Kronos Meta-Sequencer – From Ugens to Orchestra, Score and Beyond

Vesa Norilo
Centre for Music & Technology
University of Arts Helsinki
vno11100@uniarts.fi

ABSTRACT

This article discusses the Meta-Sequencer, a circular combination of an interpreter, scheduler and a JIT compiler for musical programming. Kronos is a signal processing language focused on high computational performance, and the addition of the Meta-Sequencer extends its reach upwards from unit generators to orchestras and score-level programming. This enables novel aspects of temporal recursion – a tight coupling of high level score abstractions with the signal processors that constitute the fundamental building blocks of musical programs.

1. INTRODUCTION

Programming computer systems for music is a diverse practice; it encompasses everything from fundamental synthesis and signal processing algorithms to representing scores and generative music; from carefully premeditated programs for tape music to performative live coding.

One estabilished classification of musical programming tasks, arising from the MUSIC-N tradition [1, pp. 787-796] [2], identifies three levels of abstraction:

1. Unit Generator
2. Orchestra
3. Score

Unit Generators are the fundamental building blocks of musical programs, including oscillators, filters and signal generators. Orchestras are ensembles of Unit Generators, coordinated to behave as musical instruments. Finally, scores encode control information – a high level representation of a piece to be performed by the Unit Generator Orchestra. Most MUSIC-N family languages are based on distinct domain languages for orchestras and scores; programming unit generators from scratch is rarely addressed.

This paper addresses the problem of tackling all three levels of hierarchy in a single programming language. It is based on extending Kronos [3], a functional reactive signal processing language, with the notion of a task scheduler and a script interpreter capable of driving each other. This notion enables a powerful expression of score metaphors, including temporal recursion [4]. This is the concept of the Meta-Sequencer – a programmable sequencer capable of reprogramming itself.

2. BACKGROUND

Contemporary musical programming languages often blur the lines between ugen, orchestra and score. Semantically, languages like Max [5] and Pure Data [6] would seem to provide just the Orchestra layer; however, they typically come with specialized unit generators that enable score-like functionality. Recently, Max has added a sublanguage called Gen to address ugen programming.

SuperCollider [7] employs an object-oriented approach of the SmallTalk tradition to musical programming. It provides a unified idiom for describing orchestras and scores via explicit imperative programs.

ChucK [8] introduces timing as a first class language construct. ChucK programs consist of Unit Generator graphs with an imperative control script that can perform interventions at precisely determined moments in time. To draw an analogy to MUSIC-N, the control script is like a score – although more expressive – while the unit generator graph resembles an orchestra. The ChucK model can also extend to "natively constructed" ugens [8, pp. 25-26].

Languages specifically focused on ugens are both fewer and more recent. Faust [9] is a prominent example, utilizing functional programming and block diagram algebra to enable compact descriptions of unit generators while maintaining high computational efficiency. The functional model is a good fit for computationally efficient signal processing: its traits, such as immutable values, referential transparency and suitability for equational reasoning [10] enable a high degree of compiler optimization. My own prior work with Kronos [3] is inspired by the Faust model, seeking to contribute mixed-rate and event-driven systems as well as type-based polymorphism and metaprogramming.

The problem of combining all three levels in a single language is challenging yet intriguing. Successful orchestra/score languages like Max and ChucK have some facilities for ugen programming [8, pp. 25-26]. The respective tradeoffs include semantics that differ from the rest of the environment, and computational efficiency far below machine limits. Brandt has studied ugen-type programming with temporal type constructors [11] and related tradeoffs, such as limitations in program semantics and a lack of real-time capabilities.

This study approaches the problem from the opposite direction: extending Kronos [3], a signal-processing, ugen-orchestra-focused language upward to provide score capability. This is achieved by a novel, embedded domain language inspired by the I/O Monad in Haskell [12] and the concept of temporal recursion [4].

3. META-SEQUENCER

The project that became the Kronos Metasequencer originated in an effort to improve the expressibility of outputs in Kronos programs. While functional programming [13] excels in expressing data flows and signal topologies, it is less suitable for modeling any *effects* the program should have on its surroundings. Functional programs do not encode state, but state is present in all the relevant input/output devices attached to computers. Programs must mutate that state in order to be observable (and potentially useful) to their users.

One approach to the problem is to externalize the I/O concerns. Faust [9] programs are assumed to output an audio stream. As Kronos [3] extends signal processing to event models such as MIDI and OSC [14], a more complicated solution is required. The specification of the signal destination can still be externalized: whether a program should output audio, OSC events, MIDI messages or text onto a console could be supplied to the compiler as additional parameters, not indicated in the source code in any way.

The second, more refined approach could involve type polymorphism: an appropriate output destination would be determined based on the data type of the output. Programs could specify the desired output behavior simply by returning a type such as a MIDI event. While this approach has benefits, the set of types becomes complicated as the variety of output methods grows. In addition, the compiler driver must interpret all the types: the output specification quickly starts to resemble a mini-language of its own.

3.1 Haskell and I/O

It is instructive to look at how a pure, general purpose functional language like Haskell [12] encodes I/O operations. At first glance, I/O code in Haskell appears imperative: the syntax evokes assignment and side-effectful read and write operations. A simple example from the Haskell wiki is shown in Listing 1. Despite the code appearing imperative, functional purity is not compromized: referential transparency [10] and equational reasoning remain in force.

Listing 1. Simple Haskell program with I/O

```
main = do a <- ask "What is your name?"
          b <- ask "How old are you?"
          return ()

ask s = do putStr s
           readLn
```

This implementation is powered by the I/O monad, which provides a functional representation of an imperative program, modeling the stateful effects caused by I/O actions as an implicit data flow. Monadic I/O code is a domain language within Haskell.

3.2 The I/O Domain Language

Various domain languages built in Kronos already exist; these range for small-scale experiments such as document generation to published work on graphics and animation [15]. This section describes a domain language for I/O, capable of enabling semantics such as evoked by Listing 2.

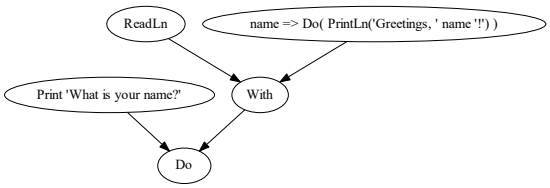

Figure 1. Abstract Syntax Tree of an Imperative Kronos Program

Listing 2. Kronos program with I/O

```
Greetings() {
  Use Actions
  Greetings = Do(
     Print("What is your name? ")
     name <- ReadLine()
     PrintLn("Greetings, " name "!") )
}
```

3.2.1 The Imperative Interpreter

As metaprogramming is one of Kronos' fundamental principles [3], the I/O domain language is based on the concept of second-order code generation: a functional dataflow program constructing the syntax tree of an imperative program. Data types are used to encode the *abstract syntax tree* of the imperative program. For example, the AST generated by running Listing 2 is shown in Figure 1.

Similar to how I/O Actions in Haskell are effective only at the root of the program entry point, the I/O language is designed around an interpreter hook placed at the very end of the program data flow. It is implemented as a foreign-function call to the compiler driver written in C++. The interpreter will then traverse the AST, executing the effectful Print and ReadLn nodes.

3.2.2 Enabling Assignment Semantics

There is a non-obvious detail in the AST shown in Figure 1: the left-arrow syntax, simulating assignment, has been translated to a node called 'With'. This node receives the I/O action whose result value is bound to 'name', and a closure encompassing the remaining I/O actions that were sequenced after it. The AST interpreter will invoke the I/O action passed as the value, and invoke the closure with the result of the I/O action as a parameter. If the result of the closure is another I/O action, the interpreter will then process that action.

To illustrate the assignment transformation, Listing 3 shows how Listing 2 is lowered.

Listing 3. Example of Assignment Transformation

```
; after left-arrow transformation
Do(
  Print("What is your name? ")
  Invoke-With(
     ReadLine()
     name => PrintLn("Greetings, " name "!")))
```

3.2.3 The Interpreter as Compiler Driver

It is noteworthy that the closures shown in Listing 3 are constructs of the core Kronos language. The interpreter

does not know how to compute: all numeric work is delegated to the dataflow compiler. This includes generation of the interpreter ASTs: the closure shown in the transformed version actually returns an imperative program to print 'name'.

Execution of programs with actions is essentially a cycle of interpretation of an imperative AST, alternating with compilation and execution of pure functional code, which may produce a new imperative AST.

For performance reasons, Kronos programs are statically typed. However, the Kronos methodology is based on type-generic programs: there are very rarely any type annotations in the source code, as the compiler will estabilish the types through whole-program type derivation.

As the interpreter drives JIT compilation, also providing the root types for the compiler, it effectively appears to the user as a dynamically typed language, where type-specific routines are compiled on demand.

3.2.4 Control Flow

An important aspect of any imperative scripting is control flow – decision points where the program flow could diverge based on run-time conditions. While such control flow is highly toxic to high performance signal processing, it is essential for many score-level tasks.

For these purposes, the imperative language contains an If-node, structured in the well-known format of truth value – then-branch – else-branch. The AST interpreter will retrieve the truth value and based on it, proceed on to either the then-branch or the else-branch.

Please recall that 'If' is a normal function. That means, on one hand, that it can be used in the variety of ways functions can: composed, applied partially, passed as a parameter value, and so on. On the other hand, data flow demands that all of its upstream children, truth-value, then-branch and else-branch must be evaluated prior to it.

With this in mind, please consider a looping structure, such as shown in Listing 4. Because one side of the conditional branch refers recursively back to itself, a straightforward implementation of 'If' would result in an infinitely deep AST and nontermination. A common strategy to address this problem in functional programming is lazy evaluation [13, p. 384], while imperative languages favor short-circuiting or minimal evaluation. Both require specific support from the compiler.

Listing 4. Recursion and Control Flow

```
Countdown(count) {
  Use Actions
  Countdown =
    If(count > 0
       { Do(
          PrintLn(count)
          Countdown(count - 1)) }
       { PrintLn("Done") })
}
```

Kronos can support a rudimentary form of explicit lazy evaluation by specifying that the then-branch and the else-branch are in fact closures, and should return the AST to be taken by the interpreter. As anonymous functions can be written simply by enclosing statements in curly braces, the resulting syntax should be quite familiar to programmers. An example of a recursive looping program is shown in Listing 4. Without the intermediate closures, this program would fail by getting stuck in an infinite loop in the AST generation stage.

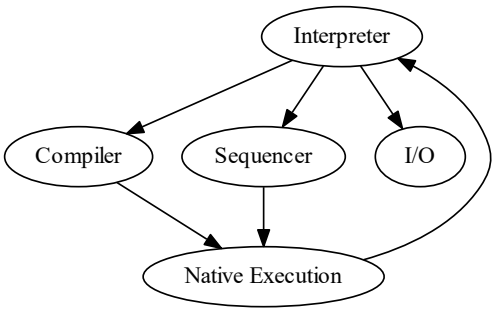

Figure 2. Meta-Sequencer Program Flow

3.3 Temporal Recursion: Meta-Sequences

So far, the imperative language features discussed in this paper have little relevance to computer music, as they are little more than the staples of imperative programming defined in a functional dataflow language designed for DSP.

However, a simple addition to the interpreter-compiler-execution cycle will bring about a significant expansion to musical possibilities. Kronos already features a sequencer for timed reactive events [16]. Extending that sequencer to schedule and fire imperative programs is a logical evolution. If, in addition, the imperative programs gain facility to program the sequencer, the expressive power of the system grows significantly.

This is the concept of the Meta-Sequencer: a fusion of an interpreter, sequencer and a JIT compiler. The program flow is shown in Figure 2. The interpreter traverses an AST, directing it to fire I/O events or to compile a Kronos function for execution either directly or as scheduled by a sequencer. The compiled function may contain an interpreter hook, cycling back to the interpreter for further actions.

In fact, this closely follows the concept of temporal recursion as presented by Sorensen [4], and related concepts in the literature [17, 11, 18].

3.4 I/O Actions in Detail

This section summarizes the imperative I/O Action language and the primitives of its AST, which are displayed in Table 1.

3.4.1 After

After is the scheduling command. It can be used to schedule an arbitrary AST for execution after a specified period in seconds. Scheduling is sample-accurate and synchronous with the audio stream.

3.4.2 Send

Send represents a discrete output event. The arguments to this command are an address pattern and value. The ad-

Table 1. I/O Actions in Kronos

Action	Arguments	Description
After	*time fn*	*run 'fn' 'time' secs later*
Do	*actions*	*run 'actions' sequentially*
For	*values fn*	*apply 'fn' to each element in 'values'*
Invoke-With	*action fn*	*Pass result of 'action' to 'fn'*
If	*p t e*	*If 'p' is true, invoke 't'hen to obtain a new AST; else invoke 'e' for it.*
Send	*address value*	*Output 'value' to 'address'*
Send-To	*id addr val*	*Send 'val' to method 'addr in instance 'id'*
Print	*value*	*Send("#pr" value)*
PrintLn	*value*	*Do(Print(value) Print("\n"))*
ReadLine		*Read line from console and return as a string*
Start	*fn*	*Start 'fn' as a reactive instance return an instance id.*
Stop	*id*	*Stops the instance 'id'*

dress pattern determines the output method. An URI-type scheme is used here: for example, OSC [14] outputs can be specified by "osc://ip:port/osc/address/pattern". The *Print* command utilizes *Send*, specifying an address pattern reserved for console output.

While arbitrary values can be passed to *Send*, the output method may not be able to handle all data types. The OSC encoder can handle primitive numbers, strings, truth values as scalars and nested arrays, but more complicated types such as closures are not supported.

3.4.3 Start, Stop, Send-To

Start instantiates a Kronos closure as a *reactive object*, responding to reactive inputs and producing a stream of outputs. Each instance is a *discrete reactive system* according to the classification presented by Van Roy [19].

The return value of the *Start* command is an instance handle. The referred instance can be stopped by passing the handle to *Stop*. This can be done by the top-level REPL or any script that fires within the sequencer. The handle is also passed to the closure itself, enabling it to stop itself.

Send-To is a convenience function that works like *Send*, but addresses an input within a specific instance identified by a handle.

4. APPLICATIONS AND IMPLICATIONS

4.1 Reactive Event Processors

The Kronos signal model is based on reactive update propagation [16]. The imperative ASTs participate in this signal model – if a reactive signal feeds a leaf of the AST, it effectively becomes an event handler for that signal. This results in a very simple definition of an OSC [14] monitor, shown in Listing 5.

Listing 5. Reactive OSC Monitor

```
; listen to float values at OSC address pattern '/a'
Start( { PrintLn(Control:Param("/a" 0)) } )
```

This instance will print a line of text representing each OSC method call that supplies a floating point value to address "/a". Signal-flow-wise, *Control:Param* returns a float scalar, which *PrintLn* translates into an imperative program to print said scalar. This is in turn sent to the interpreter via the interpreter hook implicitly placed at the root of the closure. Reactivity flows downstream from *Control:Param*, so the interpreter hook fires whenever there is OSC input.

4.2 Generative Sounds

The next example, Generative Sound, utilizes temporal recursion to construct a sonic fractal. The code is shown in Listing 6. The fractal plays a sinusoid for a specified duration, spawning delayed, recursive copies of itself to generate increasingly dense partials.

Listing 6. Sonic Fractal

```
Import Gen

Fractal(f dur g) {
  Use Actions

  ; time offset to next cluster
  time-offset = Math:Sqrt(dur)

  ; its duration is the remaining time
  next-dur = dur - time-offset

  Fractal = Do(
     ; start sinusoid at frequency 'f'
     id <- Start( { Wave:Sin(f) * g } )

     ; stop it after 'dur' seconds
     After( dur Stop( id ) )

     ; spawn two more fractals at musical intervals
     ; of 2/3 and 6/5, after time offset
     If( dur > 0.5
       { After( time-offset
         Fractal(f * 2 / 3 next-dur g / 2)
         Fractal(f * 6 / 5 next-dur g / 2) ) } )
   )
}
```

The fractal could be made more musically interesting with features such as randomized offsets or additional timbre parameters. Even the simple form demonstrates the generative power of temporal recursion.

An additional benefit of the fractal is benchmarking: recall that the control script is both sample accurate and audio-synchronous; in real-time playback, this has a significant computational impact when a high number of closures are scheduled to be compiled and played back at once.

In informal benchmarking, constructing and connecting an instance (after initial warm-up) in Listing 6 happens in $30\mu s$ on a laptop with Intel Core i7-4500U processor. Sinusoid synthesis is computationally cheap, so instantiation is the main constraint on real-time playback. As the fractal features 2^N sinusoids at step N, the software must perform a corresponding number of instantiations sample-synchronously in addition to sound synthesis. On the Core i7-4500U, it can achieve 9 steps or up to 512 instantiations. Polyphony could be increased by staggering the instantiations in time, increasing latency (and thus amortization) or grouping several sinusoids in a single instance – using oscillator banks.

4.3 Score Auralization

The final example, in Listing 7 demonstrates a simplistic MUSIC-N descendant [2] system written entirely as a single Kronos program. The program defines three functions: a unit generator *(Exp-Gen)*, an instrument *(MyInstr)* and a score player/transformer *(MyPlayer)*. The score is defined as a matrix value *(MyScore)*.

Exp-Gen is the sole representative of Kronos' core capability: signal processing. It is an exponential function generator working at audio rate, consisting of a mutiplier and unit delay feedback. In this example it is used with complex-valued parameters, resulting a machine code procedure with just a handful of instructions per generated sample. *Exp-Gen* returns a reactive stream of floating point scalars.

MyInstr is an instrument wrapper for the *Exp-Gen* generator, receiving high level parameters for duration, pitch and amplitude. It computes complex coefficients based on them, instantiates the unit generator and schedules it to stop after the amplitude has decayed sufficiently. *MyInstr* returns an I/O action that performs these steps.

MyPlayer applies *MyInstr* to the notes in the score. The score is a list of 4-value tuples. The first value indicates the start time of a note, followed by the parameters required by *MyInstr*, duration, note number and amplitude. The start time is consumed by the player, used to schedule instrument invocations with the *After* command. The rest of the tuple is passed directly on to the instrument invocation as parameters.

The layers presented are intentionally simplistic. Different parametrizations and hierarchies can be devised for abstractions like multi-timbral scores, nested scores or real-time capable instruments. For example, with a suitable nested score format, *MyPlayer* could schedule instances of itself that in turn schedule sub-scores. Such flexibility is the result of the general purpose capability of the Meta-Sequencer: with a handful of I/O hooks, an interpreter and a closely integrated high performance JIT compiler, temporal recursion [4] is sufficient for a wide range of musical constructs.

Listing 7. Simple Ugen, Instrument and Score

```
Import Complex
Import Actions

; Unit generator: output an exponential function
; Can produce a sinusoid with complex-valued params
Exp-Gen(init coef) {
  state = z-1(init state * Audio:Signal(coef))
  Exp-Gen = state
}

; Instrument: configure, start and stop the ugen
MyInstr(dur pitch amp) {
  Use Actions
  Use Math

  ; compute complex coefficients
  fsr = Audio:Rate()
  ; angular frequency from note number
  w = Pi * 880 * Pow(2 (pitch - 69) / 12) / fsr
  ; radius; decay of 1/100 in 'dur' time
  r = Pow(0.01 1 / (dur * fsr))
  coef = Complex:Polar(w r)
  init = Complex:Cons(0 amp)
  ; instantiate an ugen and stop it after
  ; twice the duration (decay of 1/10000)
  MyInstr = Do(
    id <- Start({
      Complex:Real(Exp-Gen(init coef)) })
    After(duration * 2 Stop(id))
  )
}

; Score format: <time> <duration> <note-number> <amplitude>
MyScore =
 [(0 3 60 1 )
  (1 2 64 1 )
  (2 1 67 1 )
  (3 0.1 72 0.5)
  (3.1 0.1 71 0.4)
  (3.2 0.1 70 0.3)
  (3.3 0.1 69 0.2)
  (3.4 0.1 68 0.1)]

; Construct and schedule 'MyInstr' instance for each
; note in the score.
MyPlayer(score tempo-scale instr) {
  Use Actions
  MyPlayer = For(score (time params) =>
     After(time * tempo-scale instr(params)))
}
; Usage: MyPlayer(MyScore 1 MyInstr)
```

4.4 Compiler Stack and Real-Time Playback

The Meta-Sequencer is sample-accurate and audio synchronous; the implication is that sometimes, JIT compilation must interrupt the real-time audio thread. Even simple code takes time to travel through the full LLVM stack; at minimum, compile times are in the order of milliseconds, making it hard to sustain uninterrupted real time playback.

However, the Meta-Sequencer is capable of synthesizing a surprising range of algorithms in real time, allowing for a small delay in the initial response. This is because of type determinisim in the Kronos language [3]: the compiler output depends uniquely on the type of the closure being compiled. This allows memoization of compiled closures based on their type, effectively reducing compilation of already-known closures to a simple hash table lookup.

The implication is important for a concept of *type loops* in temporal recursion. The type of each closure if determined by its captures and arguments. For example, the type loop in Listing 6 is closed: each recursion is *type-invariant* in its captures and arguments. In such a case, no additional compilation is required once the *type loop* has been completed.

5. FUTURE WORK

The introduction of the I/O language and temporally recursive sequencer extend the reach of the Kronos programming language upwards from signal processing towards representations of music and scores. This study describes the fundamentals required for such an extension; much work remains in fulfilling the nascent potential.

Immediate technical concerns include the compilation performance, as discussed in Secton 4.4. An interesting enhancement to the system would be analyze any scheduled ASTs for closures that could be compiled anticipatively.

Core Kronos features dynamic as well as static compilation. However, the AST interpreter requires a runtime

component that is so far absent from statically compiled binaries. It is viable to produce such a run time and produce dependency-free binaries from Meta-Sequencer programs with *closed type loops* (see Section 4.4).

The development of the I/O Action language and related usability aspects is also an interesting avenue for future work. Enhancing the Kronos core library towards score metaphors and any potential problems thus uncovered in the compiler design represent an important strategy of incremental improvement.

Graphical representation of imperative programs, as well as integration to GUI tools, including PWGL and ENP [20] remain compelling.

6. CONCLUSIONS

This study presented the Meta-Sequencer, an extension to the Kronos programming language [3]. The implemetation of an I/O Action Language, sourcing from the concepts in the Haskell [12] I/O Monad, was discussed. The implications for musical applications, especially with the addition of temporal recursion [4] were explained and demonstrates.

The study represents an attempt to extend a signal processing language, previously focused on unit generator and orchestra programming towards scores and musical abstractions. Kronos is an ideal platform for such a work, as it focuses on meta-programming, extensibility and domain languages.

Acknowledgments

Vesa Norilo's work has been supported by the Emil Aaltonen Foundation.

7. REFERENCES

[1] C. Roads, *the Computer Music Tutorial*. Cambridge: MIT Press, 1996.

[2] V. Lazzarini, "The Development of Computer Music Programming Systems," *Journal of New Music Research*, vol. 42, no. 1, pp. 97–110, Mar. 2013. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/09298215.2013.778890

[3] V. Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, vol. 39, no. 4, 2015.

[4] A. Sorensen and H. Gardner, "Programming With Time Cyber-physical programming with Impromptu," *Time*, vol. 45, pp. 822–834, 2010. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869526

[5] M. Puckette and D. Zicarelli, *MAX - An Interactive Graphical Programming Environment*. Opcode Systems, 1990.

[6] M. Puckette, "Pure data: another integrated computer music environment," in *Proceedings of the 1996 International Computer Music Conference*, 1996, pp. 269–272.

[7] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[8] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal2*, vol. 39, no. 4, pp. 10–29, 2015.

[9] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[10] C. Strachey, "Fundamental Concepts in Programming Languages," *Higher-Order and Symbolic Computation*, vol. 13, no. 1-2, pp. 11–49, 2000.

[11] E. Brandt, "Temporal type constructors for computer music programming," Ph.D. dissertation, Carnegie Mellon University, 2002.

[12] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler, "A history of Haskell," *Proceedings of the third ACM SIGPLAN conference on History of programming languages HOPL III*, pp. 12–1–12–55, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1238844.1238856

[13] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[14] M. Wright, A. Freed, and A. Momeni, "OpenSound Control: State of the Art 2003," in *Proceedings of NIME*, Montreal, 2003, pp. 153–159.

[15] V. Norilo, "Visualization of Signals and Algorithms in Kronos," in *Proceedings of the International Conference on Digital . . .* , York, 2012, pp. 15–18.

[16] ——, "Introducing Kronos - A Novel Approach to Signal Processing Languages," in *Proceedings of the Linux Audio Conference*, F. Neumann and V. Lazzarini, Eds. Maynooth: NUIM, 2011, pp. 9–16.

[17] R. B. Dannenberg, "Expressing Temporal Behavior Declaratively," in *CMU Computer Science, A 25th Anniversary Commemorative*, R. F. Rashid, Ed. ACM Press, 1991, pp. 47–68.

[18] G. Wakefield, W. Smith, and C. Roberts, "LuaAV: Extensibility and Heterogeneity for Audiovisual Computing," *Proceedings of the Linux Audio Conference*, 2010. [Online]. Available: https://mat.ucsb.edu/Publications/wakefield_smith_roberts_LAC2010.pdf

[19] P. Van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know," in *New Computational Paradigms for Music*, G. Assayag and A. Gerzso, Eds. Paris: Delatour France, IRCAM, 2009, pp. 9–49.

[20] M. Laurson, M. Kuuskankare, and V. Norilo, "An Overview of PWGL, a Visual Programming Environment for Music," *Computer Music Journal*, vol. 33, no. 1, pp. 19–31, 2009.