Mostly-Strongly-Timed Programming in LC

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

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

ABSTRACT

Due to its synchronous behaviour, a strongly-timed program can suffer from the temporary suspension of real-time DSP in the presence of a time-consuming task. In this paper, we propose *mostly-strongly-timed programming*, which extends strongly-timed programming with the explicit switch between synchronous context and asynchronous context. If a thread is in asynchronous context, the underlying scheduler is allowed to preempt it without the explicit advance of logical time. Time-consuming tasks can be executed asynchronously, without causing the temporary suspension of real-time DSP. We also discuss how the concept is integrated in LC, a new computer music programming language we prototyped, together with the discussion on implementation issues.

1. INTRODUCTION

The issue of timing precision is a traditional topic in computer music. Even today, when the advance of computer technology has made significant improvements in both computational speed and communication bandwidth with the external hardware, timing precision continues to be a topic of significant interest. When performing microsound synthesis techniques [16], sample-rate accuracy for scheduling microsounds is a crucial factor in rendering the output as theoretically expected.

The *strongly-timed programming* concept that Wang *et al.* proposed in the ChucK audio programming language [21] is interesting in that it contextualizes such a problem as an issue with the programming language design. It adopts the concept of *synchronous programming* [10] to an imperative programming language for interactive systems, by letting a user program explicitly control the advance of logical time. In this manner, even sample-rate accurate precise timing behaviour can be realized.

However, due to its synchronous behaviour, a strongly-timed program can suffer from the temporary suspension of real-time DSP in the presence of a time-consuming task. Such a problem can occur in any other computer music languages and systems, which take a similar synchronous approach in the design.

In this paper, we propose a new programming concept, *mostly-strongly-timed programming* [1], which extends strongly-timed programming with explicit switching between synchronous context and asynchronous context. The underlying scheduler can suspend threads in asynchronous context at an arbitrary time, even without the explicit advance of logical time; thus, the temporary suspension of real-time DSP can be avoided by enclosing the time-consuming part of a task in asynchronous context.

We adopted this concept into LC [12, 13], a new computer music programming language we prototyped. In the following sections, we briefly review the related works and describe the mostly-strongly-timed programming concept with code examples in LC, followed by discussions on the concept, together with implementation issues.

2. RELATED WORKS

2.1 The Earlier Live Computer Music Systems

For non real-time computer music languages, the issue of timing precision was not a concern in the early days of computer music, as even the sample-rate accuracy was easily achieved simply by setting the *audio-rate* and *control-rate* [7, p.468] to the same settings. However, precise timing behaviour in a computer music system became an issue of significant interest, soon after the emergence of live computer music. In the era when a computer music system still consisted of a computer and its external synthesizer hardware, the concerns were concentrated around how to deal with the limitations that comes from slow CPUs and the low bandwidth of the hardware interface of the time.

FORMULA by Anderson and Kuiliva [1, 2] is one of the most notable works in that it represents efforts made in this era. FORMULA uses the time-sliced approach inspired by *discrete-event simulation* [4], and the tasks in FORMULA are performed in system-internal logical time. By performing tasks in logical-time, the events can be given the same logical timestamps, as if they were generated or scheduled at the same time, regardless of the actual timing in real time when they are generated or scheduled. Together with the mechanism to buffer the output events, FORMULA achieved desirable timing precision for live computer music in this era.

[1] A very early discussion on mostly-strongly-timed programming is presented in [14].

2.2 Stand-Alone Live Computer Music Systems

After stand-alone real-time sound synthesis is made possible on a personal computer, the issue of timing precision was again raised. As computer music systems must process the compositional algorithms and real-time DSP simultaneously, the software design for such a computer music system had to be investigated. The popularization of microsound synthesis techniques also led to the demand for more precise timing behaviour.

2.2.1 The separation between audio computation thread and compositional algorithms thread(s)

One of the approaches taken in the design of computer music systems is to perform the real-time sound synthesis in a separate process (or a separate thread) with higher priority and perform compositional algorithms in other processes (or threads). This approach is still frequently seen in many computer languages. For example, SuperColllider [22] consists of two processes, *scserver* (the sound synthesis server) and *sclang* (the interpreter for its programming languages). *Impromptu* [19] also performs sound synthesis in a different thread than the compositional algorithms. The sound synthesis software frameworks and libraries are also frequently designed with the same approach. For example, both *Jsyn* for Java [8] and *CsoundXO* for python [11] are designed in this manner.

This approach can avoid the suspension of real-time sound synthesis, as all the compositional algorithms, including a time-consuming compositional task, are performed in a different thread/process. Instead, this makes it significantly harder to synchronize sound synthesis with the compositional algorithms. Generally speaking, the synchronization between threads and processes in today's operating systems are not so fine-grained to realize sample-rate accurate timing precision in such a software design.

2.2.2 The synchronous approach

To achieve better timing precision, many computer music languages and systems take the synchronous approach, which is based on the *ideal synchronous hypothesis.* In the ideal synchronous hypothesis, "all the computations are assumed to take zero time (that is, all temporal scopes are executed instantaneously)" *and* "during implementation, the ideal synchronous hypothesis is interpreted to imply the system must execute fast enough for the effect of the synchronous hypothesis to hold" [9, p.360].

```
01: -- define a function to print a message
02: -- repeatedly, every 1 second.
03: function printer(message)
04:   while true do
05:     print(message)
06:     wait(1) -- wait 1 second
07:   end
08: end
09: -- start ticking:
10: go(printer "tick")
11: -- start tocking after 0.5second:
12: go(0.5, printer "tock")
```

Figure 1. A simple example of a LuaAV program [20].

In practice, when designing a real-time computer music system, the ideal synchronous hypothesis is interpreted to imply that real-time DSP must be blocked until the system finishes processing all the scheduled tasks and the system must execute all the tasks before the deadline for the next DSP cycle. To achieve such behaviour, the execution of the compositional algorithms and the audio computation are normally interleaved in one thread.

While many widely-used languages are implemented with this synchronous approach[2], LuaAV [20] provides an interesting design exemplar for textual computer music languages, in that it utilizes collaborative (or non-preemptive) multi-tasking by coroutines to achieve synchronous behaviour. Figure 1 describes a simple LuaAV example [20]. In LuaAV, the user code is executed as a coroutine within the software framework. By calling the *wait* function[3] (as seen on line 06), the current coroutine explicitly yields so that the underlying sound synthesis framework can perform the audio computation.
After the given duration has passed, it resumes the coroutine. The *go* function calls are made to execute new coroutines on line 10 and line 12.

As coroutines can yield and resume much faster than native threads, it is easy to realize the fine-grained synchronization and synchronous behaviour between the compositional algorithms written as coroutines and sound synthesis, when performing both in the same audio computation thread.

2.2.3 Strongly-timed programming

The strongly-timed programming concept proposed by Wang *et al.* in the ChucK audio programming language [21] is also of significant interest in that it clearly puts this issue of precise timing in the context of the programming language concept. While most synchronous programming languages are designed for *reactive systems*[4], ChucK targets interactive systems[5], with an exclusive focus on audio programming. As a variation of synchronous programming, ChucK integrates the explicit advance of logical synchronous time within an imperative programming language.

Figure 2 illustrates a simple strongly-timed program in ChucK [21, p.43]. As seen on line 10, logical time is explicitly advanced by a user program. The audio output is computed only when logical time is advanced[6]; if there exists any active thread that is still being executed, the audio computation is blocked.

[2] For instance, "audio and message processing are interleaved in Pd" [15].
[3] The *wait* function can also wait for a certain event to occur.
[4] Reactive systems are "computer systems that continuously react to their environment at a speed determined by this environment"[10].
[5] Interactive systems are computer systems that "continuously interact with their environment, but at their own rate" [10].
[6] Similar to LuaAV, a ChucK program can wait for a certain event; the thread of execution can be suspended and logical-time can be advanced until the occurrence of the event.

While there may be a certain degree of similarity between the Figure 1 LuaAV example and the Figure 2 ChucK example, it should be emphasized that strongly-timed programming itself is not directly associated to any particular implementation strategy; while the concept may be implemented by translating a strongly-timed program to another program that utilizes coroutines, which would look similar to the LuaAV example, it is also possible to implement a virtual machine that executes the bytecode generated by its own compiler as in ChucK. There can be various implementations of a strongly-timed programming language, as it is purely a programming concept.

```
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;
09:   // advance time
10:   100::ms => now;
11: }
```

Figure 2. A simple strongly-timed program in ChucK [21, p.43].

3. MOSTLY-STRONGLY-TIMED PROGRAMMING

3.1 Extending the strongly-timed programming concept with asynchronous/preemptive behaviour

While the synchronous approach can realize fine-grained timing precision, due to the underlying ideal synchronous hypothesis, a time-consuming task can temporarily suspend real-time DSP, since audio computation is blocked until the task is finished. Such a situation is clearly not desirable in live computer music.

The proposition of strongly-timed programming implies that this problem can be considered as a problem of the programming concept applied to the language, not just as an implementation issue. One of the possible solutions suggested from this perspective is to extend the strongly-timed programming concept with asynchronous behaviour.

3.2 The mostly-strongly-timed programming concept

Based on the idea described above, we propose *mostly-strongly-timed programming*, which extends strongly-timed programming with the explicit switch between the synchronous/non-preemptive context and the asynchronous/preemptive context. In a mostly-strongly-timed program, a thread in the former context is executed synchronously as it is in a strongly-timed program and audio computation is blocked until the thread explicitly advances logical time or waits for an event. On the contrary, in the latter context, the underlying scheduler is allowed to suspend and resume a thread at any arbitrary time if necessary.

In LC, two statements, ***sync*** and ***async***, are provided for explicit context switching. These statements switch the current context to the synchronous context and to the asynchronous/preemptive context respectively. Figure 3 describes an example of mostly-strongly-timed programming in LC. As shown, the ***sync*** and ***async*** statements can be nested as desired. As seen in the comments, the time-consuming part of a thread can be preempted when necessary, just by enclosing it within an ***async*** block; thus, temporary suspension of real-time DSP can be avoided.

```
01: //create/play a sine wave oscillator patch to
02: //make the temporary suspension of DSP audible.
03: var p = patch {
04:   Sin~(440) => DAC~();
05: };
06: p->start();
07:
08: //loading 16 large sound files at once from the
09: //hard drive. This can consume lots of time and
10: //temporarily suspend real-time DSP.
11: for (var i = 0; i < 16; i+= 1){
12:   //load sample0.wav- sample15.wav
13:   LoadSndFile(i, "sample" .. i .. ".wav" );
14: }
15:
16: //this infinite loop suspends the DSP forever;
17: //it doesn't advance logical time at all,
18: //while the thread is in the 'sync' context.
19: /*
20: while(true){
21: }
22: */
23:
24: //---------------------------------------------
25: //    mostly-strongly-timed programming
26: //---------------------------------------------
27: }
28: //an array with 16 elements.
29: var wsarray = new Array(16);
30:
31: //using an 'async' statement to switch to the
32: //asynchronous/preemptive context, so that the
33: //task can be preempted by the scheduler.
34: async {
35:   //the below doesn't suspend real-time DSP,
36:   //as the thread can be preempted this time.
37:   for (var i = 0; i < 16; i+= 1){
38:     //load sample0.wav- sample15.wav
39:     LoadSndFile(i, "sample" .. i .. ".wav" );
40:   }
41:
42:   //then switch back to the 'sync' context
43:   sync {
44:     //now in the non-prepemtitve context.
45:     //the code is executed with the sample-rate
46:     //accurate timing behavior.
47:     for (var i = 0; i < 10; i+= 1){
48:       //randomly change the sine wave frequency.
49:       p.s.freq = Rand(1, 10) * 2220;
50:       now += 1::second;
51:     }
52:     //now switch to the 'async' context again.
53:     async {
54:       //extract wavesets from the buffers.
55:       //the analysis can take time if the sound
56:       //data is large. Yet, the below task won't
57:       //suspend real-time DSP as the thread
58:       //is now in the async context.
59:       for (var i = 0; i < 10; i+= 1){
60:         wsarray[i] = ExtractWavesets(i);
61:       }
62:     } //the end of the async block (lines 53-62)
63:   } //the end of the sync block (lines 43-63)
64:
65:   //now we are in the async context (lines 34- )
66:   //unlike on line 20-21, the below loop does not
67:   //suspend real-time DSP, since the thread
68:   //is currently in the async context.
69:   while(true){
70:   }
71: }//the end of the async block (lines 34-71)
```

Figure 3. A mostly-strongly-timed programming example in LC.

4. DISCUSSION

4.1 Extending strongly-timed programming with the asynchronous/preemptive behaviour

As discussed in Section 2, in a computer music system that performs real-time DSP in a separate thread, it is difficult to synchronize the timing between compositional algorithms and real-time sound synthesis. The use of the synchronous approach and logical time may compensate for such a loss of the timing precision to a considerable degree. Depending on how the runtime environment (e.g., virtual machine or interpreter) schedules internal tasks, the predictability and repeatability can be also recovered at least at the logical time level. For instance, the scheduling strategy of the ChucK virtual machine is made highly deterministic and predictable [21].

However, as repeatedly emphasized, due to the underlying ideal synchronous hypothesis, a computer music system built upon the synchronous approach can suffer from the temporary suspension of real-time DSP in the presence of a time-consuming task. As both real-time DSP and compositional algorithms are performed within the same thread, if any task blocks audio computation for a long period, the computer music system may miss the deadline for sound output.

There are various kinds of tasks that can be time-consuming in computer music. For instance, it would consume a significant amount of time to analyse large sound data. It may be argued that the temporary suspension of real-time DSP can be avoided by dividing a time-consuming task into a number of sub-tasks, interleaved by the explicit advance of logical time.

Contrary to expectation, this programming pattern is not always realizable. For example, assume that a user wants to load a large sound file from the disk. This task can consume a significant amount of time, as it involves disk access. A user may divide this task into the number of disk accesses to load the sound data a little at a time. Yet, the duration of the I/O block caused by the disk access is unpredictable; there may be other processes accessing the same disk simultaneously, or the disk itself may not be located on the same computer, but on the local area network. In both cases, each sub-task can consume more time than expected.

One of the perspectives suggested by the strongly-timed programming concept is that the issue of timing behaviour can be viewed as a problem with the programming concept applied to the language. This perspective allows further investigation as to wether there can be a programming concept that suits as a solution, temporarily putting the software framework design issues aside, which are more related to implementation.

Based on this perspective, we proposed the mostly-strongly-timed programming concept. As our view of this problem is that the temporary suspension of real-time DSP is rooted in the underlying ideal synchronous hypothesis, our approach is to extend the strongly-timed programming concept by utilizing asynchronous behaviour. By such an extension, mostly-strongly-timed programming intends to avoid temporary suspension of real-time DSP by the explicit switch to asynchronous/preemptive context when performing a time-consuming task.

Previous works already exist that extend synchronous programming languages with asynchronous behaviour. The target application domain and the background motivation of the mostly-strongly-timed programming concept greatly differ from these works. Berry *et al.* extended *Esterel* [5], a synchronous programming language for reactive systems for *communicating reactive processes*, "where a set of individual reactive synchronous processes is linked by asynchronous communication channels" in [6]. Baldamus and Schneider also discussed the extension of *Esterel* and *PURR* [17] by asynchronous concurrency and non-determinism to describe asynchronous systems and to generate more optimized code in [3].

Thus, these previous works target reactive systems. Their motivations are in communicative reactive processes or in the optimization of the generated code, whereas mostly-strongly-timed programming targets interactive systems, with a significant focus on computer music applications. The motivation here is achieving precise timing behaviour while avoiding the suspending of audio computation; both the target application domain and the background motivation of mostly-strongly-timed programming significantly differ.

4.2 The implementation issues

While strongly-timed programs in ChucK are executed as software threads within ChucK's own virtual machine, it is not difficult to translate it to the programs that utilize coroutines in another language. However, such a simple conversion is not possible for a mostly-strongly-timed program, because the underlying scheduler must be allowed to preempt threads in asynchronous context, at an arbitrary time.

A user may consider it is possible to check if the underlying scheduler is requesting a preemption, by inserting *synchronization points* into the translated program[7]. As the request status can be actively checked at the synchronization points and a coroutine can yield when the request is made, it seems possible to mimic the preemption. However, this strategy does not work well for the tasks that involve the I/O block as described. For instance, if a native function call is made to load a large file, this single native function call may consume a significant amount of time for disk access. As the compiler or virtual machine

[7] The strategy of inserting *synchronization points* into the generated code by a compiler can be also often be seen when implementing a garbage collector. "At such a synchronization point, a test of a global variable indicates if a thread switch is required, and some code is executed if this is the case" [18, p.43], for instance, the code to scan the root objects at the beginning of a garbage collection phase.

cannot simply insert a synchronization point inside a native function, as they are able to into a translated code or into a bytecode, the underlying scheduler cannot perform the preemption until the native function call is over.

Moreover, the ***sync*** and ***async*** statements seem irreplaceable with corresponding API functions and may require handling at the virtual machine level, when considering non-local exits, (e.g., execution-time constraints or exception-handling). Figure 4 describes a simple example that involves an execution-time constraint in LC. The *within-timeout* statement can be used to give an execution time constraint in LC. When the execution-time constraint given by the *within* statement is violated, the code immediately jumps to the *timeout* block; otherwise the *timeout* block is simply skipped.

In Figure 4, the code jumps when exactly five seconds have passed from line 05 to line 09. On line 09, the thread already exited *async* context, and *sync* context should be recovered. Hence, there should be no advance of logical time when the code reaches to the *timeout* block; line 09 should be executed right at the timing, when the execution-time constraint is violated (exactly five seconds after line 02 is executed).

```
01: sync {
02:   within(5::second){
03:     async {
04:       //a function call that consumes 10 sec.
05:       funcConsume10sec();
06:     }
07:   }
08:   timeout {
09:     println( "timeout!" );
10:   }
11: }
```

Figure 4. A simple execution-time constraint example in LC.

```
01: var prevCtx1;
02: {
03:   //save the current context to recover it later.
04:   prevCtx1 = GetCurrentContext();
05:   //switch to the sync context.
06:   switchToSyncContext();
07:   {
08:     var prevCtx2;
09:     within(5::second){
10:       //save the current context (sync)
11:       prevCtx2 = GetCurrentContext();
12:       switchToAsyncContext();
13:
14:       //a function call that consumes 10 sec.
15:       funcConsume10sec();
16:
17:       //recover the previous context (sync)
18:       setCurrentContext(prevCtx2);
19:     }
20:     timeout {
21:       //before the below function call,
22:       //the underlying scheduler may preempt!!
23:       //recover the previous context (sync)
24:       setCurrentContext(prevCtx2);
25:       println( "timeout!" );
26:     }
27:   }
28: }
29: //recover the original context
30: setCurrentContext(prevCtx1);
```

Figure 5. An example of performing context switching by the corresponding API function calls.

Assume that a mostly-strongly-timed program is translated to another language, the runtime environment of which is capable of performing preemption in asynchronous context, and assume that the API calls, such as *switchToSyncContext*, *switchToAscynContext* and *setCurrentContext,* can perform explicit switching and be used for recovery of the given context, respectively. Given such an assumption, it could be argued that the Figure 4 example may be translated into a program similar to the Figure 5 example.

However, the code does not work as expected, and these two examples behave differently in a certain situations. In the original Figure 4 example, when the code should jump to line 08, the current context should be recovered, since in synchronous/non-preemptive context, logical time should not be advanced, as described earlier. Yet, in the Figure 5 example, the code is still in the asynchronous context until the *setCurrentContext* API call is made on line 24 to recover the *sync* context. The underlying scheduler may preempt right after the code jumps to the time out block, before line 24 is executed, so that it can avoid allowing the virtual machine to miss the dead line for audio computation. As a result, the preemption would cause the implicit advance of logical time, as the audio computation is performed. Such behaviour clearly differs from the Figure 4 example.

Thus, unlike a purely strongly-timed program, a mostly-strongly-timed program is not well translated into other programs that utilize coroutines. It seems to be desirable to be executed a mostly-strongly-timed program in a runtime environment spefically designed for mostly-strongly-timed programing.

Considering such issues, the current proof-of-concept prototype of LC provides its own bytecode compiler and virtual machine. The virtual machine executes the software threads, which run the user programs, and audio computation is performed within the same native thread inside the virtual machine. Context switching and restoring are managed by the virtual machine, together with other features such as execution-time constraints and exception handling. Some built-in native functions that may cause I/O blocking, such as file access and console output, are implemented so that they can be performed in separate threads when called in asynchronous context so that they do not block audio computation.

While there can be various implementations of the runtime environment for mostly-strongly-timed programming, the current prototype of LC can execute mostly-strongly-timed programs as expected from the concept. The prototype proved that the concept is fairly realizable, without damaging the precise timing behaviour of the original strongly-timed programming concept and the capability of real-time DSP.

5. CONCLUSION

In this paper, we proposed a novel programming concept, *mostly-strongly-timed programming*, which extends

strongly-timed programming by the explicit switching between synchronous context and asynchronous context and its integration into the language design of LC, a new computer music programming language we prototyped.

Since the underlying scheduler can perform the preemption of the threads in asynchronous context, a mostly-strongly-timed program can avoid temporary suspension of real-time DSP in the presence of a time-consuming tasks as seen in strongly-timed programs, by enclosing the time-consuming tasks within an asynchronous context.

We also described why such an extension in the behaviour makes a mostly-strongly-timed program practically untranslatable to another program that utilizes coroutines unlike in the case of a strongly-timed program, together with the issues to consider in the implementation of the runtime environment.

6. FUTURE WORK

While the current proof-of-concept prototype proved that the mostly-strongly-timed programming concept is realizable, the programming concept itself is still in its infancy and leaves room for improvement. Further discussion on the concept and implementation is desirable.

7. REFERENCES

[1] D. P. Anderson and R. Kuivila, "A system for computer music performance," *ACM Transactions on Computer Systems (TOCS)*, *Vol. 8 (1),* 1990, pp.56-82.

[2] D. P. Anderson and R. Kuivila, "Formula: A programming language for expressive computer music," *Computer, Vol. 24(7),* 1991, pp.12-21.

[3] M. Baldamus and K. Schneider. "Extending Esterel by asynchronous concurrency", *Technical Report*, GI/GMM/ITG Fachtagung zum Entwurf Integrierter Schaltungen, 1993.

[4] J. Banks and J. S. Carson. *Discrete-event system simulation.* Pearson Education India, 1984.

[5] G. Berry *et al.*, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming, Vol 19 (2),* 1992, pp.87-152.

[6] G. Berry *et al.*, "Communicating reactive processes," *In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp.85-98

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

[8] P. Burk. "Jsyn – a real-time synthesis api for Java," In *Proceedings of the 1998 International Computer Music Conference*, 1988.

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

[10] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag. 2010.

[11] V. Lazzarini *et al.*, "A toolkit for music and audio activities on the xo computer", In *Proceedings of the 2008 International Computer Music Conference*, 2008

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

[13] H. Nishino *et al.*, "LC: A New Computer Music Programming Language with Three Core Features," *submitted to Proc. ICMC-SMC,* 2014

[14] H. Nishino. "Mostly-strongly-timed programming," In *Proc. ACM SPLASH*, 2012.

[15] M. Pukette, *Pd Documentation*, on-line at http://www.crca.ucsd.edu/~msp/Pd_documentation, 2005. Accessed on Apr 12th, 2014

[16] C. Roads. *Microsound.* The MIT Press. 2004

[17] D. Schmid *et al.,* "Formale Verifikation eingebetteter Systeme," *Informationstechnik und Technische Informatik*, *Vol.2*, 1999, pp.12-16

[18] F. Siebert, *Hard Real-time Garbage Collection In Modern Object-Oriented Programming Languages.* BoD-Books on Demand, 2002

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

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

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

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