be dynamically patched with other UAnae and UGens”. The difference is that unit analyzers “pass generic data that may be in the form of spectral frames, feature vectors, metadata, or any other (intermediate) products of analysis”[19] whereas unit

*3 Reactive systems are “computer systems that continuously react to their environment at a speed determined by this environment”, while interactive systems “continuously interact with their environment, but at their own rate”[8].

*4 An approach similar to synchronous programming can be seen in other computer music programming languages. For example, LuaAV [17] utilizes coroutines in its sound synthesis framework to realize synchronous behavior. For another example, PureData (Pd)[14] also adopts a synchronous approach. In PureData, “audio and message processing are interleaved in Pd” [13]. In other words, the audio computation is blocked until the system finishes processing all the events at the timing. It should be noted, however, that strongly-timed programming is a programming concept based on the ideal synchronous hypothesis, while these languages realize similar behavior by implementation.

Fig. 2 Strongly-timed programming example in ChucK [18].

```plaintext
01: // load the sound files to the buffer objects.
02: "sound/violin.wav" => string filename1;
03: "sound/cherokee.aif" => string filename2;
04: SndBuf src1;
05: SndBuf src2;
06: filename1 => src1.read;
07: filename2 => src2.read;
08: ...
09: // build a synthesis graph.
10: src1 => FFT ifft1 => blackhole;
11: src2 => FFT ifft2 => blackhole;
12: ifft ifft => dac;
13: 800 => ifft.gain; // gain for ifft.
14: ...
15: // set up FFT parameters.
16: 1024 => fft1.size => fft2.size => ifft.size
17: => int FFT_SIZE;
18: FFT_SIZE / 2 => int HOP_SIZE;
19: ...
20: Windowing.hann(FFT_SIZE) => fft1.window;
21: Windowing.hann(FFT_SIZE) => fft2.window;
22: Windowing.hann(FFT_SIZE) => ifft.window;
23: ...
24: // to store the cross synthesis result.
25: complex Z(FFT_SIZE / 2);
26: ...
27: // main loop.
28: while(true){
29: // perform FFT for two inputs.
30: fft1.upchuck();
31: fft2.upchuck();
32: ...
33: // cross synthesis
34: for (i => int; i < fft1.size() / 2; i++){
35: fft1.cval(i) $ polar => polar a;
36: fft2.cval(1) $ polar => polar b;
37: M(a.mag * b.mag, a.phase) => polar c;
38: c $ complex => Z[i];
39: }
40: ...
41: // perform IFFT.
42: ifft.transform(2);
43: ...
44: // sleep until the next frame.
45: HOP_SIZE := samp => now;
46: }
```

Fig. 3 Cross synthesis example in ChucK.

ChucK’s unit analyzers are designed to work within the unit-generator-based sound synthesis framework. One of the benefits of the unit-analyzer concept is that the various types of data can be obtained from the unit-generator outputs and even low-level data are made directly accessible from a user program. For instance, in the Fig. 3 example of cross synthesis[16] in ChucK, the FFT unit analyzers provide direct access to each frequency bin (lines 35–36) and the algorithm of cross synthesis can be performed only by the user code. Unlike many other unit-generator languages, there is no necessity to involve a unit-generator object designed for cross synthesis.

While ChucK is not the first language that allows the direct access to low-level data as already described earlier in this article, such a design of the unit analyzers contributes to avoiding the problem, as Brandt discussed. Even if a desired operation is not present, users can realize it by writing their own code in ChucK.

Yet, since ChucK’s unit-analyzer concept significantly depends on its unit-generator-based sound synthesis framework, such dependency can lead to usability difficulties when the hop sizes differ among the FFT and IFFT frames, as described in detail in the later section.
3. STFT in the Microsound Synthesis Framework of LC

3.1 Microsound Synthesis Framework of LC

Before we describe how STFT can be performed in LC, we provide a brief overview of its microsound synthesis framework, in which STFT modules are integrated. While LC is a computer music programming language, first developed as a host language for the LCSynth sound synthesis language [11], [12], significant extension has been made to its language specification until the current version*5.

While LC is still equipped with the unit-generator-based sound synthesis framework, it also offers the microsound synthesis framework, which is highly independent of the unit-generator concept, together with the seamless collaboration mechanism between these two different sound synthesis frameworks.

In LC, there are two objects utilized to represent microsounds: Samples and SampleBuffer objects. Both of these objects can contain an arbitrary number of sample values (as long as enough memory can be allocated in runtime). The difference between Samples and SampleBuffer is that the former is an immutable object and the latter is mutable*6.

Figure 4 briefly describes how Samples and SampleBuffer can be created. Both objects have various methods for manipulations. As Samples is immutable, calling these methods return new Sample objects, and the original object will remain unchanged. Samples and SampleBuffer are mutually convertible by the toSampleBuffer and toSamples methods, respectively.

Figure 5 describes an example of indexed access. Each sample within these objects is directly accessible by the [''] operator. While Samples is immutable, the samples within SampleBuffer can be changed by assigning a new value by indexed access.

Figures 6 and 7 describe examples of synchronous granular synthesis*7 and pitch shifting by granulation [15, p.197], respectively. As shown in these, LC can perform microsound synthesis without involving unit generators at all (as on lines 01–28 in Fig. 7), while it is also possible to generate a Samples object from a unit generator or a patch (lines 15–16 and lines 22–27 in Fig. 4, respectively) and to give a Samples object to a unit generator or a patch as the input signal to its inlet (lines 30–51 in Fig. 7). It should be noted that LC is a strongly-timed language with sample-rate accurate timing behavior (line 13 in Fig. 6 and line 49 in Fig. 7).

3.2 Short-term Fourier Transform in LC

As described in Fig. 3, Samples and Samples objects in LC allow direct access to low-level sample data. Such a feature of LC’s microsound synthesis framework can also be beneficial to perform STFT. While the current version of LC offers only basic STFT-related library functions as shown in Table 1, this accessibility to low-level data allows users to describe their desired operations within just LC, similarly to ChucK’s unit analyzers.

Figures 8 and 10 describe simple examples of cross synthesis and time-stretching by phase vocoding, respectively. While the code uses the mul method of Samples to multiply the magnitudes

---

*5 Our paper [10] gives a brief overview of the language features of LC.

*6 This difference between Samples and SampleBuffer in LC is similar to the difference between String and StringBuffer in Java [7].

*7 In synchronous granular synthesis, the sound “results from one or more streams of grain” and “the grains follow each other at regular intervals” [15, p.93].
Fig. 7 Granular pitch-shifting example in LC.

Table 1 Library functions for STFT in LC.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT(var samples)</td>
<td>FFT (The current version just perform real FFT only) and Inverse Fast Fourier Transform. FFT returns an array of [real, imag].</td>
</tr>
<tr>
<td>IFFT(var real, imag)</td>
<td>Inverse FFT.</td>
</tr>
<tr>
<td>CarToPol(var real, imag)</td>
<td>Conversion between Cartesian coordinate and polar coordinate.</td>
</tr>
<tr>
<td>PolToCar(var mag, phase)</td>
<td>Conversion between polar coordinate and Cartesian coordinate.</td>
</tr>
<tr>
<td>PFFT(var samples, window)</td>
<td>PFFT/IFFT with windowing. PFFT/IFFT returns the frame data in polar coordinate. PFFT returns an array of [mag, phase].</td>
</tr>
</tbody>
</table>

Fig. 8 Cross synthesis example in LC.

Fig. 9 Performing cross-synthesis using a for loop.

Fig. 10 Phase vocoder example (time-stretching) in LC.
synthesis and analysis. Moreover, in LC, STFT can be performed within just its microsound synthesis framework without involving any unit generator, whereas ChucK’s unit analyzers still significantly depend on its unit-generator-based sound synthesis framework. Such independence from the unit-generator concept is one of the key factors in the language design of LC that makes it possible to avoid unnecessary complexity in the implementation (in other words, usability difficulties in programming), which ChucK and other unit-generator languages with similar designs exhibit in certain situations. We discuss this issue in more detail in the next section.

4. Discussion

One of the typical problems in performing STFT within the unit-generator-based sound synthesis framework is the difficulty in implementing windowing and overlap-adding. The Fig. 1 example in Max exhibits this typical problem; windowing must be implemented and multiple FFT objects must be utilized to implement overlapping FFT frames. Furthermore, the time must be advanced to feed the input samples. This can introduce latency, even when all the sample data is already loaded in the buffer.

While the problems as above (except the latency) can be solved by providing a utility object that automatically performs buffering, windowing and overlap-adding all at once, (e.g., pfft~ in Max and the FFT unit generator in SuperCollider), there remains a serious problem of the expressiveness of the related algorithms and accessibility to lower-level data, as Brandt and Wang discussed, which is described in Section 2. Based on the idea similar to Nyquist and Chronic, both of which are non-real-time computer music languages, ChucK’s unit-analyzer concept seems to succeed in providing one solution for an interactive real-time computer music language by allowing direct access to low-level spectral data. Strongly-timed programming in ChucK also provides sample-rate accurate synchronization between the algorithms written within the language and the unit-generator-based sound synthesis. Thus, ChucK achieves a considerable degree of expressiveness in spectral processing and analysis; however, as it is highly dependent on the unit-generator-based sound synthesis framework, the latency issue remains. As it is still required to advance the logical time to feed the input samples to the FFT unit analyzer, the latency of the FFT frame size must be involved.

To make matters worse, the dependency on the unit-generator-based framework can cause another problem in ChucK when the hop sizes can differ among the FFT frames. Think of the situation in which two FFT objects are utilized in cross synthesis and the hops sizes differ between them. For instance, one may want to use a smaller hop size for one of the sound sources (from which we extract its formant for cross synthesis), as the duration of the sound source is shorter than the other. Figure 11 describes such a ChucK example in which the hop sizes for two input sources differ. As presented, the code is much more complicated than the Fig. 3 example.

To comprehend this example better, consider the situations as follows. First, we discuss what is performed in the previous example in Fig. 3. In Fig. 3, if the time is advanced for 512 samples (the hop size for IFFT), both the FFT unit analyzers (fft1 and fft2) discard the oldest 512 samples in the internal buffers and receive the input of 512 new samples from their sound sources (src1 and src2). Thus, the hop sizes for these FFT unit analyzers become the same as IFFT.

Now, suppose that we want to read the second sound source at half speed. To implement such a program, one has to change the hop size for fft2 to 256 samples but still keep the hop sizes for fft1 and ifft at 512 samples. In this case, fft2 must discard only the oldest 256 samples and accept 256 new input samples instead, but fft1 and ifft still require the advance of 512 samples. For this reason, one must utilize two pairs of SndBuf and FFT for the second sound source (source2a and fft2a, source2b and fft2b) as in the Fig. 11 example, and then update the reading positions of each SndBuf so that the samples can be fed from the correct positions of the buffers, according to the given hop sizes. This problem is indeed rooted in the design of ChucK’s unit-generator-based sound synthesis framework, in which all the unit generators and unit analyzers synchronize their audio computations with one global system time. Hence, a similar problem can also be exhibited in other unit-generator languages, if they are similarly designed.

It may be possible to describe the same algorithm just by the combination of unit generators. For instance, the algorithm to update the read position may be implemented by a phasor unit generator. The reading position from the sound buffer can be given as the sum of the output from a phasor unit-generator that ranges from zero to 1,023 and the offset value, which represents the starting position to read from the buffer for the current frame. This offset value must be updated at the beginning of each frame. The resulting code for the synthesis graph, however, would be more complicated and less comprehensive for users.

Another possible solution for this problem, which is a little simpler, is to control the number of the samples to be fed into the FFT unit analyzer for the second source by temporarily disconnecting a connection between the FFT unit analyzer and a blackhole unit generator. In ChucK, a blackhole unit generator, which simply discards any given input, is a unit-generator that can be the starting node of a depth-first-search for the audio computation, as well as the dac unit generator. A unit generator (or unit analyzer) does not produce its output, if a unit generator (or unit analyzer) is not traceable from any dac or blackhole unit generator in ChucK’s sound synthesis framework.

Thus, one can control the amount of the samples to be fed into an FFT unit analyzer by disconnecting it from a blackhole unit generator and can temporally suspend its audio computation. Figure 12 describes such an example. The example first advances the time only for the hop size for fft2 and then immediately disconnects the connection between fft2 and blackhole to deactivate fft2. After this disconnection, no more samples are fed from source2 to fft2. Then, the code advances the time until the next timing when cross synthesis must be performed. Right after waking up from sleep, the code connects fft2 again to blackhole so that it can be fed the input samples. In this way, modifying the synthesis graph in this example controls the hop size for fft2. As clearly seen, the example does not utilize an additional pair of SndBuf and FFT and is much simpler than the Fig. 11 example.
Another example of cross synthesis in ChucK.

However, aside from whether or not other unit-generator languages allow direct access to low-level data from algorithms that users describe themselves, as in Chuck’s unit analyzers, the solution in the Fig. 12 example may be more specific to ChucK than the Fig. 11 example since some unit-generator languages may not support the dynamic modification of a sound synthesis graph as required in Fig. 12. Moreover, even in the language that allows the dynamic modification of a unit-generator graph, the disconnection and reconnection may not be performed with sample-rate accuracy, failing to modify the synthesis graph at the expected timing. Furthermore, both examples can be even more complicated when the hop sizes for all the FFT/IFFT frames differ. In

![Fig. 11 Another example of cross synthesis in ChucK.](image1)

![Fig. 12 Yet another example of cross synthesis in ChucK.](image2)
the Fig. 11 example, one would need to add more pairs of SndBaf
unit generators and FFT unit analyzers. In Fig. 12, more dynamic
modification of the synthesis graphs must be involved.

In contrast, the Fig. 8 example in LC can simply use the dif-
ferent hop sizes by changing the parameters ovlp1 and ovlp2 on
line 07. Moreover, no latency is involved, as it is not neces-
sary to advance the time to feed the samples to an FFT object,
as in ChucK or other unit-generator languages. Thus, the inde-
pendence of LC’s microsound synthesis framework from the unit-
generator concept plays a significant role in avoiding unnecessary
complexity in these situations. Furthermore, since LC’s unit gen-
erators and patches can compute the output samples even without
the advance of time (lines 14–15 and lines19–25 in Fig. 4), even
when one of the source sounds is generated by a unit generator
or a patch, there would not be much difference in complexity. A
Samples object obtained from the output of a unit generator
or patch can be directly used for spectral processing.

In addition, LC’s microsound synthesis framework may also
be suggestive to the sound synthesis framework design for unit-
generator languages. The complexity in using different hop sizes
among FFT frames is largely due to the software design in which
unit generators fully synchronize their behaviors with one global
system time. Yet, this issue has not been discussed in the design
of existing computer music languages and has been neglected,
while the use of different hop sizes among FFT frames can be
important when computer musicians creatively explore spectral
synthesis and analysis techniques. As described in this article,
even to utilize a simple time-stretching technique for the formant
source in cross synthesis, hop sizes can differ among FFT frames
and then lead to complexity in the resulting code. The code can be
more complicated when applying various spectral synthesis and
analysis techniques at once. As this issue can damage the expres-
siveness of a computer music programming language and hinder
creative exploration by computer musicians, it should be re-
viewed as a usability difficulty in computer music programming.

5. Conclusion and Future Work

We described how STFT can be performed in LC within its
microsound synthesis framework and compared it to ChucK’s
unit-analyzer concept, which shows a certain degree of similarity.
Both languages provide accessibility to low-level data and the ex-
pressiveness to describe desired operations on the low-level data
within just the language, without using dedicated built-in objects.
Yet, due to the dependence on the unit-generator-based sound
synthesis framework of ChucK’s unit-analyzer concept, these two
languages can show significant differences in the complexity of
the resulting implementation, even for a simple spectral process-
ing technique in certain situations. It was observed that ChucK
cannot exhibit more complexity in the resulting code when the hop
sizes differ among FFT frames.

As this complexity is not specific to ChucK but is caused by the
design of the sound synthesis framework that fully synchronizes
the audio computation of unit generators with only one global
system time, the same problem can also be exhibited in other
unit-generator languages. Thus, LC’s sound synthesis framework
design not only provides a design exemplar for further research
in computer music language design but also is beneficial for re-
considering the design of existing unit-generator languages.

For the support of better usability and expressiveness in com-
puter music languages, it would be desirable to consider the fol-
lowing issues for further discussion, which we described by com-
paring LC’s microsound synthesis framework and ChucK’s unit-
analyzer concept: how STFT should be integrated in computer
music programming language and whether unit generators should
synchronize the audio computation completely with the advance
of global system time.

For future work, since the implementation of LC is still just a
proof-of-concept prototype and the number of library functions
is limited at this point, we are planning to provide more STFT-
related library functions and to investigate how other STFT-
related techniques can be implemented in LC and other languages
for further inquiry in computer music language design.

References

[1] Brandt, E.: Temporal Type Constructors for Computer Music Pro-
Languages: Ada 95, Real Time Java and Real Time POSIX, Addison-
Wesley (2001).
fft~, MSP Tutorial.
Composition and Sound Synthesis, Computer Music Journal, Vol.21,
No.3 (1997).
sional (2000).
Springer-Verlag (2010).
Music Programming Language that Integrates Objects and Manipu-
lations for Microsound Synthesis, Ph.D. thesis, National University
of Singapore (2014).
sis Framework in the LC Computer Programming Language,
Language that Integrates Objects and Manipulations for Microsounds,
Harmful (for Microsound Synthesis): A Novel Programming Model
[16] Settel, Z. and Lippe, C.: Real-time Timbral Transformation: FFT-
[17] Wakefield, G. et al.: LuaAV: Extensibility and Heterogeneity for Au-
timed and on-the-fly Environmetalitiy, Ph.D. thesis. Princeton Uni-
versity (2008).
[19] Wang, G et al.: Combining Analysis and Synthesis in the ChucK pro-
Hiroki Nishino received his Ph.D. degree in Integrative Sciences and Engineering from the National University of Singapore in 2014. He also received the Grant for Overseas Study by Young Artists Pola Art Foundation in 2008 and the MITOH funding from Information-Technology Promotion Agency Japan in 2009.

Ryohei Nakatsu received his Ph.D. degree in electronic engineering from Kyoto University in 1982. Since then he has been working in the area of communication technologies, art and technology, robotics, etc. He is a life fellow of IEEE, a fellow of the Institute of Electronics, Information and Communication Engineers, a fellow of the Virtual Reality Society of Japan.