Kronos: A Declarative Metaprogramming Language for Digital Signal Processing

Article *in* Computer Music Journal · December 2015

DOI: 10.1162/COMJ_a_00330

1 author:

Vesa Norilo
University of the Arts Helsinki
31 PUBLICATIONS **161** CITATIONS
SEE PROFILE

Vesa Norilo
Centre for Music and Technology
University of Arts Helsinki, Sibelius
Academy
PO Box 30
FI-00097 Uniarts, Finland
vnorilo@siba.fi

Kronos: A Declarative Metaprogramming Language for Digital Signal Processing

Abstract: Kronos is a signal-processing programming language based on the principles of semifunctional reactive systems. It is aimed at efficient signal processing at the elementary level, and built to scale towards higher-level tasks by utilizing the powerful programming paradigms of "metaprogramming" and reactive multirate systems. The Kronos language features expressive source code as well as a streamlined, efficient runtime. The programming model presented is adaptable for both sample-stream and event processing, offering a cleanly functional programming paradigm for a wide range of musical signal-processing problems, exemplified herein by a selection and discussion of code examples.

Signal processing is fundamental to most areas of creative music technology. It is deployed on both commodity computers and specialized sound-processing hardware to accomplish transformation and synthesis of musical signals. Programming these processors has proven resistant to the advances in general computer science. Most signal processors are programmed in low-level languages, such as C, often thinly wrapped in rudimentary C++. Such a workflow involves a great deal of tedious detail, as these languages do not feature language constructs that would enable a sufficiently efficient implementation of abstractions that would adequately generalize signal processing. Although a variety of specialized musical programming environments have been developed, most of these do not enable the programming of actual signal processors, forcing the user to rely on built-in black boxes that are typically monolithic, inflexible, and insufficiently general.

In this article, I argue that much of this stems from the computational demands of real-time signal processing. Although much of signal processing is very simple in terms of program or data structures, it is hard to take advantage of this simplicity in a general-purpose compiler to sufficiently optimize constructs that would enable a higher-level signal-processing idiom. As a solution, I propose a highly streamlined method for signal processing, starting from a minimal dataflow language that can describe the vast majority of signal-processing tasks with a handful of simple concepts. This language is a good fit for hardware—ranging from CPUs to GPUs and even custom-made DSP chips—but unpleasant for humans to work in. Human programmers are instead presented with a very high-level metalanguage, which is compiled into the lower-level data flow. This programming method is called Kronos.

Computer Music Journal, 39:4, pp. 30–48, Winter 2015
doi:10.1162/COMJ_a_00330

Musical Programming Paradigms and Environments

The most prominent programming paradigm for musical signal processing is the unit generator language (Roads 1996, pp. 787–810). Some examples of "ugen" languages include the Music N family (up to the contemporary Csound; see Boulanger 2000), as well as Pure Data (Puckette 1996) and SuperCollider (McCartney 2002).

The Unit Generator Paradigm

The success of the unit generator paradigm is driven by the declarative nature of the ugen graph; the programmer describes data flows between primitive, easily understood unit generators.

It is noteworthy that the typical selection of ugens in these languages is very different from the primitives and libraries available in general-purpose languages. Whereas languages like C ultimately consist of the data types supported by a CPU and the primitive operations on them, a typical ugen could be anything from a simple mathematical operation to a reverberator or a pitch tracker. Ugen languages tend to offer a large library of highly specialized ugens rather than focusing on a small orthogonal set of primitives that could be used to express any desired program. The libraries supplied with most general-purpose languages tend to be written in those languages; this is not the case with the typical ugen language.

The Constraints of the Ugen Interpreter

I classify the majority of musical programming environments as *ugen interpreters*. Such environments are written in a general-purpose programming language, along with the library of ugens that are available for the user. These are implemented in a way that allows late binding composition: the ugens are designed to be able to connect to the inputs and outputs of other, arbitrary ugens. This is similar to how traditional program interpreters work—threading user programs from predefined native code blobs—hence the term ugen interpreter.

In this model, ugens must be implemented via parametric polymorphism: as a set of objects that share a suitable amount of structure, to be able to interconnect and interact with the environment, regardless of the exact type of the ugen in question. Dynamic dispatch is required, as the correct signal-processing routine must be reached through an indirect branch. This is problematic for contemporary hardware, as the hardware branch prediction relies on the hardware instruction pointer: In an interpreter, the hardware instruction pointer and the interpreter program location are unrelated.

A relevant study (Ertl and Gregg 2007) cites misprediction rates of 50 percent to 98 percent for typical interpreters. The exact cost of misprediction depends on the computing hardware and is most often not fully disclosed. On a Sandy Bridge CPU by Intel, the cost is typically 18 clock cycles; as a point of reference, the chip can compute 144 floating-point operations in 18 cycles at peak throughput. There are state-of-the-art methods (Ertl and Gregg 2007; Kim et al. 2009) to improve interpreter performance. Most musical programming environments choose instead a simple, yet reasonably effective, means of mitigating the cost of dynamic dispatch. Audio processing is vectorized to amortize the cost of dynamic dispatch over buffers of data instead of individual samples.

Consider a buffer size of 128 samples, which is often considered low enough to not interfere in a real-time musical performance. For a ugen that semantically maps to a single hardware instruction, misprediction could consume 36 percent to 53 percent of the time on a Sandy Bridge CPU, as derived from the numbers previously stated. To reduce the impact of this cost, the ugen should spend more time doing useful work.

Improving the efficiency of the interpreter could involve either increasing the buffer size further or increasing the amount of useful work a ugen does per dispatch. As larger buffer sizes introduce latency, ugen design is driven to favor large, monolithic blocks, very unlike the general-purpose primitives most programming languages use as the starting point, or the native instructions of the hardware.

In addition, any buffering introduces a delay equivalent to the buffer size to all feedback connections in the system, which precludes applications such as elementary filter design or many types of physical modeling.

An Ousterhout Corollary

John Ousterhout's dichotomy claims that programming languages are either *systems programming* languages or *scripting languages* (Ousterhout 1998). To summarize, the former are statically typed, and produce efficient programs that operate in isolation. C is the prototypical representative of this group. The latter are dynamically typed, less concerned with efficiency, intended to glue together distinct components of a software system. These are represented by languages such as bash, or Ousterhout's own Tcl.

The Ousterhout dichotomy is far from universally accepted, although an interesting corollary to musical programming can be found. Unit generators are the static, isolated, and efficient components in most musical programming languages. They are typically built with languages aligned with Ousterhout's systems programming group. The scripting group is mirrored by the programming surfaces such as the patching interface described by Miller Puckette (1988) or the control script languages in ChucK (Wang and Cook 2003) or SuperCollider. Often, these control languages are not themselves capable of implementing actual signal-processing routines with satisfactory performance, as they focus on just acting as the glue layer between black boxes that do the actual signal processing.

A good analysis of the tradeoffs and division of labor between "systems" languages and "glue" languages in the domain of musical programming has been previously given by Eli Brandt (2002, pp. 3–4). Although the "glue" of musical programming has constantly improved over the last decades, the "systems" part has remained largely stagnant.

Beyond Ugens

To identify avenues for improvement, let us first examine a selection of musical programming languages that deviate from the standard ugen interpreter model.

Common Lisp Music: Transcompilation

Common Lisp Music (CLM, see Schottstaedt 1994) is an implementation of the Music N paradigm in Common Lisp. Interestingly, CLM attempts to facilitate the writing of signal-processing routines in Lisp, a high-level language. This is accomplished by means of transcompilation: CLM can generate a C-language version of a user-defined instrument, including compiler-generated type annotations, enabling robust optimization and code generation.

Only a narrow subset of Lisp is transcompiled by CLM, however. This subset is not, in fact, significantly different from low-level C—a lower level of abstraction than in standard C++. Indeed, CLM code examples resemble idiomatic C routines, albeit written in S-expressions. Although the metaprogramming power of Lisp could well be utilized to generate a transcompilable program from a higher-level idiom, this has not been attempted in the context of CLM.

Nyquist: Signals as Values

Nyquist (Dannenberg 1997) is a Lisp-based synthesis environment that extends the XLisp interpreter with data types and operators specific to signal processing. The main novelty here is to treat signals as value types, which enable user programs to inspect, modify, and pass around audio signals in their entirety without significant performance penalties. Composition of signals rather than ugens allows for a wider range of constructs, especially regarding composition in time.

As for DSP, Nyquist remains close to the ugen interpreter model. Signals are lazily evaluated, and buffered to improve efficiency; the concerns here are identical to those discussed in the section "The Constraints of the Ugen Interpreter." The core processing routines are in fact written in the C language. Nyquist presents an alternative, arguably more apposite, model of interacting with the signal flow, but the programmer is still constrained to merely composing relatively monolithic routines written in C.

As a significant implementation detail, Nyquist utilizes automated code generation for its operators. According to Roger Dannenberg (1997), this is to avoid errors in the formulaic but complicated infrastructure related to matching the static C code to the context of the current user program—handling mismatched channel counts, sample rates, and timings. This could be seen as a nascent form of metaprogramming: generation of low-level signal-processing primitives from a higher-level description to bypass the tedious, error-prone boilerplate code that comes with imperative signal processing. Nyquist does not, however, seem to make any attempt to generalize this capability or, indeed, to offer it to end users.

SuperCollider: Programmatic Ugen Graphs and Parameterization

SuperCollider proposes a two-layer design, offering the user a ugen interpreter system running as a server process and a control scripting environment designed for musical programming and building the ugen graphs. The graphs themselves are interpreted. SuperCollider offers the option of processing the graph per sample, but performs poorly in this configuration.

An interesting idea in SuperCollider is a form of *ugen parameterization*: channel expansion. Vectors of parameters can be applied to ugens, which then become vectorized. This programming technique is effectively functional polymorphism: The behavior of the ugen is governed by the type of data fed into it.

The key benefit is that variants of a user-defined signal path can easily be constructed ad hoc. The programmer does not need to go through the entire ugen pipeline and adjust it in multiple places to accommodate a new channel count. The system can infer some pipeline properties from the type of input signal; the pipeline is parameterized by the input type—namely, channel count. This aids in reusing an existing design in new contexts.

PWGLSynth and ChucK: Finegrained Interpretation

PWGLSynth (Laurson and Norilo 2006) and ChucK (Wang and Cook 2003) are implemented as ugen interpreters, but they operate on a per-sample basis. In PWGLSynth, this design choice results from the desire to support a variety of physics-based models, in which unit-delay recursion and precise signal timings are required. ChucK also requires a high time resolution, as it is based on the premise of interleaving the processing of a high-level control script and a conventional ugen graph with accurate timing. Such a design takes a severe performance hit from the fine-grained interpretation, but does not prevent these systems from supporting a wide range of synthesis and analysis tasks in real time.

Both environments feature a synchronous audio graph with pull semantics, offering special constructs for asynchronous push semantics, considered useful for audio analysis (Norilo and Laurson 2008b; Wang, Fiebrink, and Cook 2007). PWGLSynth is best known for its close integration to the PWGL system, including the latter's music-notation facilities. ChucK's main contribution is to enhance the ugen graph paradigm with an imperative control script, with the capability to accurately time its interventions.

Extempore/XTLang: Dynamic Code Generation

Andrew Sorensen's Extempore has recently gained signal-processing capabilities in the form of XTLang (Sorensen and Gardner 2010). XTLang is a systems-programming extension to Lisp, offering a thin wrapper over the machine model presented by LLVM (Lattner and Adve 2004) along with a framework for region-based memory management. XTLang is designed as a low-level, high-performance language, and in many cases it requires manual data-type annotations and memory management. The design of Extempore/XTLang is notable in pursuing a high degree of integration between a slower, dynamic, high-level idiom, and an efficient low-level machine representation. In effect, the higher-level language can drive the XTLang compiler, generating and compiling code on demand.

Faust: Rethinking the Fundamentals

An important example of a language designed for and capable of implementing ugens is Faust (Orlarey, Fober, and Letz 2009). Faust utilizes functional dataflow programming to enable relatively high-level description and composition of signal-processing fundamentals.

The core principle behind Faust is the composition of signal-processing functions: "block diagrams," in the Faust vernacular. Primitives can be combined in several elementary routings, including parallel, serial, and cyclic signal paths. This programming model discards the imperative style in favor of a declarative description of signal paths, allowing eloquent and compact representation of many signal-processing algorithms.

Most importantly, Faust can compose functions on the very lowest level, with sample granularity and no dispatch overhead. This is possible because Faust performs whole-program compilation, using C as the intermediate representation of a static signal-flow graph. A custom compiler is a significant technical achievement, allowing Faust to overcome the limits of interpreters as discussed previously.

Table 1. Some Musical Programming Environments

Environment	*Scheme*	*Per Sample*	*Features*
CLM	Transcompiler	x	Low-level DSP in Lisp
Nyquist	Interpreter		Signals as values
SuperCollider	Interpreter	*See note*	Ugen graph generation
PWGLSynth	Interpreter	x	Score integration
ChucK	Interpreter	x	Strong timing
Extempore	Interpreter/Compiler	x	Dynamic code generation
Faust	Compiler	x	High-level DSP

Note: SuperCollider can use very short buffers; in practice, however, this can become prohibitive in terms of performance.

Summary of Surveyed Environments

The environments surveyed are summarized in Table 1. Common Lisp Music, PWGLSynth, ChucK, Extempore, and Faust are capable of operating per sample, making them viable candidates for the fundamentals of signal processing. For SuperCollider, this capability exists in theory, but is not useful in practice owing to low performance.

Extempore and CLM in effect wrap a C-like stack-machine representation in an S-expression syntax. The programming models do not differ significantly from programming in pure C, and in many cases are lower level than standard C++. Although Extempore is interesting in the sense that signal processors can be conveniently and quickly created from Lisp, it does not seem to be designed to tackle the core issue of signal processing in a new way.

PWGLSynth and ChucK offer a ugen graph representation capable of operating on the sample level. As interpreters, these systems fall far below theoretical machine limits in computational performance. The desire to achieve adequate performance has likely governed the core design of these environments—both feature large, monolithic ugens that can only be composed in very basic ways.

Faust is the project that is most closely aligned with the goals of the present study. Discarding the ugen model entirely, it is a signal-processing language designed from the ground up for the task of high-level representation of common DSPs with a very high performance.

Towards Higher-Level Signal Processing

As is evident from the survey in the section "Beyond Ugens," solutions and formulations that address musical programming on a higher level—those that constitute the Ousterhoutian "glue"—are plentiful. Their lower-level counterparts are far fewer. Only Faust is competitive with C/C++, if one desires to design filters, oscillators, or physical models from scratch. The objective of the present study is to explore and develop this particular domain.

A Look at Faust

Programming in Faust is about the composition of block diagrams. At the leaves of its syntax tree are functions, such as sin, cos, or 5 (interpreted as a constant-valued function). These can be composed using one of the five operators merge, split, sequential, parallel, or recursive composition. In terms of a signal graph, the leaves of the Faust syntax tree are the nodes, and the composition operators describe the edges. The Faust syntax tree is therefore topologically quite far removed from the actual signal-flow graph. The programs are compact, to the point of being terse. An example of a biquad filter implemented in Faust is shown in Figure 1; this is an excerpt from the Faust tutorial.

Faust is a pure functional language (Strachey 2000). Programs have no state, yet Faust is capable of implementing algorithms that are typically stateful, such as digital filters and delay effects.

Figure 1. Biquad filter in Faust.

```
biquad(a1,a2,b0,b1,b2) = + ~ conv2(a1,a2) : conv3(b0,b1,b2)
with {
 conv3(k0,k1,k2,x) = k0*x + k1*x' + k2*x'' ;
 conv2(k0,k1,x) = k0*x + k1*x' ;
};
```

This is accomplished by lifting signal memory to a language construct. Faust offers delay operators, in addition to the recursive composition operator that introduces an implicit unit delay. By utilizing these operators, Faust functions are pure functions of current and past inputs.

Abstraction at a more sophisticated level has been available since Faust was enhanced with a term-rewriting extension by Albert Gräf (2010). Faust functions can now change their behavior based on pattern matching against the argument. As the arguments are block diagrams, this is a form of functional polymorphism with regard to the topology of signal graphs.

In summary, Faust defines a block-diagram algebra, which is used to compose an audio-processing function of arbitrary complexity. This function describes a static signal flow graph, which can be compiled into efficient C++ code.

Introducing Kronos

This section provides an overview of the Kronos programming language, which is the focus of the present study.

Designing for Code Optimization

Ideally, specification of a programming language should be separated from its implementation, delegating all concerns of time and space efficiency to the compiler. In practice, this is not always the case. In the section "The Constraints of the Ugen Interpreter," I proposed that concerns with implementation efficiency encourage ugen design that is detrimental to the language. A more widely acknowledged example is the case of tail-call optimization that many functional languages, such as Scheme (Abelson et al. 1991), require. The idiomatic programming style in Scheme relies on the fact that the compiler can produce tail-recursive functions that operate in constant space.

Because signal processing is a very narrow, focused programming task, design for optimization can be more radical.

The first assumption I propose is that multirate techniques are essential for optimizing the efficiency of signal processors. Most systems feature a distinction between audio rate and control rate. I propose that update rate should be considered to be a task for the optimizing compiler. It should be possible to use similar signal semantics for all the signals in the system, from audio to control to MIDI and the user interface, to enable an universal signal model (Norilo and Laurson 2008a).

The second assumption is that for signal processing we often desire a higher level of expressivity and abstraction when describing the signal-processor topology than during processing itself. This assumption is supported by the fact that environments like Faust, ChucK, and SuperCollider, among others, divide the task of describing signal processors into graph generation and actual processing. I propose that this division be considered at the earliest stages of language design, appropriately formalized, and subsequently exploited in optimization.

The Dataflow Language

The starting point of the proposed design is a dataflow language that is minimal in the sense of being very amenable to compiler optimization, but still complete enough to represent the majority of typical signal-processing tasks. For the building blocks, we choose arithmetic and logic on elementary data types, function application, and algebraic type composition. The language will be represented by a static signal graph, implying determinism that is useful for analysis and optimization of the equivalent machine code.

The nodes of the graph represent operators, and the edges represent signal transfer. The graph is functionally pure, which means that functions cannot induce observable side effects. As in Faust, delay and signal memory is included in the dataflow language as a first-class operator. The compiler reifies the delays as stateful operations. This restricted use of state allows the language to be referentially transparent (Strachey 2000) while providing efficient delay operations: the user faces pure functions of current and past inputs, while the machine executes a streamlined ring-buffer operation.

The language semantics are completed by allowing cycles in the signal flow, as long as each cycle includes at least one sample of delay. This greatly enhances the capability of the dataflow language to express signal processors, as feedback-delay routing is extremely commonplace. If the Kronos language is represented textually, cyclic expressions result from symbols defined recursively in terms of each other; in visual form, the cycles are directly observable in the program patch.

The deterministic execution semantics and referential transparency allow the compiler to perform global dataflow analysis on entire programs. The main use of this facility is automated factorization of signal rates: The compiler can determine the required update rates of each pure function in the system by observing the update rates of its inputs. Signal sources can be inserted into the dataflow graph as external inputs. In compilation, these become the entry points that drive the graph computation. One such entry point is the audio clock; another could represent a slider in the user interface. This factorization is one of the main contributions of the Kronos project.

The dataflow principle behind what is described here is classified by Peter Van Roy (2009) as a discrete reactive system. It will respond to a well-defined series of discrete input events with another well-defined series of discrete output events, which is true for any Kronos program or fragment of one.

The Metalanguage

As the reader may observe, the dataflow language as described here greatly resembles the result of a function composition written in Faust. Such a static signal graph is no doubt suitable for optimizing compilers; however, it is not very practical for a human programmer to write directly. As an abstraction over the static signal graph, Faust offers a block-diagram algebra and term rewriting (Gräf 2010).

For Kronos, I propose an alternative that I argue is both simpler and more comprehensive. Instead of the dataflow language, the programmer works in a *metalanguage*. The main abstraction offered by the metalanguage is polymorphic function application, implemented as System F_ω (Barendregt 1991): The behavior and result type of a function are a function of argument type, notably, argument value is not permitted to influence the result type. The application of polymorphic functions is guided by *type constraints*—the algebraic structure and semantic notation of signals can be used to drive function selection. This notion is very abstract; to better explain it, in the section “Case Studies” I show how it can encode a block-diagram algebra, algorithmic routing, and techniques of generic programming.

In essence, the metalanguage operates on types and the dataflow language operates on values. The metalanguage is used to construct a statically typed dataflow graph. The restrictions of System F_ω ensure that the complexity of functional polymorphism can be completely eliminated when moving from the metalanguage to the dataflow language; any such complexity is in the type-system computations at compile time—it is essentially free during the critical real-time processing of data flow. This enables the programmer to fully exploit very complicated polymorphic abstractions in signal processing, with a performance similar to low-level C, albeit with considerable restrictions.

Because values do not influence types, dependent types cannot be expressed. As type-based polymorphism is the main control-flow mechanism, runtime values are, in effect, shut out from influencing program flow.

Table 2. Kronos Language Features

Paradigm	Functional
Evaluation	Strict
Typing discipline	Static, strong, derived
Compilation	Static, just in time
Usage	Library, repl, command line
Backend	LLVM (Lattner and Adve 2004)
Platforms	Windows, Mac OSX, Linux
License	GPL3
Repository	https://bitbucket.org/vnorilo/k3

This restriction may seem crippling to a programmer experienced in general-purpose languages, although a static signal graph is a feature in many successful signal-processing systems, including Pure Data and Faust.

I argue that the System F_ω is an apposite formalization for the division of signal processing into graph generation and processing. It cleanly separates the high-level metaprogramming layer and low-level signal-processing layer into two distinct realms: that of types, and that of values.

A summary of the characteristics of the Kronos language is shown in Table 2. For a detailed discussion of the theory, the reader is referred to prior publications (Norilo 2011b, 2013).

Case Studies

This section aims to demonstrate the programming model described in the earlier section "Towards Higher-Level Signal Processing," via case studies selected for each particular aspect of the design. The examples are not designed to be revolutionary; rather, they are selected as a range of representative classic problems in signal processing. I wish to stress that the examples are designed to be self-contained. Although any sustainable programming practice relies on reusable components, the examples here strive to demonstrate how proper signal-processing modules can be devised relatively easily from extremely low-level primitives, without an extensive support library, as long as the language provides adequate facilities for abstraction.

Polymorphism, State, and Cyclic Graphs

A simple one-pole filter implemented in Kronos is shown in Figure 2. This example demonstrates elementary type computations, delay reification, and cyclic signal paths. The notable details occur in the unit-delay operator `z-1`. This operator receives two parameters, a forward initialization path and the actual signal path.

Notably, the signal path in this example is cyclic. This is evident in how the definitions of `y0` and `y1` are mutually recursive. Please note that `y1` is not a variable or a memory location: It is a symbol bound to a specific node in the signal graph. Kronos permits cyclic graphs, as long as they feature a delay operator. The mapping of this cycle to efficient machine code is the responsibility of the dataflow compiler, which produces a set of assignment side effects that fulfill the desired semantics.

The forward initialization path is used to describe the implicit history of the delay operator before any input has been received. Instead of being expressed directly as a numerical constant, the value is derived from the `pole` parameter. This causes the data type of the delay path to match that of the `pole` parameter. If the user chooses to utilize double precision for the pole parameter, the internal data paths of the filter are automatically instantiated in double precision. The input might still be in single precision; by default, the runtime library would inject a type upgrade into the difference equation. The upgrade semantics are based on functional polymorphism, and they are defined in source form instead of being built into the compiler.

User-defined types can also be used for the `pole` parameter, provided that suitable multiplication and subtraction operators and implicit type conversions exist. The runtime library provides a complex number implementation (again, in source form) that provides basic arithmetic and specifies an implicit type coercion from real values: if used for the pole, the filter becomes a complex resonator with complex-valued output. This type-based polymorphism can be seen as a generalization of ugen parameterization—for example, the way SuperCollider ugens can adapt to incoming channel counts.

Figure 2. One-pole filter.

```
Filter(x0 pole) {
    y1 = z-1( (pole - pole) y0 )
    ; y1 is initially zero, subsequently delayed y0.
    ; The initial value of zero is expressed as 'pole - pole' to ensure
    ; that the feedback path type matches the pole type.

    y0 = x0 - pole * y1
    ; Compute y0, the output.

    Filter = y0
    ; This is the filter output.
}

; Straightforward single-precision one-pole filter:
;   example1 = Filter(sig 0.5)
; Upgrade the signal path to double precision:
;   example2 = Filter(sig 0.5d)

; Use as a resonator via a complex pole and reduction to real part.
Resonator(sig w radius) {
    ; 'Real' and 'Polar' are functions in namespace 'Complex'
    Resonator = Complex:Real(Filter(sig Complex:Polar(w radius)))
}
```

Figure 2

For a more straightforward implementation, the complicated type computations can be ignored. Declaring the unit delay as `y1 = z-1(0 y0)` would yield a filter that was fixed to single-precision floating point; different types for the input signal or the pole would result in a type error at the `z-1` operator. This approach is likely more suitable for beginning programmers, although they should have little difficulty in using (as opposed to coding) the generic version.

An implementation of a biquad filter is shown in Figure 3. This filter is identical to the Faust version in Figure 1. Because Kronos operates on signal values instead of block diagrams, the syntax tree of this implementation is identical to the signal flow graph. This is arguably easier to understand than the Faust version.

Higher-Order Functions in Signal Processing

Polymorphism is a means of describing something more general than a particular filter implementation. For instance, the previous example described the principle of unit-delay recursion through a feedback coefficient with different abstract types.

An even more fundamental principle underlies this filter model and several other audio processes: That of recursive composition of unit delays. This can be described in terms of a binary function of the feedforward and feedback signals into an output signal.

Figure 3. Biquad filter, based on the realization known as Direct Form II (Smith 2007).

```
Biquad(sig b0 b1 b2 a1 a2) {
    zero = sig - sig

    ; feedback section
    y0 = sig - y1 * a1 - y2 * a2
    y1 = z-1(zero y0)
    y2 = z-1(zero y1)

    ; feedforward section
    Biquad = y0 * b0 + y1 * b1 + y2 * b2
}
```

Figure 3

Figure 4. Generic recursion.

```
; Routes a function output back to its first argument through a unit delay.
Recursive(sig binary-func) {
    state = binary-func(z-1(sig - sig state) sig)
    Recursive = state
}

Filter2(sig pole) {
    ; Lambda arrow '=>' constructs an anonymous function: the arguments
    ; are on the left hand side, and the body is on the right hand side.
    dif-eq = (y1 x0) => x0 - pole * y1
    ; onepole filter is a recursive composition of a simple multiply-add expression.
    Filter2 = Recursive( sig dif-eq )
}

Buzzer(freq) {
    ; Local function to wrap the phasor.
    wrap = x => x - Floor(x)
    ; Compose a buzzer from a recursively composed increment wrap.
    Buzzer = Recursive( freq (state freq) =>
        wrap(state + Frequency-Coefficient(freq Audio:Signal(0))) )
}

; example usage
;Filter2(Buzzer(440) 0.5)
```

Recursive Routing Metafunction

In Kronos, the presence of first-class functions—or functions as signals—allows for higher-order functions. Such a function can be designed to wrap a suitable binary function in a recursive composition as previously described. The implementation of this metafunction is given in Figure 4, along with example usage to reconstruct the filter from Figure 2 as well as a simple phasor, used here as a naive sawtooth oscillator. This demonstrates how to implement a composition operator, such as those built into Faust, by utilizing higher-order functions.

The recursive composition function is an example of algorithmic routing. It is a function that generates signal graphs according to a generally useful routing principle. In addition, parallel and serial routings are ubiquitous, and well suited for expression in the functional style.

Schroeder Reverberator

Schroeder reverberation is a classic example of a signal-processing problem combining parallel and serial routing (Schroeder 1969). An example implementation is given in Figure 5 along with routing metafunctions, `Map` and `Fold`. Complete implementations are shown for demonstration purposes—the functions are included in source form within the runtime library.

Further examples of advanced reverberators written in Kronos can be found in an earlier paper by the author (Norilo 2011a).

Sinusoid Waveshaper

Metaprogramming can be applied to implement reconfigurable signal processors. Consider a polynomial sinusoid waveshaper; different levels of precision are required for different applications. Figure 6 demonstrates a routine that can generate a polynomial of any order in the type system.

In summary, the functional paradigm enables abstraction and generalization of various signal-processing principles such as the routing algorithms described earlier. The application of first-class functions allows flexible program composition at compile time without a negative impact on runtime performance.

Figure 5. Algorithmic routing.

```
Fold(func data) {
    ; Extract two elements and the tail from the list.
    (x1 x2 xs) = data
    ; If tail is empty, result is 'func(x1 x2)'
    ; otherwise fold 'x1' and 'x2' into a new list head and recursively call function.
    Fold = Nil?(xs) : func(x1 x2)
           Fold(func func(x1 x2) xs)
}

; Parallel routing is a functional map.
Map(func data) {
    ; For an empty list, return an empty list.
    Map = When(Nil?(data) data)
    ; Otherwise split the list to head and tail,
    (x xs) = data
    ; apply mapping function to head, and recursively call function.
    Map = (func(x) Map(func xs))
}

; Simple comb filter.
Comb(sig feedback delay) {
    out = rbuf(sig - sig  delay sig + feedback * out)
    Comb = out
}

; Allpass comb filter.
Allpass-Comb(sig feedback delay) {
    vd = rbuf(sig - sig delay v)
    v = sig - feedback * vd
    Allpass-Comb = feedback * v + vd
}

Reverb(sig rt60) {
    ; List of comb filter delay times for 44.1 kHz.
    delays = [ #1687 #1601 #2053 #2251 ]
    ; Compute rt60 in samples.
    rt60smp = Rate-of( sig ) * rt60
    ; A comb filter with the feedback coefficient derived from delay time.
    rvcomb = delay => Comb(sig Math:Pow( 0.001  delay / rt60smp ) delay)
    ; Comb filter bank and sum from the list of delay times.
    combs-sum = Fold( (+) Map( rvcomb delays ) )
    ; Cascaded allpass filters as a fold.
    Reverb = Fold( Allpass-Comb [combs-sum (0.7 #347) (0.7 #113) (0.7 #41)] )
}
```

Multirate Processing: FFT

Fast Fourier transform (FFT)–based spectral analysis is a good example of a multirate process. The signal is transformed from an audio-rate sample stream to a much slower and wider stream of spectrum frames. Such buffered processes can be expressed as signal-rate decimation on the contents of ring

Figure 6. Sinusoid waveshaper.

```
Horner-Scheme(x coefficients) {
    Horner-Scheme = Fold((a b) => a + x * b coefficients)
}

Pi = #3.14159265359

Cosine-Coefs(order) {
    ; Generate next exp(x) coefficient from the previous one.
    exp-iter = (index num denom) => (
        index + #1          ; next coefficient index
        num * #2 * Pi       ; next numerator
        denom * index)      ; next denominator
    flip-sign = (index num denom) => (index Neg(num) denom)
    ; Generate next cos(pi w) coefficient from the previous one.
    sine-iter = x => flip-sign(exp-iter(exp-iter(x)))
    ; Generate 'order' coefficients.
    Cosine-Coefs = Algorithm:Map(
            (index num denom) => (num / denom)
            Algorithm:Expand(order sine-iter (#2 #-2 * Pi #1)))
}

Cosine-Shape(x order) {
    x1 = x - #0.25
    Cosine-Shape = x1 * Horner-Scheme(x1 * x1 Cosine-Coefs(order))
}
```

buffers, with subsequent transformations. Figure 7 demonstrates a spectral analyzer written in Kronos. For simplicity, algorithmic optimization for real-valued signals has been omitted. The FFT, despite the high-level expression, performs similarly to a simple nonrecursive C implementation. It cannot compete with state-of-the-art FFTs, however.

Because the result of the analyzer is a signal consisting of FFT frames at a fraction of the audio rate, the construction of algorithms such as overlap-add convolution or FFT filtering is easy to accomplish.

Polyphonic Synthesizer

The final example is a simple polyphonic FM synthesizer equipped with a voice allocator, shown in Figure 8. This is intended as a demonstration of how the signal model and programming paradigm can scale from efficient low-level implementations upwards to higher-level tasks.

The voice allocator is modeled as a ugen receiving a stream of MIDI data and producing a vector of voices, in which each voice is represented by a MIDI note number, a gate signal, and a "voice age" counter. The allocator is a unit-delay recursion around the vector of voices, utilizing combinatory logic to lower the gate signals for any released keys and insert newly pressed keys in place of the least important of the current voices. The allocator is driven by the MIDI signal, so each incoming MIDI event causes the voice vector to update. This functionality depends on the compiler to deduce data flows and provide unit-delay recursion on the MIDI stream.

To demonstrate the multirate capabilities of Kronos, the example features a low-frequency oscillator (LFO) shared by all the voices. This LFO is just another FM operator, but its update rate is downsampled by a factor of `krate`. The LFO modulates the frequencies logarithmically. This is contrived, but should demonstrate the effect of compiler optimization of update rates, since an expensive power function is required for each frequency computation. Table 3 displays three

Figure 7. Spectrum analyzer.

```
Stride-2(Xs) {
    ; Remove all elements of Xs with odd indices.
    Stride-2 = []
    Stride-2 = When(Nil?(Rest(Xs)) [First(Xs)])
    (x1 x2 xs) = Xs
    Stride-2 = (x1 Recur(xs))
}

Cooley-Tukey(dir Xs) {
    Use Algorithm
    N    = Arity(Xs)                ; FFT size
    sub  = 'Cooley-Tukey(dir _)
    even = sub(Stride-2(Xs))        ; compute even sub-FFT
    odd  = sub(Stride-2(Rest(Xs)))  ; compute odd sub-FFT

    ; Compute the twiddle factor for radix-2 FFT.
    twiddle-factor = Complex:Polar((dir * Math:Pi / N) * #2 #1) * 1
    ; Apply twiddle factor to the odd sub-FFT.
    twiddled = Zip-With(Mul odd Expand(N / #2 (* twiddle-factor) Complex:Cons(1 0)))

    (x1 x2 _) = Xs

    Cooley-Tukey =
        N  < #1 : Raise("Cooley-Tukey FFT requires a power-of-two array input")
        N == #1 : [First(Xs)] ; terminate FFT recursion
        ; Recursively call function and recombine sub-FFT results.
            Concat(
                Zip-With(Add even twiddled)
                Zip-With(Sub even twiddled))
}

Analyzer(sig N overlap) {
    ; Gather 'N' frames in a buffer.
    (buf i out) = rcsbuf(0 N sig)
    ; Reduce sample rate of 'buf' by factor of (N / overlap) relative to 'sig'.
    frame = Reactive:Downsample(buf N / overlap)
    ; Compute forward FFT on each analysis frame.
    Analyzer = Cooley-Tukey(#1 frame)
}
```

benchmarks of the example listing with different control rate settings on an Intel Core i7-4500U at 2.4GHz. With control rate equaling audio rate, the synthesizer is twice as expensive to compute as with a control rate set to 8. The benefit of lowering the control rate becomes marginal after about 32. This demonstrates the ability of the compiler to deduce data flows and eliminate redundant computation—note that the only change was to the downsampling factor of the LFO.

Discussion

In this section, I discuss Kronos in relation to prior work and initial user reception. Potential future work is also identified.

Figure 8. Polyphonic synthesizer (continued on next page).

```
Package Polyphonic {
    Prioritize-Held-Notes(midi-bytes voices) {
        choose = Control-Logic:Choose
        (status note-number velocity) = midi-bytes
        ; Kill note number if event is note off or note on with zero velocity.
        kill-key = choose(status == 0x80 | (status == 0x90 & velocity == 0i)
                          note-number -1i)
        ; New note number if event is note on and has nonzero velocity.
        is-note-on = (status == 0x90 & velocity > 0i)
        ; A constant specifying highest possible priority value.
        max-priority = 2147483647i
        ; Lower gate and reduce priority for released voice.
        with-noteoff = Map((p k v) => (p - (max-priority & (k == kill-key))
                            k
                            v & (k != kill-key))
                            voices)
        ; Find oldest voice by selecting lowest priority.
        lowest-priority = Fold(Min Map(First voices))
        ; Insert new note.
        Prioritize-Held-Notes =
            Map((p k v) => choose((p == lowest-priority) & is-note-on
                                  (max-priority note-number velocity)
                                  (p - 1i k v))
                with-noteoff)
    }

    Allocator(num-voices allocator midi-bytes) {
            ; Create initial voice allocation with running priorities so that the allocator
            ; always sees exactly one voice as the oldest voice.
            voice-init = Algorithm:Expand(num-voices (p _ _) => (p - 1i 0i 0i) (0i 0i 0i))
            ; Generate and clock the voice allocator loop from the MIDI stream.
            old-voices = z-1(voice-init Reactive:Resample(new-voices midi-bytes))
            ; Perform voice allocation whenever the MIDI stream ticks.
            new-voices = allocator(midi-bytes old-voices)
            Allocator = new-voices
    }
}
```

Kronos and Faust

Among existing programming environments, Faust is, in principle, closest to Kronos. The environments share the functional approach. Faust has novel block-composition operands that are powerful but perhaps a little foreign syntactically to many users. Kronos emphasizes high-level semantic metaprogramming for block composition.

Kronos programs deal with signal values, whereas Faust programs deal with block diagrams. The former have syntax trees that correspond one-to-one with the signal flow, and the latter are topologically very different. I argue that the correspondence is an advantage, especially if a visual patching environment is used (Norilo 2012). If desired, the Kronos syntax can encode block-diagram algebra with higher-order functions, down to custom infix operators. Faust can also encode signal-flow topology by utilizing term rewriting (Gräf 2010), but only in the feedforward case.

Figure 8. Polyphonic synthesizer (continued from previous page).

```
; —— Synthesizer ——————————————————————————————————————————————
FM-Op(freq amp) {
    ; apply sinusoid waveshaping to a sawtooth buzzer
    FM-Op = amp * Approx:Cosine-Shape(Abs(Buzzer(freq) - 0.5) #5)
}

FM-Voice(freq gate) {
    ; attack and decay slew per sample
    (slew+ slew-) = (0.003 -0.0001)
    ; upsample gate to audio rate
    gate-sig = Audio:Signal(gate)
    ; slew limiter as a recursive composition over clipping the value differential
    env = Recursive( gate-sig (old new) => old + Max(slew- Min(slew+ new - old)) )
    ; FM modulator osc
    mod = FM-Op(freq freq * 8 * env)
    ; FM carrier osc
    FM-Voice = FM-Op(freq + mod env)
}

; —— Test bench ———————————————————————————————————————————————
Synth(midi-bytes polyphony krate) {
    ; transform MIDI stream into a bank of voices
    voices = Polyphonic:Allocator( polyphony
        Polyphonic:Prioritize-Held-Notes
        midi-bytes )
    lfo = Reactive:Downsample(FM-Op(5.5 1) krate)
    ; make a simple synth from the voice vector
    Synth = Fold((+)
              Map((age key gate) => FM-Voice(
                440 * Math:Pow(2 (key - 69 +  lfo * gate / 256) / 12) ; freq
                gate / 128) ; amp
              voices))
}
```

Kronos is designed as a System F_ω compiler, complete with a multirate scheme capable of handling event streams as well. The multirate system in Faust (Jouvelot and Orlarey 2011) is a recent addition and less general, supporting "slow" signals that are evaluated once per block, audio signals, and, more recently, up- and downsampled versions of audio signals. The notion of an event stream does not exist as of this writing.

The strengths of Faust include the variety of supported architectures (Fober, Orlarey, and Letz 2011), generation of block-diagram graphics, symbolic computation, and mathematical documentation. The compiler has also been hardened with major projects, such as a port of the Synthesis Toolkit (Michon and Smith 2011).

Kronos and Imperative Programming

Poing Imperatif by Kjetil Matheussen (2011) is a source-to-source compiler that is able to lower object-oriented constructs into the Faust language. Matheusen's work can be seen as a study of isomorphisms between imperative programming and Faust. Many of his findings apply directly to Kronos as well. Programs in both languages have state, but it is provided as an abstract language construct

Table 3. Impact of Update Rate Optimization

krate	*μsec per 1,024 Samples*
1	257
2	190
8	127
32	118
128	114

and reified by the compiler. Poing Imperatif lowers mutable fields in objects to feedback-delay loops—constructs that represent such abstract state. Essentially, a tick of a unit-delay signal loop is equivalent to a procedural routine that reads and writes an atom of program state. The key difference from a general-purpose language is that Kronos and Faust enforce locality of state—side effects cannot be delegated to subroutines. Matheusen presents a partial workaround: Subroutines describe side effects rather than perform them, leaving the final mutation to the scope of the state.

The reactive capabilities of Kronos present a new aspect in the comparison with object-oriented programming. Each external input to a Kronos program has a respective update routine that modifies some of the program state. The inputs are therefore analogous to object methods. A typical object-oriented implementation of an audio filter would likely include methods to set the high-level design parameters such as corner frequency and Q, and a method for audio processing. The design-parameter interface would update coefficients that are internal to the filter, which the audio process then uses. Kronos generates machine code that looks very similar to this design. The implicitly generated memory for the signal-clock boundaries contains the coefficients: intermediate results of the signal path that depend only on the design parameters.

At the source level, the object-oriented program spells out the methods, signal caches, and delay buffers. Kronos programs declare the signal flow, leaving method factorization and buffer allocation to the compiler. This is the gist of the tradeoff offered: Kronos semantics are more narrowly defined, allowing the programmer to concentrate exclusively on signal flow. This is useful when the semantic model suits the task at hand; but if it does not, the language is not as flexible as a general-purpose one.

User Evaluation

I have been teaching the Kronos system for two year-long courses at the University of Arts Helsinki, as well as intensive periods in the Conservatory of Cosenza and the National University of Ireland, Maynooth. In addition, I have collected feedback from experts at international conferences and colloquia, for example, at the Institut de Recherche et de Coordination Acoustique/Musique (IRCAM) in Paris and the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University.

Student Reception

The main content of my Kronos teaching consists of using the visual patcher and just-in-time compiler in building models of analog devices, in the design of digital instruments, and in introducing concepts of functional programming. The students are majors in subjects such as recording arts or electronic music. Students generally respond well to filter implementation, as the patches correspond very closely to textbook diagrams. They respond well to the idea of algorithmic routing, many expressing frustration that it is not more widely available, but they struggle to apply it by themselves. Many are helped by terminology from modular synthesizers, such as calling `Map` a bank and `Reduce` a cascade. During the longer courses, students have implemented projects, such as AudioUnit plug-ins and mobile sound-synthesis applications.

Expert Reception

Among the expert audience, Kronos has attracted the most positive response from engineers and signal-processing researchers. Composers seem to be less interested in the problem domain it tackles. Many experts have considered the Kronos syntax to be easy to follow and intuitive, and its compilation speed and performance to be excellent. A common doubt is with regard to the capability of a static dataflow graph to express an adequate number of algorithms. Adaptation of various algorithms to the model is indeed an ongoing research effort.

Recently, a significant synergy was discovered between the Kronos dataflow language and the WaveCore, a multicore DSP chip designed by Verstraelen, Kuper, and Smit (2014). The dataflow language closely matches the declarative WaveCore language, and a collaborative effort is ongoing to develop Kronos as a high-level language for the WaveCore chip.

Current State

Source code and release files for the Kronos compiler are available at https://bitbucket.org/vnorilo/k3. The code has been tested on Windows 8, Mac OS X 10.9, and Ubuntu Linux 14, for which precompiled binaries are available. The repository includes the code examples shown in this article. Both the compiler and the runtime library are publicly licensed under the GNU General Public License, Version 3.

The status of the compiler is experimental. The correctness of the compiler is under ongoing verification and improvement by means of a test suite that exercises a growing subset of possible use cases. The examples presented in this article are a part of the test suite.

Future Work

Finally, I discuss the potential for future research. The visual front end is especially interesting in the context of teaching and learning signal processing, and core language enhancements could further extend the range of musical programming tasks Kronos is able to solve well.

Visual Programming and Learnability

Kronos is designed from the ground up to be adaptable to visual programming. In addition to the core technology, supporting tools must be built to truly enable it. The current patcher prototype includes some novel ideas for making textual and visual programming equally powerful (Norilo 2012).

Instantaneous visual feedback in program debugging, inspection of signal flow, and instrumentation are areas where interesting research could be carried out. Such facilities would enhance the system's suitability for pedagogical use.

Core Language Enhancements

Type determinism (as per System F_ω) and early binding are key to efficient processing in Kronos. It is acknowledged, however, that they form a severe restriction on the expressive capability of the dataflow language.

Csound is a well-known example of an environment where notes in a score and instances of signal processors correspond. For each note, a signal processor is instantiated for the required duration. This model cleanly associates the musical object with the program object.

Such a model is not immediately available in Kronos. The native idiom for dynamic polyphony would be to generate a signal graph for the maximum number of voices and utilize a dynamic clock to shut down voices to save processing time. This is not as neat as the dynamic allocation model, because it forces the user to specify a maximum polyphony.

More generally, approaches to time-variant processes on the level of the musical score are interesting; works such as Eli Brandt's (2002) Chronic offer ideas on how to integrate time variance and the paradigm of functional programming. Dynamic mutation could be introduced into the dataflow graph by utilizing techniques from class-based polymorphic languages, such as type erasure on closures.

In its current state, Kronos does not aim to replace high-level composition systems such as Csound or Nyquist (Dannenberg 1997). It aims to implement the bottom of the signal-processing stack well, and thus could be a complement to a system operating on a higher ladder of abstraction. Both of the aforementioned systems could, for example, be extended to drive the Kronos just-in-time compiler for their signal-processing needs.

Conclusion

This article has presented Kronos, a language and a compiler suite designed for musical signal processing. Its design criteria are informed by the requirements of real-time signal processing fused with a representation on a high conceptual level.

Some novel design decisions enabled by the DSP focus are whole-program type derivation and compile-time computation. These features aim to offer a simple, learnable syntax while providing extremely high performance. In addition, the ideas of ugen parameterization and block-diagram algebra were generalized and described in the terms of types in the System F_ω. Abstract representation of state via signal delays and recursion bridges the gap between pure functions and stateful ugens.

All signals are represented by a universal signal model. The system allows the user to treat events, control, and audio signals with unified semantics, with the compiler providing update-rate optimizations. The resulting machine code closely resembles that produced by typical object-oriented strategies for lower-level languages, while offering a very high-level dataflow-based programming model on the source level. As such, the work can be seen as a study of formalizing a certain set of programming practices for real-time signal-processing code, and providing a higher-level abstraction that conforms to them. The resulting source code representation is significantly more compact and focused on the essential signal flow—provided that the problem at hand can be adapted to the paradigm.

References

Abelson, H., et al. 1991. “Revised Report on the Algorithmic Language Scheme.” *ACM SIGPLAN Lisp Pointers* 4(3):1–55.

Barendregt, H. 1991. “Introduction to Generalized Type Systems.” *Journal of Functional Programming* 1(2):124–154.

Boulanger, R. 2000. *The Csound Book*. Cambridge, Massachusetts: MIT Press.

Brandt, E. 2002. “Temporal Type Constructors for Computer Music Programming.” PhD disssertation, Carnegie Mellon University, School of Computer Science.

Dannenberg, R. B. 1997. “The Implementation of Nyquist, a Sound Synthesis Language.” *Computer Music Journal* 21(3):71–82.

Ertl, M. A., and D. Gregg. 2007. “Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters.” *ACM TOPLAS Notices* 29(6):37.

Fober, D., Y. Orlarey, and S. Letz. 2011. “Faust Architectures Design and OSC Support.” In *Proceedings of the International Conference on Digital Audio Effects*, pp. 213–216.

Gräf, A. 2010. “Term Rewriting Extensions for the Faust Programming Language.” In *Proceedings of the Linux Audio Conference*, pp. 117–122.

Jouvelot, P., and Y. Orlarey. 2011. “Dependent Vector Types for Data Structuring in Multirate Faust.” *Computer Languages, Systems and Structures* 37(3):113–131.

Kim, H., et al. 2009. “Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware.” *IEEE Transactions on Computers* 58(9):1153–1170.

Lattner, C., and V. Adve. 2004. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation.” *International Symposium on Code Generation and Optimization* 57(c):75–86.

Laurson, M., and V. Norilo. 2006. “From Score-Based Approach towards Real-Time Control in PWGLSynth.” In *Proceedings of the International Computer Music Conference*, pp. 29–32.

Matheussen, K. 2011. “Poing Impératif: Compiling Imperative and Object Oriented Code to Faust.” In *Proceedings of the Linux Audio Conference*, pp. 55–60.

McCartney, J. 2002. “Rethinking the Computer Music Language: SuperCollider.” *Computer Music Journal* 26(4):61–68.

Michon, R., and J. O. Smith. 2011. “Faust-STK: A Set of Linear and Nonlinear Physical Models for the Faust Programming Language.” In *Proceedings of the International Conference on Digital Audio Effects*, pp. 199–204.

Norilo, V. 2011a. “Designing Synthetic Reverberators in Kronos.” In *Proceedings of the International Computer Music Conference*, pp. 96–99.

Norilo, V. 2011b. “Introducing Kronos: A Novel Approach to Signal Processing Languages.” In *Proceedings of the Linux Audio Conference*, pp. 9–16.

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

Norilo, V. 2013. "Recent Developments in the Kronos Programming Language." In *Proceedings of the International Computer Music Conference*, pp. 299–304.

Norilo, V., and M. Laurson. 2008a. "A Unified Model for Audio and Control Signals in PWGLSynth." In *Proceedings of the International Computer Music Conference*, pp. 13–16.

Norilo, V., and M. Laurson. 2008b. "Audio Analysis in PWGLSynth." In *Proceedings of the International Conference on Digital Audio Effects*, pp. 47–50.

Orlarey, Y., D. Fober, and S. Letz. 2009. "Faust: An Efficient Functional Approach to DSP Programming." In G. Assayag and A. Gerszo, eds. *New Computational Paradigms for Music*. Paris: Delatour, IRCAM, pp. 65–97.

Ousterhout, J. K. 1998. "Scripting: Higher-Level Programming for the 21st Century." *Computer* 31(3):23–30.

Puckette, M. 1988. "The Patcher." In *Proceedings of International Computer Music Conference*, pp. 420–429.

Puckette, M. 1996. "Pure Data: Another Integrated Computer Music Environment." In *Proceedings of the International Computer Music Conference*, pp. 269–272.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, Massachusetts: MIT Press.

Schottstaedt, B. 1994. "Machine Tongues XVII: CLM; Music V Meets Common Lisp." *Computer Music Journal* 18:30–37.

Schroeder, M. R. 1969. "Digital Simulation of Sound Transmission in Reverberant Spaces." *Journal of the Acoustical Society of America* 45(1):303.

Smith, J. O. 2007. *Introduction to Digital Filters with Audio Applications*. Palo Alto, California: W3K.

Sorensen, A., and H. Gardner. 2010. "Programming with Time: Cyber-Physical Programming with Impromptu." In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages, and Applications*, pp. 822–834.

Strachey, C. 2000. "Fundamental Concepts in Programming Languages." *Higher-Order and Symbolic Computation* 13(1-2):11–49.

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

Verstraelen, M., J. Kuper, and G. J. M. Smit. 2014. "Declaratively Programmable Ultra-Low Latency Audio Effects Processing on FPGA." In *Proceedings of the International Conference on Digital Audio Effects*, pp. 263–270.

Wang, G., and P. R. Cook. 2003. "ChucK: A Concurrent, On-the-Fly, Audio Programming Language." In *Proceedings of the International Computer Music Conference*, pp. 1–8.

Wang, G., R. Fiebrink, and P. R. Cook. 2007. "Combining Analysis and Synthesis in the ChucK Programming Language." In *Proceedings of the International Computer Music Conference*, pp. 35–42.