Synchronous Programming In Audio Processing: A Use Case Study

Karim Barkati[1] Pierre Jouvelot[2]

[1]Équipe Ingénierie des Connaissances Musicales
Ircam – Institut de Recherche et de Coordination Acoustique / Musique, France

[2]CRI, Mathématiques et systèmes
Mines ParisTech, France

November 30, 2011

Outline

Overview

Motivation: Comparative Study of MPLs and SPLs

Two rich fields with a synchronous core

- Music programming languages rely on an audio processing core, i.e., synchronous signal computations.
- Synchronous programming languages provide robust ways to specify and verify synchronous computations.
- 50+ years MPLs (late 1950's) VS 30 years SPLs (early 1980's).

Motivation: Comparative Study of MPLs and SPLs

Two rich fields with a synchronous core

- Music programming languages rely on an audio processing core, i.e., synchronous signal computations.
- Synchronous programming languages provide robust ways to specify and verify synchronous computations.
- 50+ years MPLs (late 1950's) VS 30 years SPLs (early 1980's).

Lack of a comparative study

- No comprehensive work focused on comparing MPLs and SPLs yet...
- ... while it may provide mutual benefits.

Motivation: Comparative Study of MPLs and SPLs

Two rich fields with a synchronous core

- Music programming languages rely on an audio processing core, i.e., synchronous signal computations.
- Synchronous programming languages provide robust ways to specify and verify synchronous computations.
- 50+ years MPLs (late 1950's) VS 30 years SPLs (early 1980's).

Lack of a comparative study

- No comprehensive work focused on comparing MPLs and SPLs yet...
- ... while it may provide mutual benefits.

Objectives

- Present an overview of both fields.
- Provide a selection of key languages with a typical use case.
- Highlight insights on how they can be compared.

Overview

Synchronous Programming Languages Overview I

Textual languages: Esterel, Lustre, Signal, ConcurrentML, Larissa, Lucid Synchrone, Quartz, ReactiveML, RMPL, SL, SOL, StreamIt, 8 1/2.

Visual languages and environments: Argos, Statecharts, SyncCharts, Argonaute, Polis, Polychrony, Scade, Simulink/Matlab.

Language extensions: ECL (C), Jester (Java), Reactive-C (C), Realtime Concurrent C (C), RTC++ (C++), Scoop (Eiffel), SugarCubes (Java).

Hardware description languages: Lava, SystemC, Verilog, VHDL.

Models and intermediate formats: Averest, DC+, OC, SDL, ULM, UML Marte.

Music Programming Languages Overview I

Textual languages: (old) Music-N language family (I, II, III, IV, V), Csound, SAOL, Nyquist, ChucK, Impromptu.

Visual programming environments: Max/MSP, Pure Data, jMax, Open Sound World.

Physical modeling systems: Modalys, Chant, Genesis / Cordis-Anima.

Miscellaneous: Kyma (graphical sound design environment), STK (C++ toolkit).

Computer-aided composition: PatchWork, Common Music, Haskore, Elody, OpenMusic and PWGL.

10 Studied Languages

0.	**Matlab**	The MathWorks; J. Little, C. Moler, S. Bangert
1.	**Signal**	IRISA/INRIA; Albert Benveniste, Paul Le Guernic
2.	**Lustre**	CNRS/Verimag; Paul Caspi, Nicolas Halbwachs
3.	**Lucid Synchrone**	Verimag, Paris 11; P. Caspi, G. Hamon, M. Pouzet
4.	**Esterel**	INRIA/CMA; Gérard Berry *et al.*
5.	**Omp Stream**	MINES ParisTech/CRI; Antoniu Pop
6.	**Csound**	MIT; Barry Vercoe
7.	**SuperCollider**	Univ. Texas; James McCartney
8.	**Pure Data**	UCSD; Miller Puckette
9.	**ChucK**	Princeton Univ.; Ge Wang, Perry Cook
10.	**Faust**	GRAME; Yann Orlarey *et al.*

Running Example: The Oscillator Use Case

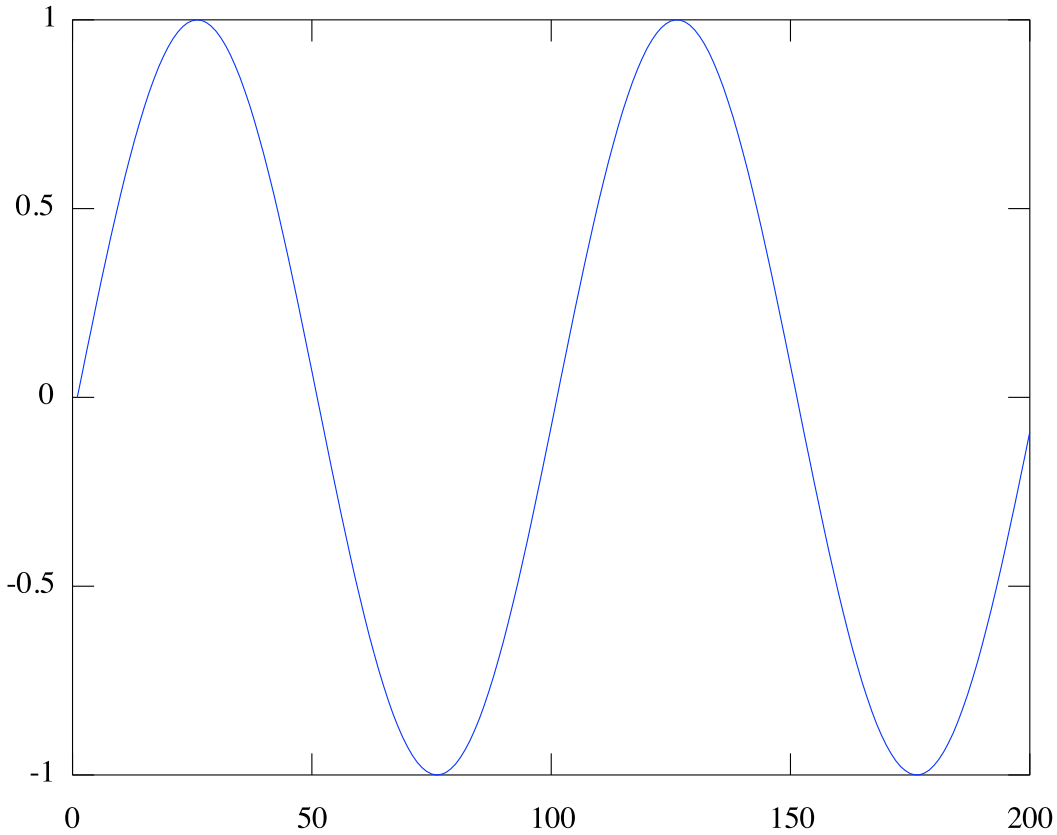


Figure: 200 first output samples

Running Example: The Oscillator Use Case

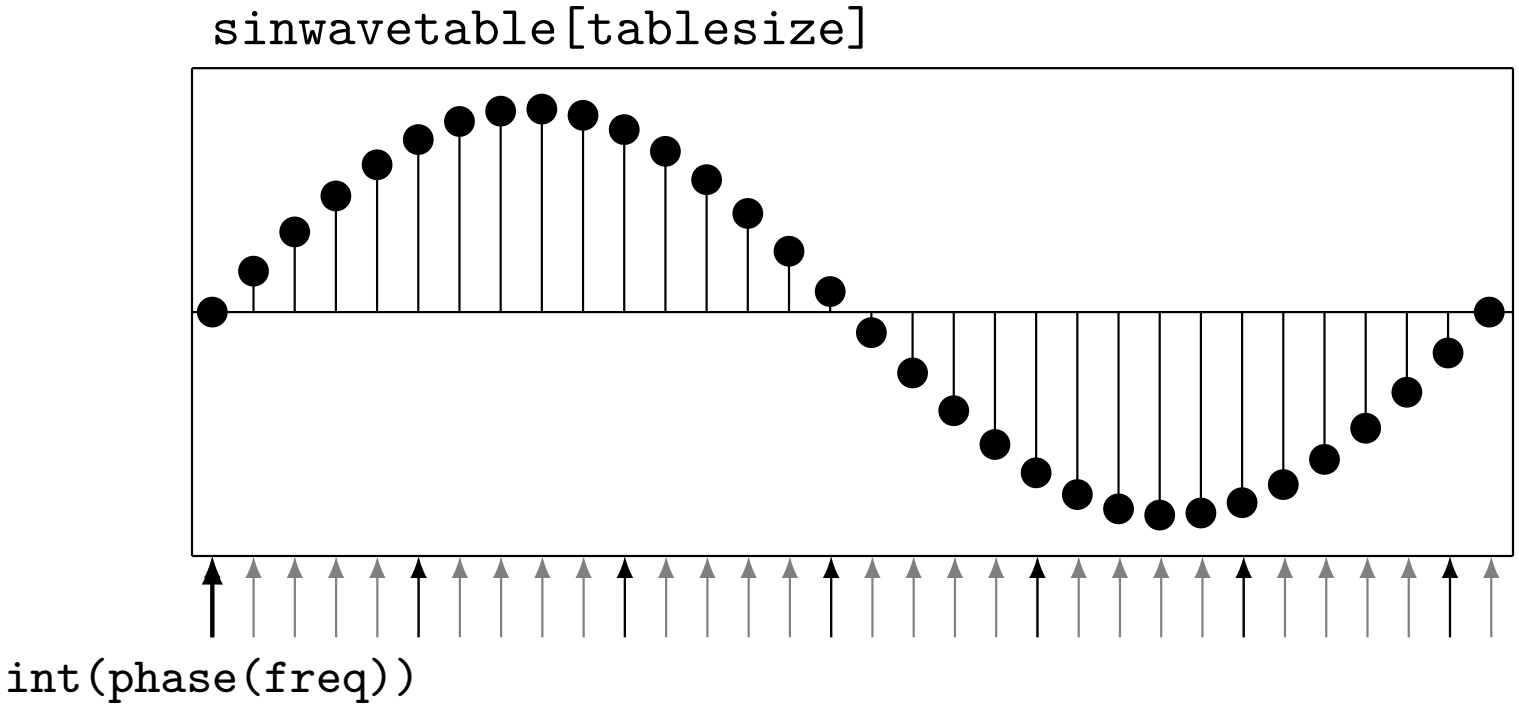


Figure: Truncated lookup table oscillator scheme

Running Example: The Oscillator Use Case

```
const tablesize = 65536                    // number of sound samples
const sinwaveform[tablesize]               // sampled sinusoid (one period)
const samplingfreq = 44100                 // audio sampling rate (Hz)
const freq = 440                           // 'A' diapason frequency (Hz)
const twopi = 6.2831853071795862

func osc(freq)                             // main function
func rdtable(index)                        // dynamic table read access
func phase(freq)                           // phase for each tick
func decimal(float x)                      // decimal part of x in [0;1]
```

Table: General constants and functions for the oscillator implementation

Running Example: The Oscillator Use Case

```
1  function [waveform] = osc(freq)
2      tablesize = bitshift(1, 16);
3      samplingfreq = 44100;
4      outputsize = 200;
5      twopi = 2 * pi;
6
7      ratio(1) = 0;
8      waveform(1) = 0;
9
10     for i = 1 : tablesize
11       sinwaveform(i) = sin( (i * twopi) / tablesize);
12     end
13
14     for i = 2 : outputsize
15       ratio(i) = decimal( (freq / samplingfreq) + ratio(i-1) );
16       phase = tablesize * ratio(i);
17       waveform(i) = sinwaveform(uint16(phase));
18     end
19 end
20
21 function [y] = decimal(x)
22 y = x - floor(x);
23 end
```

Overview

Signal: osc.sig Implementation I

```
1  process osc =
2       ( ? event inputClock;
3         ! dreal output;
4       )
5     (| output ^= inputClock
6      | output := rdtable(integer(phase(freq)))
7      |)
8     where
9     constant dreal freq = 440.0;
10    constant integer samplingfreq = 44100;
11    constant integer tablesize = 2**16;
12    constant dreal twopi = 6.2831853071795862 3;
13    process rdtable =
14         ( ? integer tableindex;
15           ! dreal sample;
16         )
17       (| sample := sinwaveform[tableindex]
18        |)
19       where
20       constant [tablesize]dreal sinwaveform =
21         [{i to (tablesize-1)}:sin((dreal(i)*twopi)/dreal(tablesize))];
```

Signal: osc.sig Implementation II

```
22      end
23    %rdtable%;
24    function phase =
25         ( ? dreal freq;
26           ! dreal phi;
27         )
28       (| index := decimal((freq/dreal(samplingfreq))+index$)
29        | phi := dreal(tablesize)*index
30        |)
31       where
32       dreal index init 0.0;
33       end
34    %phase%;
35    function decimal =
36         ( ? dreal decimalIn;
37           ! dreal decimalOut;
38         )
39       (| decimalOut := decimalIn-floor(decimalIn) |)
40    %decimal%;
41    end
42  %osc%;
```

Lustre: osc.lus Implementation I

```
1  -- WARNING : Does *not* work, because of *dynamic* array accesses!
2
3  include "math.lus"
4
5  const samplingfreq = 44100;
6  const tablesize = 65536;
7  const timeTab = time(tablesize, 0);
8  const sinwaveform = sintable(timeTab);
9  const twopi = 6.28318530717958623;
10
11 node time( const n: int; start: int ) returns ( t: int^n );
12 let
13   t[0] = start;
14   t[1..n-1] = t[0..n-2] + 1^(n-1);
15 tel
16
17 node sintable ( x: int ) returns ( y: real );
18 let
19   y = sin(((real x)*twopi) / (real tablesize));
20 tel
21
```

Lustre: osc.lus Implementation II

```
22 node decimal ( X : real ) returns ( Y : real );
23 let
24   Y = X - floor(X);
25 tel
26
27 node phase ( freq : real ) returns ( Y : real );
28 var index : real;
29 let
30   index = 0.0 -> decimal((freq/(real samplingfreq)) + pre(index));
31   Y = (real tablesize) * index;
32 tel
33
34 node rdtable ( tableindex : int ) returns ( Y : real );
35 let
36   Y = sinwaveform[tableindex]; -- Dynamic array access.
37 tel
38
39 node osc ( freq : real ) returns ( Y : real );
40 let
41   Y = rdtable(int phase(freq));
42 tel
```

Lucid Synchrone: osc.ml Implementation I

```
1  let static tablesize = 65536
2  let static samplingfreq = 44100
3  let static twopi = 6.28318530717958623
4  let ftablesize = float_of_int tablesize
5
6  let static sinwaveform = Array.make tablesize 0.0
7  let static gen_sin () =
8    let rec feed i =
9      match i with
10       | 0 -> ()
11       | i ->
12         (Array.set sinwaveform (i-1)
13                    (sin((float_of_int (i-1)) *. twopi /. ftablesize));
14          feed (i-1))
15     end
16   in feed tablesize
17 let static sidefeeding = gen_sin ()
18
19 let decimal x = x -. floor(x)
20
21 let node phase freq =
```

Lucid Synchrone: osc.ml Implementation II

```
22   let rec index = 0.0 ->
23         decimal((freq /. (float_of_int samplingfreq)) +. pre(index)) in
24   int_of_float (ftablesize *. index)
25
26 let rdtable tableindex = Array.get sinwaveform tableindex
27
28 let node osc freq = rdtable(phase(freq))
```

Esterel: `osc.strl` Implementation

```
1  module Osc:
2
3     function floor_int (double) : integer;
4     constant tableSize_cte = 65536 : integer;
5
6     input I : double;
7     output O : double;
8     output current_index : integer;
9
10       signal index : integer, phase : double, sample : double in
11          every I do
12             run Phase [signal I / freq, phase / phi];
13             ||
14             loop
15                emit index(floor_int(?phase));
16                emit current_index(?index);
17                run RdTable [signal index / tableindex];
18                emit O(?sample);
19             each tick
20          end every
21       end signal
22
23 end module
```

Esterel: rdtable.strl and execmachine.c

```
module RdTable:

   function sinwaveform (integer) : double;

   input tableindex : integer;
   output sample : double;

   emit sample(sinwaveform(?tableindex));

end module
```

```
#include "main.h"

int main () {
  int i = 0;
  Osc_reset ();
  init_sinwaveform ();
  for (; i < 65535; i++) {
  Osc_I_I(440.0);
    Osc();
  }
}
```

Esterel: main.h Implementation I

```
1  #ifndef EXECMACHINE_H
2  #define EXECMACHINE_H
3  #include <stdio.h>
4  #include <math.h>
5
6  double sinwaveform_TAB[65536];
7  void init_sinwaveform () {
8    int i = 0;
9    for (; i < 65536; i++) {
10     sinwaveform_TAB[i] = sin((i*2*M_PI)/tableSize_cte_db);
11   }
12 }
13
14 double sinwaveform (int i) {
15   if (i > 65536) {
16     fprintf(stdout, "Bad table index: %d\n", i);
17     return 0.0;
18   }
19   return sinwaveform_TAB[i];
20 }
21
```

Esterel: `main.h` Implementation II

```
22 double floor_db (double d) {
23   return (double) floor(d); }
24
25 int floor_int (double d) {
26   return floor(d); }
27
28 #endif
```

Esterel: phase.strl Implementation

```
1  module Phase:
2
3     constant samplingfreq_cte = 44100.0 : double;
4     constant tableSize_cte_db = 65536.0 : double;
5
6     input freq : double;
7     output phi : double;
8
9     signal step : double in
10       emit step(?freq/samplingfreq_cte);
11       var index := 0.0 : double, preindex := 0.0 : double in
12          signal decpart := 0.0 : double in
13             every immediate tick do
14                run Decimal [signal step / I, decpart / O];
15                index := ?decpart + preindex;
16                preindex := index;
17                emit phi(tableSize_cte_db*index)
18             end every
19          end signal
20       end var
21    end signal
22
23 end module
```

OpenMP Stream Extension: osc.c Implementation I

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  #define freq 440
6  #define outputsize 200
7  #define twopi 6.28318530717958623
8
9  static inline float decimal (float x) { return x - floor (x); }
10
11
12 int main (int argc, char **argv) {
13   int i;
14   int tablesize = 1 << 16;
15   int samplingfreq = 44100;
16   float *sinwaveform = (float *) malloc (tablesize * sizeof (float));
17
18   for (i = 0; i < tablesize; ++i)
19   sinwaveform[i] = sin (((float) i) * twopi / ((float) tablesize));
20
21
```

OpenMP Stream Extension: osc.c Implementation II

```
22  #pragma omp parallel num_threads (2) default (none)
23     shared (tablesize, sinwaveform, samplingfreq) {
24    #pragma omp single {
25      float f_sf_ratio, index, phase;
26      float dec_add = 0.0; int i = 0;
27      while (i++ < outputsize) {
28        #pragma omp task shared (samplingfreq) output (f_sf_ratio)
29        num_threads (2) {
30          f_sf_ratio = ((float) freq) / ((float) samplingfreq);
31        }
32        #pragma omp task input (f_sf_ratio) output (index)
33        shared (dec_add) {
34          dec_add = decimal (dec_add + f_sf_ratio);
35          index = dec_add;
36        }
37        #pragma omp task shared (tablesize) input (index) output (phase){
38          phase = index * tablesize;
39        }
40        #pragma omp task shared (sinwaveform) input (phase)
41        shared (stdout) {
42          fprintf (stdout, "%f \t %f\n", sinwaveform[(int)phase], phase);
43        }
```

OpenMP Stream Extension: osc.c Implementation III

```
44        }}}
45    return 0;
46  }
```

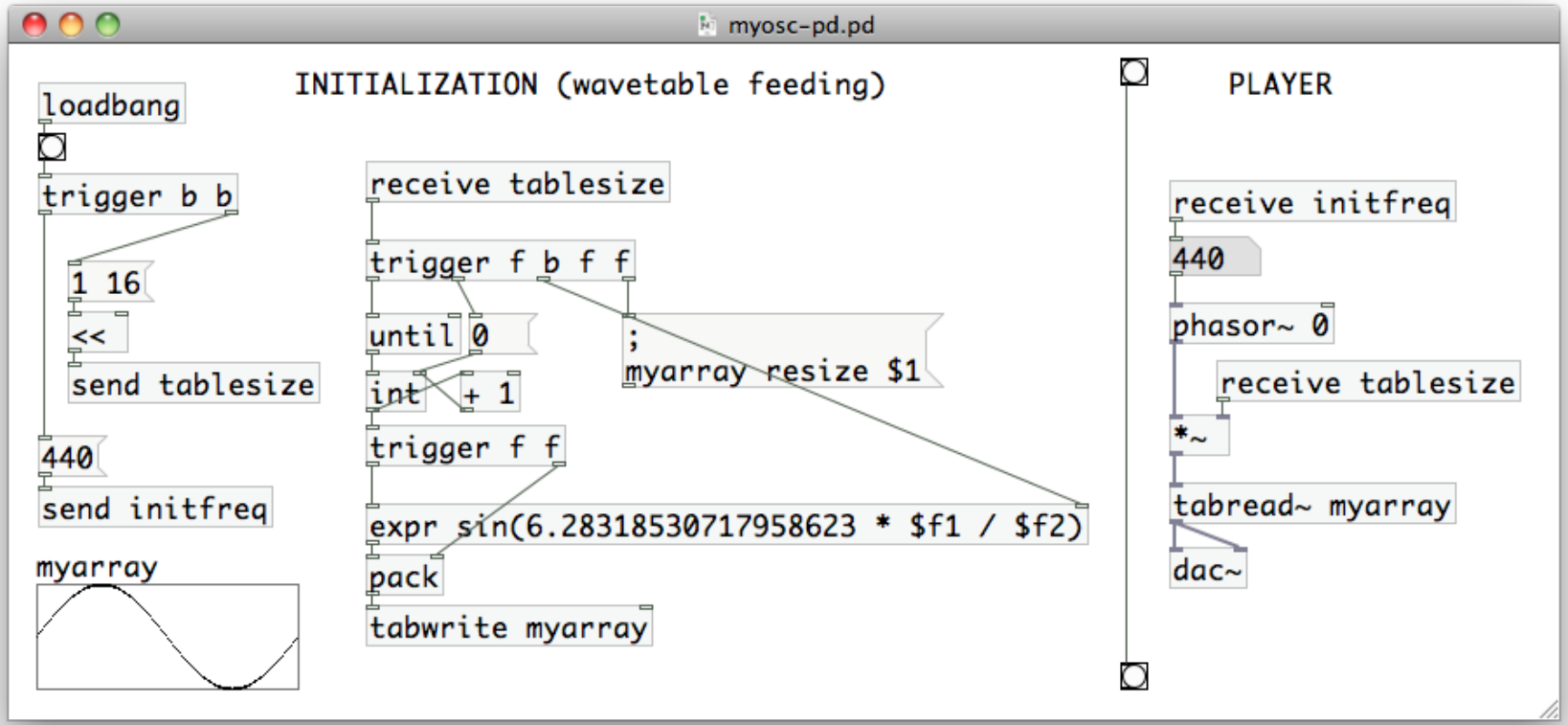
Csound: osc.csd Implementation

```
1  <CsoundSynthesizer>
2  <CsInstruments> ; corresponds to the old '.orc' file
3  sr              =       44100
4  kr              =       441
5  ksmps           =       100
6  nchnls          =       1
7
8                  instr 1
9  aosc            oscil   p4, p5, 1
10                 out     aosc
11                 endin
12 </CsInstruments>
13
14 <CsScore> ; corresponds to the old '.sco' file
15 ; use GEN10 to compute a sine wave
16 f1      0       4096    10      1
17 ;ins    strt    dur     amp     freq
18 i1      0       2       20000   440 ; one note
19 e
20 </CsScore>
21 </CsoundSynthesizer>
```

SuperCollider: osc.scd Implementation

```
1 (
2         var tablesize = 1 << 16;
3         b = Buffer.alloc(s, tablesize, 1); // allocate a Buffer
4         b.sine1(1.0, true, false, true);   // fill the Buffer
5         {OscN.ar(b, 440, 0, 1)}.play       // N: Non-interpolating
6 )
7 b.free;
```

Pure Data: osc.pd Implementation



ChucK: osc.ck Implementation

```
1 Phasor drive => Gen10 g10 => dac; // gen10 sinusoidal lookup table
2
3 [1.] => g10.coefs; // load up the partials amplitude coeffs
4 440 => drive.freq; // set frequency for reading through table
5
6 while (true) // infinite time loop
7 {
8     500::ms => now; // advance time
9 }
```

Faust: osc.dsp Implementation

```
1  import("math.lib"); // for SR
2
3  tablesize      = 1 << 16;
4  samplingfreq   = SR;
5  twopi          = 6.28318530717958623;
6
7  time           = (+(1) ~ _) - 1; // 0,1,2,3,...
8  sinwaveform    = float(time)*(twopi)/float(tablesize) : sin;
9
10 decimal(x)     = x - floor(x);
11 phase(freq)    =
12     freq/float(samplingfreq) : (+ : decimal) ~ _ : *(float(tablesize));
13 osc(freq)      = rdtable(tablesize, sinwaveform, int(phase(freq)) );
14
15 process = osc(440);
```

Overview

Discussion

Concisiveness for audio programs

- MPLs audio oscillator programs much shorter than SPLs ones
- Hopefully reduce software defects (intrinsic value of DSLs)

Discussion

Concisiveness for audio programs

- MPLs audio oscillator programs much shorter than SPLs ones
- Hopefully reduce software defects (intrinsic value of DSLs)

Time

- MPLs: mostly a **hardware** notion, regarding the sampling rate
- SPLs: mostly a **logical** notion, to schedule computations

Discussion

Concisiveness for audio programs

- MPLs audio oscillator programs much shorter than SPLs ones
- Hopefully reduce software defects (intrinsic value of DSLs)

Time

- MPLs: mostly a **hardware** notion, regarding the sampling rate
- SPLs: mostly a **logical** notion, to schedule computations

Signals

- MPLs: simple mappings from regularly-spaced time ticks to values
- SPLs: abstract, complex clocks where time events might be absent

Discussion

Concisiveness for audio programs

- MPLs audio oscillator programs much shorter than SPLs ones
- Hopefully reduce software defects (intrinsic value of DSLs)

Time

- MPLs: mostly a **hardware** notion, regarding the sampling rate
- SPLs: mostly a **logical** notion, to schedule computations

Signals

- MPLs: simple mappings from regularly-spaced time ticks to values
- SPLs: abstract, complex clocks where time events might be absent

Timing layers and non-determinism

- MPLs: low-level synchronous layer + high-level interactive layers
- SPLs: synchronous layer (*sample-* or *event-driven*); GALS

Overview

Conclusions

Music and Synchronous Programming Languages

A practical, use case-oriented survey of key general-purpose and music-specific synchronous programming languages, implementing a simple yet significative audio processing algorithm (frequency-parameterized oscillator generator).

Conclusions

Music and Synchronous Programming Languages

A practical, use case-oriented survey of key general-purpose and music-specific synchronous programming languages, implementing a simple yet significative audio processing algorithm (frequency-parameterized oscillator generator).

Focus: Languages Design

- Paradigms and concepts
- Timing approach
- Program size
- Audio processing fit
- Bibliography (over 100 references on MPLs, SPLs and DSLs)

Conclusions

Music and Synchronous Programming Languages

A practical, use case-oriented survey of key general-purpose and music-specific synchronous programming languages, implementing a simple yet significative audio processing algorithm (frequency-parameterized oscillator generator).

Focus: Languages Design

- Paradigms and concepts
- Timing approach
- Program size
- Audio processing fit
- Bibliography (over 100 references on MPLs, SPLs and DSLs)

Future Work

- Performance
- Event management