Synchronous Programming in Audio Processing: A Lookup Table Oscillator Case Study

KARIM BARKATI
IRCAM[1], France
and
PIERRE JOUVELOT
CRI, Mathématiques et systèmes, MINES ParisTech, France

The adequacy of a programming language to a given software project or application domain is often considered a key factor of success in software development and engineering, even though little theoretical or practical information is readily available to help make an informed decision. In this paper, we address a particular version of this issue by comparing the adequacy of general-purpose synchronous programming languages to more domain-specific languages (DSL) in the field of computer music. More precisely, we implemented and tested the same lookup table oscillator example program, one of the most classical algorithms for sound synthesis, using a selection of significant synchronous programming languages, half of which designed as specific music languages – Csound, Pure Data, SuperCollider, ChucK, Faust – and the other half being general synchronous formalisms – Signal, Lustre, Esterel, Lucid Synchrone and C with the OpenMP Stream Extension (Matlab/Octave is used for the initial specification). The advantages of both approaches are discussed, providing practical insights to both software developers and language designers regarding the choice of programming language styles when tackling audio applications.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Real-time and Embedded Systems*; *Signal Processing Systems*; D.1.m [**Software**]: Programming Techniques—*Miscellaneous*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*; D.3.2 [**Software**]: Language Classifications—*Concurrent, Distributed, and Parallel Languages*; *Data-flow Languages*; *Specialized Application Languages*; *Very High-Level Languages*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; E.1 [**Data**]: Data Structures—*Arrays*; J.5 [**Computer Applications**]: Arts and Humanities—*Performing Arts*; J.7 [**Computer Applications**]: Computers in Other Systems—*Real Time*; K.2 [**Computing Milieux**]: History of Computing

General Terms: Design, Languages

Additional Key Words and Phrases: Synchronous programming languages, Music programming languages, Computer music, Signal processing, Timing

1. INTRODUCTION

The understanding of the existence of a close relationship between music and mathematics has been mentioned since the ancient Greeks. This is therefore not surprising that programming language designers have considered the musical domain as a venue of choice for their investigations from the early days of the field of computing [Van Roy 2009]: programming is indeed a constructive approach to mathematical reasoning. There are of course multiple ways to use programming languages in mu-

[1]This work was done as part of a postdoctoral stay at MINES ParisTech.

sic applications, from low-level audio processing to more abstract music notation manipulation processes to the higher sphere of music composition frameworks.

Most digital audio applications are based on the algorithmic real-time processing of streams of sound samples; these signals are subject to strong timing requirements since latencies and delays in music signals are easily detectable even by untrained human ears. Such stringent constraints call for languages that are able to manage somewhat significant data throughputs while enforcing strong timing constraints. A family of programming languages has been developed to specifically deal with such timing issues: synchronous languages [Benveniste and Berry 1991b; Benveniste et al. 2003]. This particular programming paradigm is based on the key idea of synchronizing concurrent computing processes on clocks. The relationship of such an approach with the temporal structure of music and real-time audio computing is obviously relevant and justifies the current interest of the music community for such a framework.

The purpose of this paper is to provide a comparative survey of the current portfolio of major synchronous programming languages that can be used in the specific field of audio processing. We believe such an analysis is particularly pertinent today, since this domain has seen a recent significant growth both in the industrial and research worlds, with the widespread use of gadgets such as MP3 players or the introduction of new programming paradigms such as Faust [Orlarey et al. 2009] or ChucK [Wang et al. 2003]. This survey adopts a two-pronged approach: we look both at key music programming languages, i.e., audio-specific languages designed in the computer music community, and general signal-processing programming languages to see how they relate to each other while adopting, with some variations, the synchronous framework. Our approach is a pragmatic one: we use a running example throughout this article, namely the implementation of a lookup table oscillator. For each implementation, we try to stick to the programming style of each language in order to highlight its idiosyncratic aspects.

We believe our work can be of help both to anyone who needs to decide which language is best adapted to a given digital audio project and to language designers. Indeed, the adequacy of a programming language to a project or application domain is often considered a key factor of success in software development, even though little theoretical or practical information is readily available to help make an informed decision. Our intent here is to provide practical insights to both software developers and project managers regarding the choice of a programming language style when tackling audio applications. Language designers might also benefit from our work, in particular for DSL[2] applications, since cross-fertilization between languages via the borrowing of existing language features is a common way to improve language designs.

The structure of the paper is the following. Section 2 provides a brief general overview of the most prominent music and synchronous programming languages. Section 3 contains a description of our running example, which we consider a typical use case for most current audio applications; we provide both an informal and a formal, in Matlab/Octave, specification of our target program, `osc`, a standard wave synthesis algorithm. The two following sections, namely Section 4 for music-

[2]Domain-Specific Language.

specific languages and Section 5 for synchronous languages, present our attempts at implementing `osc` in a few selected languages. For each one of these languages, we use the same presentation format: (1) a brief overview introduces the language – using the own words of its author(s) –, (2) the `osc` example we coded in this particular formalism and (3) a set of notes, when we felt that some explanations were needed for what we assumed would be the most difficult programming details to understand by the reader. Section 6 provides some high-level comments about the plus and minuses of the various approaches. We conclude in Section 7.

2. SYNCHRONOUS PROGRAMMING FOR MUSIC

From theoretical time-complexity issues to user interaction management, the concept of time has a structuring effect on the way computer technology impacts the programming world. In languages specifically dedicated to music programing, music events are expressed along strict timing constraints. Synchronous general-purpose languages also consider (logical) time, along which control and computation are scheduled, as a key design ingredient; they adhere to the "synchronous hypothesis", which emphasizes time constraints and determinism. Even though developed within a totally different research community, music-specific languages also follow this synchronous hypothesis. We survey below these two approaches, and end this section with the list of key representative languages we use in the remaining of this paper.

2.1 Computer Music Languages

Computer music, a vibrant and dynamic research field, targets one of the oldest application domains of computers, starting in the late 1950s on mainframes, with two main branches [Loy and Abbott 1985]: computer-aided composition (CAC) and digital audio synthesis. For this study, we are interested in the latter.

The CAC branch deals mostly with the *note* paradigm, using a symbolic approach, and usually aims at producing scores. It started in 1956 with the MUSICOMP[3] language of Lejaren Hiller and Robert Baker (both chemists at that time) at the University of Illinois using an Illiac I [Baker and Hiller 1963; Hiller and Baker 1964]. Main CAC languages include PatchWork [Laurson and Duthen 1989], Common Music [Taube 1991], Haskore [Hudak et al. 1996], Elody [Orlarey et al. 1997], OpenMusic [Assayag et al. 1999] and PWGL [Laurson et al. 2009].

The digital audio synthesis branch is mostly focused on the *sound* paradigm, using signal processing and physical modeling approaches; the overall goal here is to synthesize files or streams of audio samples, while moving in the mid-1980s to a more "real-time" paradigm better fitted to live performances. This branch started around 1957 with the MUSIC I language of Max Mathews [Mathews et al. 1969], then an engineer at the Bell Telephone Laboratories (Murray Hill, New Jersey), on an IBM 704; at the time, hours of computation were necessary to get a few seconds of sound. The concept of *unit generator*, implemented in Mathews' Music-N languages, will prove itself a pervasive concept in audio signal processing, both in computer music languages and hardware synthesizers.

[3]MUsic Simulator-Interpreter for COMpositional Procedures.

We classify below several significant audio synthesis languages and systems, hence providing a global picture of the domain – note though that some boundaries may be somehow artificial as some languages belong to several categories:

Textual languages. Music-N family languages (I, II, III, IV, V) [Mathews et al. 1969], Csound [Vercoe 1992; Boulanger et al. 2000], SAOL [Scheirer and Vercoe 1999], FAUST [Orlarey et al. 2009], Nyquist [Dannenberg 1997], SuperCollider [McCartney 1996], ChucK [Wang et al. 2003], Impromptu [Sorensen 2005];

Visual programming environments. Max/MSP [Puckette 1991; Zicarelli 1998], Pure Data [Puckette 1996], jMax [Déchelle et al. 1999], Open Sound World [Chaudhary et al. 2000];

Physical modeling systems. Modalys [Eckel et al. 1995], Chant [Rodet et al. 1984], Genesis/Cordis-Anima [Castagné and Cadoz 2002; Cadoz et al. 1993];

Miscellaneous. Kyma [Scaletti 1987] (graphical sound design environment), STK [Cook and Scavone 1999] (C++ toolkit).

2.2 Synchronous Languages

Synchronous programming languages appeared in the early 1980s in France, with Esterel (École des mines de Paris and INRIA, Sophia Antipolis), Lustre (Verimag/CNRS, Grenoble) and Signal (INRIA, Rennes), as an academic research field mixing control theory and computer science [Benveniste and Berry 1991a; Halbwachs 1993; 2005], before becoming of high industrial interest for critical systems [Benveniste et al. 2003] such as those present in avionics, trains and nuclear power plants. The idea of *synchrony* was arising also through Milner's work on communicating systems [Milner 1980], AFCET[4]'s Grafcet [Baker et al. 1987] and Harel's Statecharts formalism [Harel 1987]. Synchronous languages are high-level, engineer-friendly, robust, specification formalisms, rooted in the concepts of *discrete time* and *deterministic concurrency.* Time, seen here as a succession of shared logical instants generated by regular (hence synchronous) support clocks, is usually not explicitly mentioned in the definition of more traditional programming languages. Such a notion is however of paramount importance in the design and implementation of data-flow and control software for *reactive systems* [Harel and Pnueli 1985; Halbwachs 1993] where interactions with external environment processes are subject to time constraints, memory constraints, security constraints and determinism requirements. Due to such stringent objectives, these languages are often equipped with timing and concurrency mathematical models that are structured around automata theory and a typical core hypothesis of instantaneous calculus and communication between logical instants; this paradigm is called the "synchronous hypothesis". The execution of a synchronous program amounts then, at least in principle, to the sequencing of an infinite loop of tightly time-constrained sets of *atomic reactions*, thus preserving both concurrency, inherent to reactive systems, and determinism, highly desirable for critical systems.

To illustrate the richness and diversity of this research field, we provide below a list of forty synchronous languages of interest. This list is neither intended to be exhaustive nor limited to a particular paradigm; it is rather a large-spectrum

[4] *Association française pour la cybernétique économique et technique.*

overview, mixing together very different languages toward a taxonomy of a significant set of synchronous and synchronous-oriented languages. We loosely categorized them using the criteria of syntax (textual, graphic), language definition approach (full-fledged or language extension) and application domain specificities (generic, hardware, models):

Textual languages. Esterel [Berry and Cosserat 1985], Lustre [Caspi et al. 1987], Signal [Gautier et al. 1987; Gamatié 2009], ConcurrentML [Reppy 1999], Larissa [Altisen et al. 2006], Lucid Synchrone [Caspi and Pouzet 1996], Quartz [Schneider 2000], ReactiveML [Mandel and Pouzet 2005], RMPL [Ingham et al. 2001], SL [Boussinot and De Simone 1996], SOL [Bharadwaj 2002], StreamIt [Thies et al. 2002], 8 1/2 [Giavitto 1991].

Visual languages and environments. Argos [Maraninchi 1991], Statecharts [Harel 1987], SyncCharts [André 1996], Argonaute [Maraninchi 1990], Polis [Balarin 1997], Polychrony [Le Guernic et al. 2003], Scade [Dormoy 2008], Simulink/Matlab [Caspi et al. 2003];

Language extensions (original language). ECL (C) [Lavagno and Sentovich 1999], Jester (Java) [Antonotti et al. 2000], Reactive-C (C) [Boussinot 1991], Realtime concurrent C (C) [Gehani and Ramamritham 1991], RTC++ (C++) [Ishikawa et al. 1992], Scoop (Eiffel) [Compton 2000], SugarCubes (Java) [Boussinot and Susini 1998];

Hardware description languages. Lava [Bjesse et al. 1998], SystemC [Initiative 2006], Verilog [Thomas and Moorby 2002], VHDL [IEEE standard 1988];

Models and intermediate formats. Averest [Schneider and Schuele 2005], DC+ [Pnueli et al. 1998], OC [Girault 2005], SC [Girault 2005], DC [Girault 2005], CP [Girault 2005], SDL [Ellsberger et al. 1997], ULM [Boudol 2004], UML Marte [Mallet and André 2009].

2.3 Key Language Representatives

The family of significant synchronous languages dedicated to music is clearly more limited than the one of general purpose, although music seems an obvious application field for the synchronous programming paradigm and its associated languages. Indeed, many concurrent processes (associated to instruments or artists) are intimately linked to the audio sampling frequency that drives the production of sound samples. One of our goals with this research work is to illustrate that a bridge can be made between the synchronous and music programming paradigms we just surveyed.

Even though the brief presentation, above, of existing music and synchronous languages is by no means exhaustive, it makes it obvious that these families of languages offer a very wide variety of possible candidates for our comparison survey. To make our use case analysis project realistic, we need to make a selection to end up with a manageable small subset of these languages. We based our selection criteria on the availability of each language and its associated tools, the import of its design principles on the history of the synchronous and computer music paradigms and a rough assessment of the size of its user base. We thus end this section with the selection of the representative languages that are the basis for our use case study, listed in Table I.

Csound	MIT	B. Vercoe
SuperCollider	Univ. Texas	J. McCartney
Pure Data	UCSD	M. Puckette
ChucK	Princeton Univ.	G. Wang, P. Cook
Faust	GRAME	Y. Orlarey *et al.*
Signal	IRISA/INRIA	A. Benveniste, P. Le Guernic
Lustre	CNRS/Verimag	P. Caspi, N. Halbwachs
Lucid Synchrone	Verimag & Paris 11	P. Caspi, G. Hamon, M. Pouzet
Esterel	MINES/INRIA	G. Berry *et al.*
OpenMP Stream	MINES ParisTech	Antoniu Pop

Table I. Selection of music and general-purpose synchronous languages

3. THE OSCILLATOR USE CASE

Our survey of significant technological tools for the synchronous programming of audio applications is grounded on practical terms. We decided to perform for such an analysis a use case study and picked `osc`, an implementation of a sound oscillator, as test case. The `osc` example is particularly significant for the audio domain since this simple truncated lookup table oscillator algorithm is one of the most classical algorithms of the sound synthesis field and is also involved in other important methods, such as wavetable synthesis, additive synthesis or FM synthesis (frequency modulation).

3.1 Presentation

The purpose of `osc` is to output, in a programming language-specific manner, the successive samples of a sinusoidal waveform; the wave frequency is a parameter of this process, and can be changed at start-up time. Although there are multiple ways to implement such a general specification, we tried to stick to the same scheme in the various programming environments we tested in order to perform an as objective as possible assessment. The basic idea is to always loop over the same single sinusoidal vector for all frequencies, but to decide which sound samples to output according to what the requested frequency `freq` is; for instance, picking every other sample will provide a signal with a frequency twice that of the original if the sinusoidal vector is always looped over at the same rate. In more details (see also Figure 1):

—during the initialization stage, one period of the *sin* function is sampled and `tablesize` samples are stored in the vector `sinwaveform`;

—the main function `osc(freq)` loops over this vector indefinitely while outputting the successive sound samples of appropriate phase, i.e., vector index, `phase(freq)`, for each time tick, according to the provided frequency `freq`;

—each phase `phase(freq)` is the product of `tablesize` and the index $i(n)$ defined as $i(n) = \{i(n-1) + \texttt{freq/samplingfreq}\}$, where $\{x\}$ denotes the decimal part of any number x and `samplingfreq` is the audio output sampling frequency – this kind of recursive equation is a typical tenet of both synchronous languages and digital signal processing (DSP) applications;

—the audio sample corresponding to each particular phase is provided by the `rdtable` function, returning `sinwaveform[int(phase(freq))]`, where `int` computes the integer floor of its argument;

—each sampled data point is finally output, in a more or less platform-specific manner.

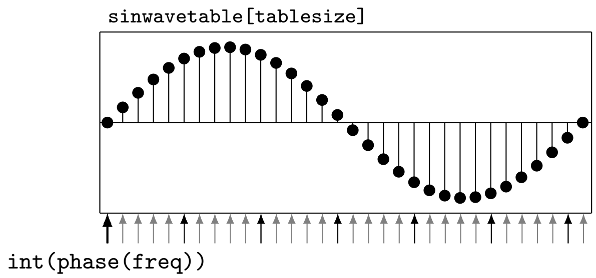

Fig. 1. Truncated lookup table oscillator

3.2 Interface

At the highest level, nine general constants and functions have to be provided to ensure the existence of a working implementation of `osc`; the corresponding signature is listed in Table II. Of course, depending on the particular programming language and its abstraction level, some of these functions will be indeed visible in the `osc` program text; for others, they will only be implicitly present in our implementation.

```
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 = 6.28318530717958623
void osc(freq)                       // main function
float rdtable(index)                 // dynamic table read access
float phase(freq)                    // phase for each tick
float decimal(float x)               // decimal part of x in [0;1]
```

Table II. Oscillator signature: general constants and functions

Environment. In order for `osc` to be implemented in a particular language, the following features need to be available in the programming environment:

—the two mathematical functions `sin` and `floor` (used to compute the decimal part of a number);

—the ability to perform dynamic reads of tables (vectors of samples);

—a looping construct for table initialization and `osc`.

Usage. The `osc` process is launched by calling the main function `osc` with the chosen frequency as argument, e.g., `osc(440)`. Here we use a constant value for readability purposes, but ideally the `freq` argument should be an input signal, i.e, a stream of frequencies, for example [523.25, 587.33, 659.26, ...] – this particular succession of frequencies would in fact yield a sequence of notes, here [C, D, E, ...].

In theory, this algorithm never terminates, as it is a synchronous program responding to input frequency data. In practice, depending on the language at hand, the user will have to interrupt the execution, in many cases by typing `'ctrl-c'`, or a finite vector of `outputsize` samples will be computed off-line. For the validation of our tests, we computed a reference vector of sound samples; it begins with the following rounded values, corresponding to the 440 Hz diapason at the 44,100 Hz sampling rate: [0.0000, 0.0626, 0.1250, 0.1869, 0.2481, 0.3083, 0.3673, 0.4249, 0.4807, 0.5347, 0.5866, 0.6362, 0.6833, 0.7277, ...]. The output of each implementation has been compared to this reference vector, any mismatch being an indication of something wrong with the corresponding implementation.

3.3 Specification

We provide here an implementation of `osc` using indifferently Octave[5] or Matlab[6], a well-known "high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis and numeric computation". This straightforward imperative implementation should be easily understood by most readers, and serves here as a more formal specification of our use case. Here, the standard output is the array `waveform`.

```
function [waveform] = osc(freq)
tablesize = bitshift(1, 16);
samplingfreq = 44100;
outputsize = 200;
twopi = 2 * pi;

indexes(1) = 0;
waveform(1) = 0;

for i = 1 : tablesize
  sinwaveform(i) = sin( (i * twopi) / tablesize);
end

for i = 2 : outputsize
  indexes(i) = decimal( (freq / samplingfreq) + indexes(i-1) );
  phase = tablesize * indexes(i);
  waveform(i) = sinwaveform(uint16(phase));
end
end

function [y] = decimal(x)
y = x - floor(x);
end
```

[5]Octave is an open-source variant of Matlab: http://www.gnu.org/software/octave/.
[6]http://www.mathworks.com.

Notes

—The `bitshift` expression shifts here a `1` sixteen times on the left, yielding 2^{16} as required.

—The keyword `function` introduces the definition of a function. Here the `decimal` function yields in the return value named `y` the decimal part of its argument `x`, and is used to perform a round-robin in the $[0;1[$ interval.

—The intrinsic function `uint16` returns an unsigned 16-bit integer that approximates its floating point argument, here `phase`. This truncation of the floating point phase values provides the integer indexes needed to access the wave table in read mode.

4. MUSIC LANGUAGES

In some loose sense, all music-specific programming languages use, in one way or another, synchronous idioms, since they have to deal with temporal streams of audio samples. We decided to adopt here a somewhat historical order to present key music programming languages:

—Csound is, in a way, the father of modern audio synthesis languages [Vercoe 1992; Boulanger et al. 2000];

—SuperCollider adopts an object-oriented programming approach, inspired by the SmallTalk language [McCartney 1996];

—Pure Data is, like Max/MSP [Puckette 2002], a typical representative of the visual programming paradigm often adopted by the computer music community, thanks to its appeal to the contemporary music composers [Bresson et al. 2009];

—ChucK exemplifies the importance of on-the-fly programming that now can occur even during music performance through "live coding" practices [Wang et al. 2003];

—finally, FAUST promotes the functional paradigm onto a block-diagram algebra, striving to balance expressivity and run-time performance [Orlarey et al. 2002; 2009].

4.1 Csound

Presentation. The following position statement is extracted from the Csound official site, http://www.csounds.com.

> Csound is a sound design, music synthesis and signal processing system, providing facilities for composition and performance over a wide range of platforms. It is not restricted to any style of music, having been used for many years in the creation of classical, pop, techno, ambient, experimental, and (of course) computer music, as well as music for film and television.

Oscillator. The Csound implementation `osc.csd` of the oscillator can be found below. We tested Csound version 5.11 (float samples) Sep 24 2009, with the graphical interface QuteCsound version 0.4.4.

```
<CsoundSynthesizer>
<CsInstruments>
sr              =       44100
kr              =       441
ksmps           =       100
nchnls          =       1

                instr 1
aosc            oscil   p4, p5, 1
                out     aosc
                endin
</CsInstruments>
<CsScore>
; use GEN10 to compute a sine wave
f1      0       65536   10      1
;ins    strt    dur     amp     freq
i1      0       2       20000   440
e
</CsScore>
</CsoundSynthesizer>
```

Notes

—Csound is the oldest musical language of this study: it is a C-based audio DSL following Music11, also developed by Barry Vercoe at MIT in the 1970s, and the MUSIC-N languages initiated by Max Mathews at the Bell Labs in the 1960s. Several aspects present in the Csound language (possibly inherited from previous languages) still persist in later musical languages, so we detail here several aspects of Csound that will be of use for the understanding of most of the sections dedicated here to musical languages.

—In the header, the two assignments "`sr = 44100`" and "`kr = 441`" stand for *sample rate* and *control rate*, which are two fundamental concepts in computer music. The first one specifies the discrete audio sampling rate, set according to the Nyquist frequency, which is twice the 20,000 Hz upper human listening bound needed at sampling time to avoid aliasing. The second one specifies the "control" rate; to save computing resources, it is usually set to a value smaller than the audio rate, since it is mostly used to manage music control information, which do not require the very high temporal resolution requested by sound signals.

—The Csound code of `osc.csd` presented here gathers into one file both the so-called *orchestra* and *score* parts, using XML-like sections, although Csound code had originally been stored in two distinct '`.orc`' and '`.sco`' files. "An *orchestra* is really a computer program that can produce sound, while a *score* is a body of data which that program can react to." [Vercoe 1992].

—The Csound syntax makes intensive use of one-letter prefixes:
  —in an instrument definition, "`a`" and "`k`" specify signal rates (so that "`aosc`" is an audio-rate oscillator signal);
  —in an instrument definition, "`p`" followed by a number specifies a reference to the corresponding *parameter* field in the score part (here `p4` corresponds to `20000` and `p5` to `440` when used in `i1`, see below);

- in the score part, "`f`" followed by a list of numbers declares a *function* table (here, the table `f1` will be computed at `0` second, on `65536` points, using the *unit generator* `10`, with a relative energy of `1` for the fundamental frequency; see next notes);
- in the score part, "`i`" followed by a list declares a Csound note that references an instrument to be played (here `i1` requests the computation of `instr 1`, from time `0`, during `2` seconds, with an amplitude of `20000`, at a frequency of `440` Hz);
- finally, `e` asks for the execution of the score.

—The oscillator is synthesized by `instr1` in the orchestra part, using the `oscil` generator, which is a simple direct synthesis oscillator without interpolation. This generator has three arguments: its amplitude `p4`, its frequency `p5` and its function table number `1`, which refers to `f1` in the score part; this latter function table relies on the tenth *unit generator* called "GEN10", specified as the fourth argument of `f1`.

—As GEN10 is also used in several musical languages in the next sections, we cite here its extensive description from the Canonical Csound Reference Manual [Vercoe et al. 2007]:

> **GEN10** – Generate composite waveforms made up of weighted sums of simple sinusoids.
> **Description**: These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 1 pfield using *GEN10*.
> **Syntax**: `f # time size 10 str1 str2 str3 str4 ...`
> **Initialization**: *size* – number of points in the table. Must be a power of 2 or power-of-2 plus 1. *str1, str2, str3, etc.* – relative strengths of the fixed harmonic partial numbers 1, 2, 3, etc., beginning in p5. Partials not required should be given a strength of zero.
> **Note**: These subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on *GEN10*, namely that the partials have to be harmonic and in phase, do not apply to *GEN09* or *GEN19*. In each case the composite wave, once drawn, is then rescaled to unity if p4 was positive. A negative p4 will cause rescaling to be skipped.

4.2 SuperCollider

Presentation. The following position statement is extracted from the SourceForge http://supercollider.sourceforge.net site, since the SuperCollider official site, http://www.audiosynth.com, seems to be less well maintained.

> SuperCollider is an environment and programming language for real-time audio synthesis and algorithmic composition. It provides an interpreted object-oriented language which functions as a network client to a state of the art, real-time sound synthesis server.
> SuperCollider was written by James McCartney over a period of many years, and is now an open source (GPL) project maintained and devel-

> oped by various people. It is used by musicians, scientists, and artists working with sound.

Oscillator. The SuperCollider implementation `osc.scd` of the oscillator can be found below. We tested SuperCollider version 3.4, rev 10205.

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

Notes

—In SuperCollider [McCartney 1996], which has over 250 unit generators (cf. [Valle et al. 2007]), such an oscillator could have been achieved through at least four different ways:
  (1) the one presented here, using a `Buffer` filled by a `sine1` pattern and played by a non-interpolating `OscN` wavetable oscillator;
  (2) replacing `OscN` by an interpolating `Osc` wavetable oscillator;
  (3) using `BufRd`, `BufWr` and `SinOsc` (since `BufRd` is to be filled by a *unit generator*);
  (4) using directly `SinOsc`.

—Since Version 3, SuperCollider is build upon a client/server architecture (communicating via OSC[7]), with a synthesis application on the server side and a remote language application on the client side. So here the `free` method allows the client to tell the server to free the memory of the buffer previously used.

—The first level of parenthesis surrounding the main block of code is a syntactic trick that is used to ensure that the enclosed lines of code will be launched at the same time by the SuperCollider interpreter; this occurs, in practice, when one simply double-clicks inside one of the parentheses (cf. [Valle et al. 2007]).

4.3 Pure Data

Presentation. The following statement is extracted from the Pd-FlossManual, available at http://en.flossmanuals.net/PureData. The Pure Data official site is http://puredata.info.

> Pure Data (or Pd) is a real-time graphical programming environment for audio, video, and graphical processing. Pure Data is commonly used for live music performance, VeeJaying, sound effects, composition, audio analysis, interfacing with sensors, using cameras, controlling robots or even interacting with websites. Because all of these various media are handled as digital data within the program, many fascinating opportunities for cross-synthesis between them exist. Sound can be used to manipulate video, which could then be streamed over the internet to

[7]Berkeley CNMAT's *Open Sound Control* protocol [Wright 2005].

another computer which might analyze that video and use it to control a motor-driven installation.

Programming with Pure Data is a unique interaction that is much closer to the experience of manipulating things in the physical world. The most basic unit of functionality is a box, and the program is formed by connecting these boxes together into diagrams that both represent the flow of data while actually performing the operations mapped out in the diagram. The program itself is always running, there is no separation between writing the program and running the program, and each action takes effect the moment it is completed.

Oscillator. The Pd implementation `osc.pd` of the oscillator can be found in Figure 2. We tested Pure Data version 0.42.5-extended-20091222.

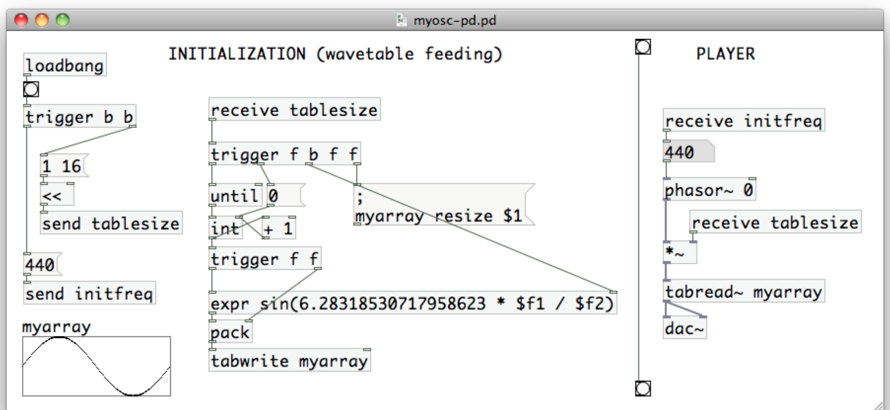

Fig. 2. The osc.pd implementation in Pure Data.

Notes

—Pure Data is a graphical programming environment [Puckette 1996], where a graphical window containing Pure Data code (i.e. boxes and connections) is called a *patch*. The implementation shown on Figure 2 is a screenshot of the patch `osc.pd`.

—This patch is graphically separated into two main parts labelled `PLAYER` and `INITIALIZATION`; the separation is drawn by an idiosyncratic line made up of a connection between two dummy `bang` objects. A `bang` object is used to trigger the most primitive of all event messages, in which only the information that a constant *bang* value has to be sent is encoded.

—Three `trigger` objects are used in order to enforce the sequential order of message emission (*right-to-left* order in Pure Data), which is crucial for such an event-driven language. The syntax of the `trigger` object includes a list of types, where the number of elements determines the number of outputs and where the type names (or one-letter type aliases) determine the type for the corresponding output, allowing type conversion. Here, letters `f` and `b` stand respectively for types `float` and `bang`.

—The `pack` object takes a series of inputs and, when its leftmost input receives a message (according to the right-to-left order), it outputs a concatenated list. Here, the `tabwrite` object receives lists of two elements (*value*, *index*) from `pack` to fill the array `myarray`.

—We use here an `expr` object to simplify the coding of the coefficient calculation by gathering several operations in a one-line fashion, but it could also be done (less clearly) with a traditional Pure Data chain of arithmetic and trigonometric objects.

—Object names ending with a "∼" (tilde) character denote *signal objects*, running at audio rate, and bold connections denote *signal connections*. "[Tilde objects] use continuous audio streams to intercommunicate, and also communicate with other ('control') Pd objects using messages." [Puckette 2007]

—To read arrays, we use a `phasor∼` object which outputs a sawtooth signal between 0. and 1., here multiplied by the table size for table lookup. The dummy argument `0` allows `phasor∼` to receive a non-signal message for the frequency (at control rate).

—The `dac∼` object, for *digital-to-analog converter*, transfers the real-time audio outputs of Pure Data patches to the audio driver of the underlying operating system.

4.4 ChucK

Presentation. The following position statement is a mix of texts from the official site http://chuck.cs.princeton.edu and the ChucK manual.

> ChucK is a new (and developing) audio programming language for real-time synthesis, composition, performance, and now, analysis. ChucK presents a new time-based, concurrent programming model that's highly precise and expressive (we call this strongly-timed), as well as dynamic control rates, and the ability to add and modify code on-the-fly. In addition, ChucK supports MIDI, OSC, HID device, and multi-channel audio. It's fun and easy to learn, and offers composers, researchers, and performers a powerful programming tool for building and experimenting with complex audio synthesis/analysis programs, and real-time interactive control.

Oscillator. The ChucK implementation `osc.ck` of the oscillator can be found below. We tested ChucK version 1.2.1.3 (dracula).

```
Phasor drive => Gen10 g10 => dac; // gen10 sinusoidal lookup table

[1.] => g10.coefs; // load up the partials amplitude coeffs
440 => drive.freq; // set frequency for reading through table

while (true) // infinite time loop
```

```
{
    500::ms => now; // advance time
}
```

Notes

—The ChucK language is specifically designed to allow *on-the-fly* audio programming [Wang et al. 2003].

—The heart of ChucK's syntax is based around the massively overloaded *ChucK operator*, written as '`=>`'. "[This operator] originates from the slang term 'chuck', meaning to throw an entity into or at another entity. The language uses this notion to help express sequential operations and data flow" [Wang et al. 2003]. The ChucK operator's behavior relies on the strong typing system of this imperative language, depending on the type of both its left and right arguments.

—Several elements of ChucK, such as `Gen10` (cf. Csound section), `Phasor` or `dac` (cf. Pure Data section), are inspired by features existing in previous musical languages.

—Here, the `drive` phasor is declared and piped to the `g10` generator, itself connected to the digital-to-audio converter.

—Like in Csound, `Gen10` has to be fed with a list of relative coefficients specifying the harmonics of the spectra; here a single-element array `[1.]` yields a single sinusoid.

—The infinite time loop allows the computing and playing processes declared above it to run; the loop body merely advances time by the arbitrary duration of `500::ms`. Note that the timing model mandates the attachment of a time unit to each duration, such as milliseconds in "`500::ms`".

—Modifying the special variable `now` has the effect of advancing time, suspending the current process until the desired time is reached, and providing the other processes and audio synthesis engine with the computing resources needed to run in parallel.

—The value of `now`, which holds the current time, only changes when it is explicitly modified [Wang and Cook 2007]. "The amount of time advancement *is* the control rate in ChucK." [Wang et al. 2003]

4.5 Faust

Presentation. The following position statement is extracted from the Faust official site, http://faust.grame.fr.

> FAUST is a compiled language for real-time audio signal processing. The name FAUST stands for Functional AUdio STream. Its programming model combines two approaches: functional programming and block diagram composition. You can think of FAUST as a structured block diagram language with a textual syntax.
>
> FAUST is intended for developers who need to develop efficient C/C++ audio plugins for existing systems or full standalone audio applications. Thanks to some specific compilation techniques and powerful optimizations, the C++ code generated by the Faust compiler is usually very fast.

It can generally compete with (and sometimes outperform) hand-written C code.

Programming with FAUST is somehow like working with electronic circuits and signals. A FAUST program is a list of definitions that defines a signal processor block-diagram: a piece of code that produces output signals according to its input signals (and maybe some user interface parameters).

Oscillator. The FAUST implementation `osc.dsp` of the oscillator can be found below. We tested FAUST version 0.9.13.

```
import("math.lib");  // for SR and PI

tablesize       = 1 << 16;
samplingfreq    = SR;
twopi           = 2.0 * PI;

time            = (+(1) ~ _) , 1 : -;  // 0,1,2,3,...
sinwaveform     = twopi*float(time)/float(tablesize) : sin;

decimal(x)      = x - floor(x);
phase(freq)     = freq/float(samplingfreq) :
                  (+ : decimal) ~ _ : *(float(tablesize));
osc(freq)       = rdtable(tablesize, sinwaveform, int(phase(freq)) );

process = osc(440);
```

Notes

—The FAUST language combines a block-diagram algebra [Orlarey et al. 2002] with a functional paradigm [Orlarey et al. 2004].

—The keyword `process` is analogous to `main` in C and has to be defined [Orlarey et al. 2004].

—The sample rate constant `SR` is defined in the imported library file `math.lib` as a *foreign constant*, which is linked to the actual sampling rate of the host application through the architecture compilation mechanism of FAUST and determined at initialization time [Smith III 2010]. This typical DSL feature protects against incompatibilities.

—FAUST uses five block-diagram composition operators [Orlarey et al. 2004]: sequential composition `A:B`, parallel composition `A,B`, recursive composition `A~B`, split composition `A<:B` and merge composition `A:>B` (the last two are not used here).

—The `time` processor definition "`time = (+(1) ~ _) , 1 : -;`" uses the three essential block-diagram composition operators plus two more key elements. From right to left, there are:

  —the sequential composition operator "`:`", to connect the two inputs of the "`-`" processor with the output signals of the preceding processors to compute differences;

  —the parallel composition operator "`,`", to combine the two parallel processors that feed the "`-`" processor (note that order matters for the subtraction);

—the recursive composition operator "∼", to specify a one-sample feedback increment that generates a series of natural numbers, starting at 1 (Figure 3 shows the block-diagram schema of the `time` processor, where the small square on the output denotes the implicit sample delaying operation);
—the identity block "`_`", used here in the one-sample recursive loop to directly connect the output to the input of the increment processor, with no other processor than the *identity* one;
—the partial application "`+(1)`", using the curried form of the processor "`+`" to fix one of its arguments with the "`1`" value.

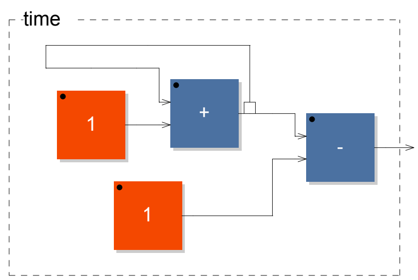

Fig. 3. Block-diagram schema of the `time` processor

—Integer operations are defined modulo the (implementation-dependent) machine integer size, so the increment operation above never overflows, but simply performs a round-robin on its domain.
—Infix notation is allowed in FAUST as syntactic sugar, as in the `decimal` definition where "`x - floor(x)`" is equivalent to "`x, floor(x) : -`".
—The read-only table `rdtable` stores values from a stream at initialization time. The data will then be accessed during execution. Its three parameters are the size of the table, the initialization stream, and the read-index signal [Gaudrain and Orlarey 2003].
—FAUST default output is the audio system within which a FAUST `process` is run.

5. SYNCHRONOUS LANGUAGES

There are various ways to introduce synchronicity in general-purpose programming languages. We focus, in this section, on our oscillator use case example to survey the usual approaches described in the literature:

—The direct approach, taken by Signal, Lustre and Esterel, is to make the notion of synchronous computation at the core of the language design *per se*;
—A more indirect route is to add the notion of streams to an existing language, and among the multiple existing proposals we decided to present two such integrations as illustration: Lucid Synchrone, for the functional paradigm, over OCaml, and OpenMP Stream Extension, for the imperative one, over C and OpenMP.

5.1 Signal

Presentation. The following position statement is extracted from the Signal official site, http://www.irisa.fr/espresso/Polychrony.

> Signal is based on synchronized data-flow (flows + synchronization): a process is a set of equations on elementary flows describing both data and control.
>
> The Signal formal model provides the capability to describe systems with several clocks (polychronous systems) as relational specifications. Relations are useful as partial specifications and as specifications of non-deterministic devices (for instance a non-deterministic bus) or external processes (for instance an unsafe car driver).
>
> Using Signal allows to specify an application, to design an architecture, to refine detailed components down to RTOS[8] or hardware description. The Signal model supports a design methodology which goes from specification to implementation, from abstraction to concretization, from synchrony to asynchrony.

Oscillator. The Signal implementation `OSC.SIG` of the oscillator can be found below. We tested Signal version V4.16.

```
process osc =
     ( ? event inputClock;
       ! dreal output;
     )
   (| output ^= inputClock
    | output := rdtable(integer(phase(freq)))
    |)
   where
   constant dreal freq = 440.0;
   constant integer samplingfreq = 44100;
   constant integer tablesize = 2**16;
   constant dreal twopi = 6.28318530717958623;
   process rdtable =
        ( ? integer tableindex;
          ! dreal sample;
        )
      (| sample := sinwaveform[tableindex]
       |)
      where
      constant [tablesize] dreal sinwaveform =
        [{i to (tablesize -1)}:sin((dreal(i)*twopi)/dreal(tablesize))];
      end;
   function phase =
        ( ? dreal freq;
          ! dreal phi;
        )
      (| index := decimal((freq/dreal(samplingfreq))+index$)
       | phi := dreal(tablesize)*index
       |)
      where
```

[8]Real-Time Operating System.

```
      dreal index init 0.0;
     end;
  function decimal =
       ( ? dreal decimalIn;
         ! dreal decimalOut;
       )
     (| decimalOut := decimalIn-floor(decimalIn) |);
  end;
```

Notes

—Signal processes manipulate signals, i.e., named streams of typed data, either as input "?" or output "!", here of floating point numbers in double precision `dreal`. Local subprocesses are defined similarly.

—Process behavior is defined via sets of functional equations on signals between the `(|` and `|)` enclosing symbols; these equations constrain either the values in a given signal, via the `:=` connector, or clocks, via the `^=` connector, used here to impose that signals `output` and `inputClock` share the same timing information.

—Arrays, such as `sinwaveform` of `tablesize` elements, are defined by intension, at initialisation time, using implicit quantification over indices such as `i` here.

—Data equations are functional, and the `$` postfix is used to reference the previous item in a stream, while `init` is used to specify the initial value.

5.2 Lustre

Presentation. The site http://www-verimag.imag.fr/The-Lustre-Toolbox.html is Lustre official repository and the following position statement is extracted from Wikipedia[9].

> Lustre is a formally defined, declarative, and synchronous dataflow programming language, for programming reactive systems. It began as a research project in the early 1980s. In 1993, it progressed to practical, industrial use, in a commercial product, as the core language of the industrial environment SCADE, developed by Esterel Technologies. It is now used for critical control software in aircraft, helicopters, and nuclear power plants.

Oscillator. The Lustre implementation `osc.lus` of the oscillator can be found below. We tested Lustre version V4.

```
-- WARNING : Does *not* work, because of *dynamic* array accesses!

include "math.lus"

const samplingfreq = 44100;
const tablesize = 65536;
const timeTab = time(tablesize, 0);
const sinwaveform = sintable(timeTab);
const twopi = 6.28318530717958623;

node time( const n: int; start: int ) returns ( t: int^n );
```

[9] http://en.wikipedia.org/wiki/Lustre_\%28programming_language\%29.

```
let
  t[0] = start;
  t[1..n-1] = t[0..n-2] + 1^(n-1);
tel

node sintable ( x: int ) returns ( y: real );
let
  y = sin(((real x)*twopi) / (real tablesize));
tel

node decimal ( X : real ) returns ( Y : real );
let
  Y = X - floor(X);
tel

node phase ( freq : real ) returns ( Y : real );
var index : real;
let
  index = 0.0 -> decimal((freq/(real samplingfreq)) + pre(index));
  Y = (real tablesize) * index;
tel

node rdtable ( tableindex : int ) returns ( Y : real );
let
  Y = sinwaveform[tableindex]; -- Dynamic array access.
tel

node osc ( freq : real ) returns ( Y : real );
let
  Y = rdtable(int phase(freq));
tel
```

Notes

- Lustre sees computation as the processing of data exchanged between nodes. Within each node, functional definitions of data streams are expressed as possibly recursive equations.
- In a stream definition, `pre` is used to denote the previous value in the argument stream, while the arrow `->` operator ("followed-by") is used to distinguish the initial value from the recursive expression when defining a stream by induction.
- Streams are typed, and `^` is used to introduce aggregate vector types (its second argument is the vector size).
- The definition of the array `t` in `time` is by induction over array slices: `t[0]` is 0, while, for all i in `[1..n-1]`, `t[`i`]` is `t[`$i-1$`]`+1, since `1^(n-1)` is an array of `n-1` elements, all initialized to 1.
- Array `timeTab` is initialized via `time`; its elements are integers from 0 to `tablesize`-1, numbering all samples in one sine period.
- Interestingly, the Lustre version we used is in fact *unable* to manage dynamic accesses to arrays, a feature clearly required to implement variable frequencies in `osc`. Newer, non open-source versions of Lustre, such as Lustre V6, do provide dynamic arrays.

5.3 Esterel

Presentation. The following position statement is taken from the original Esterel site, http://www-sop.inria.fr/meije/esterel/esterel-eng.html.

> Esterel is both a programming language, dedicated to programming reactive systems, and a compiler which translates Esterel programs into finite-state machines. It is one of a family of synchronous languages, like SyncCharts, Lustre, Argos or Signal, which are particularly well-suited to programming reactive systems, including real-time systems and control automata.
>
> The Esterel v5 compiler can be used to generate a software or hardware implementation of a reactive program. It can generate C-code to be embedded as a reactive kernel in a larger program that handles the interface and data manipulations. It can also generate hardware in the form of netlists of gates, which can then be embedded in a larger system. Extensive optimization is available. We provide a graphical symbolic debugger for Esterel. We also provide support for explicit or BDD-based verification tools that perform either bisimulation reduction or safety property checking.
>
> Esterel is now experimentally used by several companies and taught in several universities.

Oscillator. The Esterel implementation `Osc.strl` of the oscillator can be found below (see Notes also). We tested version V5 (and GCC 4.2.1 for the C files handling array accesses).

```
module Osc:
  function floor_int (double) : integer;
  constant tableSize_cte = 65536 : integer;
  input I : double;
  output O : double;

  signal index : integer, phase : double, sample : double in
    every I do
      run Phase [signal I / freq, phase / phi];
      ||
      loop
        emit index(floor_int(?phase));
        run RdTable [signal index / tableindex];
        emit O(?sample);
      each tick
    end every
  end signal
end module

module RdTable:
  function sinwaveform (integer) : double;
  input  tableindex : integer;
  output sample : double;
  emit sample(sinwaveform(?tableindex));
end module
```

```
module Decimal:
  function floor_db(double) : double;
  input I : double;
  output O : double;
  emit O(?I-floor_db(?I))
end module

module Phase:
  constant samplingfreq_cte = 44100.0 : double;
  constant tableSize_cte_db = 65536.0 : double;
  input freq : double;
  output phi : double;

  signal step : double in
    emit step(?freq/samplingfreq_cte);
    var index := 0.0 : double, preindex := 0.0 : double in
      signal decpart := 0.0 : double in
        every immediate tick do
          run Decimal [signal step / I, decpart / O];
          index := ?decpart + preindex;
          preindex := index;
          emit phi(tableSize_cte_db*index)
        end every
      end signal
    end var
  end signal
end module
```

Notes

- The Esterel implementation of the oscillator relies on the Esterel `Osc.strl` module file provided above and C helper functions, not presented here, that handle arrays, a data structuring mechanism not provided by the public-domain version of Esterel we used. We wrote the following functions, declared as external C functions in Esterel modules via the `function` keyword:
  - `void init_sinwaveform()`, that initializes a local C array with the double-formatted 65536 samples of a one-period sine function;
  - `double sinwaveform(int i)`, that returns the `i`-th value in this local sampled sine C array;
  - `double floor_db(double d)` and `int floor_int(double b)`, that return the floor value of `d` in either `double` or `int` format;
  - and, finally, `int main()`, that calls `init_sinwaveform()`, provides the initial frequency value of 440 Hz to the Esterel input `I` in the `Osc` module via a call to `Osc_I_I(440.0)` and then calls `Osc()` to initiate Esterel processing. These last two functions are generated by the Esterel compiler.
- Esterel modules specify signals and signal computations that operate on the occurrence of events, including `tick`s issued by a logical clock.
- Esterel supports both internal or external signals. For instance, `Osc` defines internal signals `index`, `phase` (via Module `Phase`) and `sample` (via Module `RdTable`). As for external signals, frequency values appear on `I` while audio samples are output on `O`.

—Esterel signal computations `emit` values of interest on outputs from values read on inputs, using the `?I` notation. All computations are assumed to be performed at each tick in zero time.
—The keyword `every` is syntactic sugar for an event-controlled looping statement. For instance, each time a new frequency value appears on `I` in `Osc`, Module `Phase` is run in a separate thread, with appropriate bindings for input `freq` and output `phi`, to get the proper sequences of array indices in the `phase` signal while an infinite `loop` thread emits on `O` the samples obtained via the `RdTable` module, using the `index` and `phase` signals.
—`Phase` uses the internal `step` signal, which relies itself on the internal signal `decpart`, initialized at `0.0` and that emits the successive indices of the samples in the sine wavetable appropriate for yielding a sine of the given frequency `freq`.
—The `immediate` keyword indicates that the corresponding code is evaluated immediately, even when the tick is already present, as is the case when the program starts.

5.4 Lucid Synchrone

Presentation . The following position statement is taken from the official Lucid Synchrone site, http://www.di.ens.fr/~pouzet/lucid-synchrone.

> Lucid Synchrone is an experimental language for the implementation of reactive systems. It is based on the synchronous model of time as provided by Lustre combined with some features from ML languages. The main characteristics of the language are the following:
> —It is a strongly typed, higher-order functional language managing infinite sequences or streams as primitive values. These streams are used for representing input and output signals of reactive systems and are combined through the use of synchronous data-flow primitives *à la* Lustre.
> —The language is founded on several type systems (e.g., type and clock inference, causality and initialization analysis) which statically guarantee safety properties on the generated code...
> —The language is built above Objective Caml used as the host language. Combinatorial values are imported from Objective Caml and programs are compiled into Objective Caml code. A simple module system is provided for importing values from the host language or from other synchronous modules.
> —It allows to combine data-flow equations with complex state machines (Mealy and Moore machines with various forms of transitions). This allows to describe mixed systems or Mode-automata as originally introduced by Maraninchi & Rémond.
> —Data-types (product types, record types and sum types) can be defined and accessed through pattern matching constructions.

Oscillator. The Lucid Synchrone implementation `osc.ls` of the oscillator can be found below. We tested Version 3.0b.

```
let static tablesize = 65536
let static samplingfreq = 44100
let static twopi = 6.28318530717958623
let ftablesize = float_of_int tablesize

let static sinwaveform = Array.make tablesize 0.0
let static gen_sin () =
  let rec feed i =
    match i with
      | 0 -> ()
      | i ->
        (Array.set sinwaveform (i-1)
           (sin((float_of_int (i-1)) *. twopi /. ftablesize));
         feed (i-1))
    end
  in feed tablesize
let static sidefeeding = gen_sin ()

let decimal x = x -. floor(x)

let node phase freq =
  let rec index = 0.0 ->
    decimal((freq /. (float_of_int samplingfreq)) +. pre(index)) in
  int_of_float (ftablesize *. index)

let rdtable tableindex = Array.get sinwaveform tableindex

let node osc freq = rdtable(phase(freq))
```

Notes

—Lucid Synchrone imports most of its value and typing constructs from OCaml, a mostly-functional object-oriented language in which (possibly recursive) functions are first-class values [Leroy et al. 2010]. It is also inspired by Lustre signal processing concepts.

—Following Lustre, Lucid Synchrone adds to OCaml the notion of a *node*, defined via `let node` declarations. These nodes are used to manipulate streams, which are infinite sequences of values linked to a particular clock.

—In a stream definition i `->` s, i denotes the default, first value of the stream while s is the inductive definition of a stream element. A reference to the previous stream value is allowed using the `pre` operator.

—`Array` is an OCaml module, used here within Lucid Synchrone, that provides standard operations to define (`make`) or manipulate (`set` and `get`) array elements.

—In OCaml, integer operators use the traditional syntax (such as `*` for multiplication), while floating-point constructs use a different notation, via the addition of a dot (`"."`) suffix to the integer notation.

5.5 OpenMP Stream Extension

Presentation. The OpenMP Stream Extension comes as a GCC CVS branch, at http://gcc.gnu.org/viewcvs/branches/omp-stream. Below follows its position statement.

> The stream-computing extension to OpenMP enables the expression of flow dependences between OpenMP tasks. This allows to statically specify the program's dynamic task graph, where tasks are connected through streams that transparently privatize the data. The programming model is conducive to making relevant data-flow explicit and to structuring programs in ways that allow simultaneously exploiting pipeline, data and task parallelism. Stream computations help reduce the severity of the memory wall in two complementary ways: (1) decoupled producer/consumer pipelines naturally hide memory latency; and (2) they favor local, on-chip communications, bypassing global memory.
>
> This extension provides dataflow semantics close to Kahn process networks and guarantees functional determinism, a major asset in the productivity race. In contrast with common streaming frameworks, the communication patterns can be dynamic, while preserving the determinism of arbitrarily merging and splitting data streams. The GCC prototype implementation of the OpenMP extension for stream-computing has been shown to be efficient to exploit mixed pipeline- and data-parallelism, even in dynamic task graphs [Pop and Cohen 2011]. It relies on compiler and runtime optimizations to improve cache locality and relies on a highly efficient lock-free and atomic operation-free synchronization algorithm for streams.

We need to emphasize here that the OpenMP Stream Extension is particularly interesting for our survey since, built on top of an imperative language (C) extended with asynchronous parallel constructs (OpenMP), this language extension is not strictly synchronous. Yet it offers to programmers the ability to perform parallel signal processing operations that loosely adhere to the synchronous hypothesis. Indeed, all stream operations are specified to be deterministic and to not require explicit synchronization actions. This illustrates how synchronous-like operations could be added to other existing traditional languages.

Oscillator. The OpenMP Stream Extension [Pop 2011] implementation `osc.c` of the oscillator can be found below. We tested a prototype directly with its author, Antoniu Pop (from MINES ParisTech's Computer Science Research Center).

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define freq 440
#define outputsize 200
#define twopi 6.28318530717958623

static inline float decimal (float x) { return x - floor (x); }

int main (int argc, char **argv) {
  int i;
  int tablesize = 1 << 16;
  int samplingfreq = 44100;
  float *sinwaveform = (float *) malloc (tablesize * sizeof (float));
```

```
  for (i = 0; i < tablesize; ++i)
  sinwaveform[i] = sin (((float) i) * twopi / ((float) tablesize));

  #pragma omp parallel num_threads (2) default (none)
                       shared (tablesize, sinwaveform, samplingfreq) {
    #pragma omp single {
      float f_sf_ratio, index, phase;
      float dec_add = 0.0; int i = 0;
      while (i++ < outputsize) {
        #pragma omp task shared (samplingfreq) output (f_sf_ratio)
                         num_threads (2) {
          f_sf_ratio = ((float) freq) / ((float) samplingfreq);
        }
        #pragma omp task input (f_sf_ratio) output (index)
                         shared (dec_add) {
          dec_add = decimal (dec_add + f_sf_ratio);
          index = dec_add;
        }
        #pragma omp task shared (tablesize) input (index)
                         output (phase) {
          phase = index * tablesize;
        }
        #pragma omp task shared (sinwaveform) input (phase)
                         shared (stdout) {
          fprintf (stdout, "%f \t %f\n",
                   sinwaveform[(int)phase], phase);
        }}}}
  return 0;
}
```

Notes

—OpenMP Stream Extension is an upward-compatible extension of the OpenMP standard [Dagum and Menon 1998], which extends sequential languages with options for parallel execution. OpenMP has multiple language bindings, and its C variant uses `#pragma omp` C preprocessor-like directives to describe thread-parallel tasking.

—The `parallel` pragma is an OpenMP-specific directive used to open a parallel section in which multiple tasks may be used on up to `num_threads` parallel threads. A separate task is launched when a `task` pragma is encountered, running the statement following it on one of these threads. The `single` pragma enforces its following sequence of code to be run by only one thread (which runs here the `while` loop that starts all required tasks).

—All variables declared within a parallel block are local to each task by default; global variables accessed by a particular parallel construct have to be listed in the `shared` parameter.

—OpenMP Stream Extension extends the `task` pragma, used to specify an OpenMP parallel task, with the `input` and `output` parameters that introduce stream processing into OpenMP. A stream parallel task processes its input signals to yield output data accordingly.

6. DISCUSSION

We look, in this section, at some of the issues raised by our implementations of `osc`, in particular related to the differences between DSL and general-purpose languages, the subtle differences in the notions of time and signals these formalisms introduce, the way they handle aggregate data structures such as arrays, and finally the integration of asynchrony.

6.1 The DSL vs. General-Purpose Languages Debate

In this work, we surveyed 10 languages, 5 specific to computer music applications and 5 general synchronous languages. Each of these languages provides in one way or another answers to the same design questions, such as how to manage time or what are signals supposed to represent for the problems at hand. Yet, the final language design decisions, summarized in Table IV, vary widely, mostly along the line of whether the corresponding language is intended to be used in a somewhat limited application domain, i.e., strives to be a music-specific DSL, or able to tackle a wide range of time-constrained problems, i.e., is a general-purpose synchronous language.

	Music programming languages	Synchronous languages
Code size	much shorter	longer
Time	mostly a hardware notion	mostly a logical notion
Signals	simple tick mappings	abstract and complex clocks
Layers	low-level + interactive high-level	synchronous layer + GALS

Table IV. Design concepts comparison summary (see Section 6.4 for GALS)

Of course, the DSL vs. General-Purpose separation line is not enough to automatically imply which particular programming traits a given language should adopt. To get a feel for the spectrum of notions spanned by this survey, we summarize in Table V the main features of these programming tools, from their core computing paradigm to the way they deal with recursion to their handling of parallelism.

	Paradigm	Delay	Initialization	Parallelism
Csound	orch. + score	`delay`	2^{nd} arg.	orchestra
SuperCollider	object oriented	`DelayN`	3^{rd} arg.	instances
Pure Data	visual	`delread~`	2^{nd} arg.	diagram
ChucK	on-the-fly	`Delay`	`.delay`	orchestra
FAUST	functional	~	fixed to 0	`,` or `par`
Signal	relational	*signal*`$`	`init`	implicit
Lustre	equational	`pre`	`->`	implicit
Esterel	imperative	`pre(?`*signal*`)`	`init`	`||`
Lucid Sync.	functional	`pre`	`->`	implicit
OMP Stream	imperative	window	explicit	implicit

Table V. Salient design points of surveyed languages

One obvious result of our work is that using a single simple yet significant application to illustrate the expressiveness power of various programming languages provides an interesting and practical point of view for software development tool selection. In particular, choosing an audio application as our main running test case yields a specific example of the intrinsic value of Domain Specific Languages as a general and practical approach to the software productivity wall [Mernik et al. 2005; Van Deursen et al. 2000]. Indeed and not surprisingly, we were able to use all these tools to get the work done, except for a technical limitation of the open-source version of Lustre we used. Yet all our programs are much shorter, and thus quite probably correct, using music-specific DSLs than general-purpose synchronous languages. Even though the idea that the number of bugs introduced in a particular program is mainly a function of the number of lines of code and is rather independent of the programming language used may be mostly folklore, we feel that our analysis illustrates in a very concrete manner that conciseness, and hopefully then lack of software defects, clearly lies within the field of DSLs.

6.2 Time and Signals

Although time is, as we mentioned in Section 2, the core concept that structures the definition of all the languages used in this use case study, it is obvious that music-oriented and reactive systems have a somewhat different view of what this notion means. In the traditional synchronous programming world, time is mostly a logical notion, around which computations are scheduled; indeed, multiple clocks can even be defined, e.g., via the `^=` symbol in Signal. For music aficionados, time is a hardware notion deeply linked to the speed at which sound is sampled by input and output converters; the key notion here is the "sampling rate", e.g., via the `SR` predefined identifier in Faust. In some sense, music languages, as DSLs, are more closely linked to the practical matters at hand than the more abstract, hence more general, traditional synchronous programming languages.

Consequently, the notion of what a signal is varies also in the two communities, even though both use the same foundation, i.e., the concept of time. Following a more pragmatic approach, music synchronous languages view signals as mappings from regularly-spaced, sampling rate-sequenced time ticks to values. Traditional languages have to deal with more complex clocks, for instance where time events might even be absent, which leads to more abstract notions of signals. In music applications, all values of a sampled signal are defined (barring computing errors such as a 1/0 division), while general signals may yield undefined values for some time events, and such undefined values are first-class in these languages [Benveniste et al. 2003].

The apparently limited approach of what time and signals must be in music applications is mitigated by the fact that there is a strong tendency in this community to address timing issues as a two-tiered problem: a low-level synchronous layer, that deals with concrete and predictive sampled signals, and a higher-level interactive, and in fact mostly asynchronous, level, that schedules these activities in response to the user. These two strata are often embedded in the same language or environment, using one scheduler at audio rate (typically 44,100 Hz, with high priority and low latency based on buffering techniques) and another one at a much lower "control rate", usually managing MIDI events (medium priority and latency) and GUI objects (low priority and high latency). On the other hand, traditional synchronous languages tend to address only the issues relevant to the first class of problems, using either a *sample-driven* execution scheme as in Lustre or an *event-driven* one as in Signal, and rely on a different programming paradigm to link the synchronous modules together, possibly using a GALS[10] approach [Teehan et al. 2007]. Thus, they usually require more general and flexible notions for time and signals than the ones found in audio languages, since they do not have to follow the fundamental rhythm of the otherwise primary audio sampling rate.

These two approaches regarding this core notion of time in the synchronous layer lead to different approaches to the compilation process. Traditional synchronous languages are more specification-oriented than audio languages; the programmer provides equations defining clock and signal values, relying on the compiler to implement them in efficient sequences of computations. Audio/music frameworks have to deal less with the issue of reifying relationships between logically synchronized computations than with the efficient implementation of explicitly synchronized processes (see for instance Pure Data connected graphs or FAUST functional expressions).

Synchronous languages, because of their more abstract and logical view of time and signals, are formally defined through complex mathematical semantic models [Manna and Pnueli 1995; Schneider 2004]. These formal specifications are moreover of key importance given the domains these languages target, i.e., within strongly reactive and very often mission-critical environments. Music, on the other hand, can, and has to, deal with more "soft" constraints: the notion of truth is more in the ear of the listener/composer than in the strict structure of a mathematical proof. Of course, the human ear is quite a subtle device, and professional listeners have been shown to be quite sensitive to even very small differences in two audio signals. Moreover, even for music applications, a trend is appearing, which calls for more assurance in the fidelity of the audio processing methods, in particular when one wishes to address the issue of long-term and exact preservation of the world musical heritage [Guercio et al. 2007; Bachimont et al. 2003; Barkati et al. 2011].

6.3 Dynamic Array Access

Table lookup is about performance: computer music makes an intensive use of wavetables to avoid the expensive computation of trigonometric functions like sine functions for each sample at given audio rates, typically 44,100 times by second, for audio synthesis (wavetable synthesis, waveshaping, etc.). It is noticeable that most of the music programming languages studied here, except Pure Data and FAUST, borrow the idea of the GEN routines introduced in Csound; they are used as data generators to fill so-called *function tables* [Boulanger et al. 2000]. For instance, our oscillator uses the *GEN10* routine to fill the oscillator sample table at initialization time with a sum of sinusoids (only one here); this table is then read using a wrap-around lookup process. Pure Data handles an explicit array data structure and an explicit phasor to generate the reading indexes, while FAUST provides implicit table initialization and reading operation via the (not quite functional) `rdtable` ternary function.

[10]GALS stands for "Globally Asynchronous, Locally Synchronous".

	Initialization	Dynamic Access
Csound	`f1 0 65536 10 1 ; GEN10`	`oscil p4, p5, 1`
SuperCollider	`b.sine1`	`{OscN.ar(b,440,0,1)}.play`
Pure Data	`tabwrite myarray`	`tabread~ myarray`
ChucK	`[1.]  => g10.coefs`	`Gen10 g10 => dac`
FAUST	`rdtable` 1^{st} and 2^{nd} arg.	`rdtable` 3^{rd} arg.
Signal	`[i to (size-1):sin(...)]`	`sinwaveform[tableindex]`
Lustre v4	*external C code*	*imported function*
Lucid Sync.	OCaml `Array.make` and `.set`	OCaml `Array.get`
Esterel v5	*external C code*	*imported function*
OMP Stream	C code	C code

Table VI. Table support

Our study reveals that general-purpose synchronous languages are often poorly equipped to support tables. Of course, they cannot be expected to provide music-specific GEN-like routines, but, more surprisingly at first sight, most of them simply do not handle array data structures, as is the case with the Esterel and Lucid Synchrone languages in the versions we used. Furthermore, the Lustre version we tested do provide an array syntax, but only as syntactic sugar for variables numbering, not for dynamic array access. Of course, it is generally possible to handle array initialization and dynamic access by importing foreign functions, C functions in most languages, as we did in our Esterel implementation of the oscillator, or OCaml functions in Lucid Synchrone, as we shown. The underlying reason for not handling arrays in the languages themselves is often the difficulty of ensuring that the synchronous time and memory constraints are still enforced, which is crucial for critical synchronous applications.

Commercial and more recent versions of Lustre and Esterel do handle arrays. Nevertheless, our survey suggests that the designers of synchronous languages could look at the GEN-like mechanism inspired by music programming languages as a safe strategy for the introduction of array data structures in these formalisms. Another possible approach to ensure a mathematically-correct integration of arrays and synchronous constraints is to couple in a single analysis the rates of signals with the size of the elements they convey (see for instance [Jouvelot and Orlarey 2011]).

6.4 Event Management

Our survey focused on the audio signal processing part of the computer music domain, since audio DSP shares obvious features with synchronous applications, among which time and signal concepts – even though these are subtly different in both fields, as we have shown. The DSP part corresponds to the *sampled scheme* of evaluation of synchronous languages, where the main loop handles each sample, as opposed to their *control scheme*, where the main loop manages each interaction event. Synchronous languages often rely on a two-tier strategy to handle the integration of asynchrony, for instance via a GALS (*globally asynchronous, locally synchronous*) approach; most computer music languages are, them, inherently hybrid along the synchronous/asynchronous separation line.

Indeed, in addition to the DSP part, most music programming languages also embed event management for musical note entities and interactive remote control of parameters; this is typically done using message passing standards such as MIDI[11] or OSC[12]. At the implementation level, musical languages handle this event part in various ways, either asynchronously, using non-deterministic FIFOs, or synchronously, via a dedicated scheduler running at *control rate*, typically in a dedicated thread than runs at a lower priority than the DSP one.

A challenging idea suggested by our survey would be to study if and how musical programming languages could improve their event management processes by borrowing from the mathematically-well founded control handling of the sophisticated synchrony traits introduced by synchronous languages, and thus benefit from their formal consistency.

7. CONCLUSION

We performed a practical, use case-oriented survey of 10 key music-specific and general-purpose synchronous programming languages, implementing in each of them a simple yet significant audio processing algorithm, namely a frequency-parameterized oscillator. We believe this survey provides the first bridge between two mature and widely successful computing fields, the more than 50-year old computer music domain and the 40-year old niche of synchronous programming languages. Our work showed that the wide variety of existing music and synchronous languages leads to a large spectrum of program sizes and styles. We believe this application-oriented comparison work, and the discussion points it led to, can be of use to both programmers and language designers interested in synchronous solutions for their problems.

Our present work has focused primarily on programming language design issues. It would be interesting to see whether our findings regarding DSLs' benefits can be leveraged to more complex use case applications. Future work needs also to address the implementation, performance and event management aspects of such a comparison, since these factors are also key in the decisions leading to the choice of a particular language or language paradigm in software projects.

ACKNOWLEDGMENT

The motivation for this study comes from the French ASTREE[13] ANR 2008 CORD 003 01 research project, which addresses the preservation issues of synchronous programs in computer music, using the FAUST real-time audio processing language as the core foundation for such preservation efforts.

REFERENCES

ALTISEN, K., MARANINCHI, F., AND STAUCH, D. 2006. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. *Science of Computer Programming 63,* 3, 297–320.

[11] *Musical Instrument Digital Interface*, an industrial protocol defined in 1983.
[12] *Open Sound Control*, a content format for digital device communication defined at CNMAT in 2002.
[13] ASTREE stands for "*Analyse et synthèse de traitements temps réel*", i.e., "Analysis and Synthesis of Real-Time Processes".

André, C. 1996. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96 IMACS Multiconference: computational engineering in systems applications*. 19–29.

Antonotti, M., Ferrari, A., Flesca, A., and Sangiovanni-Vincentelli, A. 2000. Jester: An Esterel based reactive java extension for reactive embedded systems. In *Forum on specification & Design Languages*.

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. 1999. Computer-assisted composition at Ircam: from PatchWork to OpenMusic. *Computer Music Journal 23,* 3, 59–72.

Bachimont, B., Blanchette, J., Gerzso, A., Swetland, A., Lescurieux, O., Morizet-Mahoudeaux, P., Donin, N., and Teasley, J. 2003. Preserving interactive digital music: a report on the Mustica research initiative. In *Proceedings of the third International Conference on Web Delivering of Music*. IEEE, 109–112.

Baker, A., Johnson, T., Kerpelman, D., and Sutherland, H. 1987. Grafcet and SFC as factory automation standards advantages and limitations. In *American Control Conference*. IEEE, 1725–1730.

Baker, R. and Hiller, L. 1963. Musicomp: MUsic Simulator-Interpreter for COMpositional Procedures for the IBM 7090.

Balarin, F. 1997. *Hardware-software co-design of embedded systems: the POLIS approach*. Vol. 404. Springer Netherlands.

Barkati, K., Fober, D., Letz, S., and Orlarey, Y. 2011. Two recent extensions to the Faust compiler. In *Proceedings of the Linux Audio Conference*.

Benveniste, A. and Berry, G. 1991a. Another look at real-time programming. In *Special Section of the Proceedings of the IEEE*. Vol. 79.

Benveniste, A. and Berry, G. 1991b. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE 79,* 9, 1270–1282.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and De Simone, R. 2003. The synchronous languages twelve years later. *Proceedings of the IEEE 91,* 1, 64–83.

Berry, G. and Cosserat, L. 1985. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*. 389–448.

Bharadwaj, R. 2002. SOL: A verifiable synchronous language for reactive systems. *Electronic Notes in Theoretical Computer Science 65,* 5, 140–154.

Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. 1998. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. ACM, 174–184.

Boudol, G. 2004. ULM: A core programming model for global computing. *Programming Languages and Systems*, 234–248.

Boulanger, R. et al. 2000. *The Csound Book*. Vol. 309. MIT press.

Boussinot, F. 1991. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience 21,* 4, 401–428.

Boussinot, F. and De Simone, R. 1996. The SL synchronous language. *Software Engineering, IEEE Transactions on 22,* 4, 256–266.

Boussinot, F. and Susini, J. 1998. The SugarCubes tool box: a reactive Java framework. *Software: Practice and Experience 28,* 14, 1531–1550.

Bresson, J., Agon, C., and Assayag, G. 2009. Visual Lisp/CLOS programming in OpenMusic. *Higher-Order and Symbolic Computation 22,* 1, 81–111.

Cadoz, C., Luciani, A., and Florens, J. 1993. Cordis-Anima: a modeling and simulation system for sound and image synthesis: the general formalism. *Computer music journal*, 19–29.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. 2003. From simulink to Scade/Lustre to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN Notices 38,* 7, 153–162.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. 1987. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*.

Caspi, P. and Pouzet, M. 1996. Synchronous Kahn networks. In *ACM SIGPLAN Notices*. Vol. 31. 226–238.

Castagné, N. and Cadoz, C. 2002. Genesis : a friendly musician-oriented environment for mass-interaction physical modeling. In *Proceedings of the International Computer Music Conference*. Goteborg, Suede, 330–337.

Chaudhary, A., Freed, A., and Wright, M. 2000. An open architecture for real-time music software. *Proceedings of the International Computer Music Conference*.

Compton, M. 2000. Scoop: An investigation of concurrency in Eiffel. *Master's thesis, Department of Computer Science, The Australian National University*.

Cook, P. and Scavone, G. 1999. The synthesis toolkit (STK). In *Proceedings of the International Computer Music Conference*. 164–166.

Dagum, L. and Menon, R. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE 5,* 1, 46–55.

Dannenberg, R. 1997. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal 21,* 3, 50–60.

Déchelle, F., Borghesi, R., De Cecco, M., Maggi, E., Rovan, B., and Schnell, N. 1999. jMax: an environment for real-time musical applications. *Computer Music Journal 23,* 3, 50–58.

Dormoy, F. 2008. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*. 1–9.

Eckel, G., Iovino, F., and Caussé, R. 1995. Sound synthesis by physical modelling with modalys. In *Proc. International Symposium on Musical Acoustics*. 479–482.

Ellsberger, J., Hogrefe, D., and Sarma, A. 1997. *SDL: formal object-oriented language for communicating systems*. Prentice Hall.

Gamatié, A. 2009. *Designing embedded systems with the Signal programming language: synchronous, reactive specification*. Springer Verlag.

Gaudrain, E. and Orlarey, Y. 2003. A Faust tutorial. Tech. rep., Grame, Lyon.

Gautier, T., Le Guernic, P., and Besnard, L. 1987. Signal: A declarative language for synchronous programming of real-time systems. In *Functional programming languages and computer architecture*. Springer, 257–277.

Gehani, N. and Ramamritham, K. 1991. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems 3,* 4, 377–405.

Giavitto, J.-L. 1991. A synchronous data-flow language for massively parallel computers. In *Proceedings of the International Conference on Parallel Computing'91*, D. J. Evans, G. R. Joubert, and H. Liddell, Eds. London, UK, 391.

Girault, A. 2005. A survey of automatic distribution method for synchronous programs. In *International workshop on synchronous languages, applications and programs, SLAP*. Vol. 5.

Guercio, M., Barthélemy, J., and Bonardi, A. 2007. Authenticity issue in performing arts using live electronics. In *Sound and Music Computing Conference Proceedings*.

Halbwachs, N. 1993. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub.

Halbwachs, N. 2005. A synchronous language at work: the story of Lustre. In *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*. IEEE Computer Society, 3–11.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of computer programming 8,* 3, 231–274.

Harel, D. and Pnueli, A. 1985. *On the development of reactive systems*. Weizmann Institute of Science, Dept. of Computer Science.

Hiller, L. and Baker, R. 1964. Computer Cantata: A study in compositional method. *Perspectives of New Music 3,* 1, 62–90.

Hudak, P., Makucevich, T., Gadde, S., and Whong, B. 1996. Haskore music notation: An algebra of music. *Journal of Functional Programming 6,* 3, 465–483.

IEEE standard. 1988. VHDL language reference manual. *IEEE Std*, 1076–1987.

Ingham, M., Ragno, R., and Williams, B. 2001. A reactive model-based programming language for robotic space explorers. In *Proceedings of ISAIRAS-01*.

Initiative, O. 2006. IEEE standard SystemC language reference manual. *IEEE Computer Society 2002,* March.

Ishikawa, Y., Tokuda, H., and Mercer, C. 1992. An object-oriented real-time programming language. *Computer 25,* 10, 66–73.

Jouvelot, P. and Orlarey, Y. 2011. Dependent vector types for data structuring in multirate faust. *Comput. Lang. Syst. Struct. 37*, 113–131.

Laurson, M. and Duthen, J. 1989. PatchWork, a graphical language in PreForm. In *Proceedings of the International Computer Music Conference*. San Francisco, CA, 172–173.

Laurson, M., Kuuskankare, M., and Norilo, V. 2009. An overview of PWGL, a visual programming environment for music. *Computer Music Journal 33,* 1, 19–31.

Lavagno, L. and Sentovich, E. 1999. ECL: a specification environment for system-level design. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, 511–516.

Le Guernic, P., Talpin, J. P., and Le Lann, J. C. 2003. Polychrony for system design. *Journal of circuits systems and computers 12,* 3, 261–304.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. 2010. The Objective Caml system release 3.12 documentation and user's manual. Tech. rep., INRIA.

Loy, G. and Abbott, C. 1985. Programming languages for computer music synthesis, performance, and composition. *ACM Computing Surveys (CSUR) 17,* 2, 235–265.

Mallet, F. and André, C. 2009. On the semantics of UML/MARTE clock constraints. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 305–312.

Mandel, L. and Pouzet, M. 2005. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, 82–93.

Manna, Z. and Pnueli, A. 1995. *Temporal verification of reactive systems: safety*. Vol. 2. Springer Verlag.

Maraninchi, F. 1990. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. In *Automatic Verification Methods for Finite State Systems*. 38–53.

Maraninchi, F. 1991. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*.

Mathews, M., Miller, J., Moore, F., Pierce, J., and Risset, J. 1969. *The technology of computer music*. The MIT Press, Boston.

McCartney, J. 1996. SuperCollider, a new real time synthesis language. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, 257–258.

Mernik, M., Heering, J., and Sloane, A. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR) 37,* 4, 316–344.

Milner, R. 1980. *A calculus of communicating systems*. Vol. 92. Springer-Verlag.

Orlarey, Y., Fober, D., and Letz, S. 1997. Elody: A Java + MidiShare based music composition environment. In *Proceedings of the International Computer Music Conference*. Thessaloniki, Greece.

Orlarey, Y., Fober, D., and Letz, S. 2002. An algebra for block diagram languages. In *Proceedings of International Computer Music Conference*. 542–547.

Orlarey, Y., Fober, D., and Letz, S. 2004. Syntactical and semantical aspects of Faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications 8,* 9, 623–632.

Orlarey, Y., Fober, D., and Letz, S. 2009. Faust: an efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music*, A. G. and G. A., Eds. IRCAM/Delatour France.

Pnueli, A., Shtriehman, O., and Siegel, M. 1998. Translation validation for synchronous languages. *Automata, Languages and Programming*, 235–246.

Pop, A. 2011. Leveraging streaming for deterministic parallelization : an integrated language, compiler and runtime approach. Ph.D. thesis, MINES ParisTech.

Pop, A. and Cohen, A. 2011. A stream-computing extension to OpenMP. In *Proc. of the 6th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*.

Puckette, M. 1991. Combining event and signal processing in the Max graphical programming environment. *Computer music journal*, 68–77.

Puckette, M. 1996. Pure Data: Another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, 37–41.

Puckette, M. 2002. Max at seventeen. *Computer Music Journal 26,* 4, 31–43.

Puckette, M. 2007. *The Theory and Technique of Electronic Music.* World Scientific Pub Co Inc.

Reppy, J. 1999. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, England.

Rodet, X., Potard, Y., and Barriere, J. 1984. The Chant project: from the synthesis of the singing voice to synthesis in general. *Computer Music Journal 8,* 3, 15–31.

Scaletti, C. 1987. Kyma: An object-oriented language for music composition. In *Proceedings of the International Computer Music Conference*. 49–56.

Scheirer, E. and Vercoe, B. 1999. SAOL: The MPEG-4 structured audio orchestra language. *Computer Music Journal 23,* 2, 31–51.

Schneider, K. 2000. A verified hardware synthesis of Esterel programs. In *Proceedings of the IFIP WG10*. Kluwer, BV, 205–214.

Schneider, K. 2004. *Verification of reactive systems: formal methods and algorithms*. Springer Verlag.

Schneider, K. and Schuele, T. 2005. Averest: Specification, verification, and implementation of reactive systems. In *Conference on Application of Concurrency to System Design (ACSD)*.

Smith III, J. 2010. Audio signal processing in Faust. Tech. rep., CCRMA.

Sorensen, A. 2005. Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2009*.

Taube, H. 1991. Common Music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 21–32.

Teehan, P., Greenstreet, M., and Lemieux, G. 2007. A survey and taxonomy of GALS design styles. *Design & Test of Computers, IEEE 24,* 5, 418–428.

Thies, W., Karczmarek, M., and Amarasinghe, S. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Grenoble, France.

Thomas, D. and Moorby, P. 2002. *The Verilog hardware description language*. Vol. 1. Springer Netherlands.

Valle, A. et al. 2007. The SuperCollider help book.

Van Deursen, A., Klint, P., and Visser, J. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices 35,* 6, 26–36.

Van Roy, P. 2009. Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music*.

Vercoe, B. 1992. *Csound: a manual for the audio processing system and supporting programs with tutorials*. Massachusetts Institute of Technology.

Vercoe, B. et al. 2007. The canonical Csound reference manual.

Wang, G. and Cook, P. 2007. *The ChucK Manual 1.2.1.3*. Princeton University.

Wang, G., Cook, P., et al. 2003. ChucK: A concurrent, on-the-fly audio programming language. In *Proceedings of the International Computer Music Conference*. 219–226.

Wright, M. 2005. Open sound control: an enabling technology for musical networking. *Organised Sound 10,* 03, 193–200.

Zicarelli, D. 1998. An extensible real-time signal processing environment for Max. In *Proceedings of the International Computer Music Conference*. 463–466.

Received Month Year; November Year; accepted Month Year.