Introducing Kronos
A Novel Approach to Signal Processing Languages

Vesa Norilo
Centre for Music & Technology, Sibelius Academy
Pohjoinen Rautatiekatu 9
00100 Helsinki,
Finland,
vnorilo@siba.fi

Abstract

This paper presents an overview of Kronos, a software package aimed at the development of musical signal processing solutions. The package consists of a programming language specification as well JIT Compiler aimed at generating high performance executable code.

The Kronos programming language aims to be a functional high level language. Combining this with run time performance requires some unusual tradeoffs, creating a novel set of language features and capabilities.

Case studies of several typical musical signal processors are presented and the suitability of the language for these applications is evaluated.

Keywords

Music, DSP, Just in Time Compiler, Functional, Programming language

1 Introduction

Kronos aims to be a programming language and a compiler software package ideally suited for building any custom DSP solution that might be required for musical purposes, either in the studio or on the stage. The target audience includes technologically inclined musicians as well as musically competent engineers. This prompts a re-evaluation of design criteria for a programming environment, as many musicians find industrial programming languages very hostile.

On the other hand, the easily approachable applications currently available for building musical DSP algorithms often fail to address the requirements of a programmer, not providing enough abstraction nor language constructs to facilitate painless development of more complicated systems.

Many software packages from Pure Data[Puckette, 1996] to Reaktor[Nicholl, 2008] take the approach of more or less simulating a modular synthesizer. Such packages combine a varying degree of programming language constructs into the model, yet sticking very closely to the metaphor of connecting physical modules via patch cords. This design choice allows for an environment that is readily comprehensible to anyone familiar with its physical counterpart. However, when more complicated programming is required, the apparent simplicity seems to deny the programmer the special advantages provided by digital computers.

Kronos proposes a solution more closely resembling packages like Supercollider[McCartney, 2002] and Faust[Orlarey et al., 2004], opting to draw inspiration from computer science and programming language theory. The package is fashioned as a just in time compiler[Aycock, 2003], designed to rapidly transform user algorithms into efficient machine code.

This paper presents the actual language that forms the back end on which the comprehensive DSP development environment will be built. In Section 2, *Language Design Goals*, we lay out the criteria adopted for the language design. In Section 3, *Designing the Kronos Language*, the resulting design problems are addressed. Section 5, *Case Studies*, presents several signal processing applications written in the language, presenting comparative observations of the efficacy our proposed solution to each case. Finally, Section 6, *Conclusion*, summarizes this paper and describes future avenues of research.

2 Language Design Goals

This section presents the motivation and aspirations for Kronos as a programming language. Firstly, the requirements the language should be able to fulfill are enumerated. Secondly, summarized design criteria are derived from the requirements.

2.1 Musical Solutions for Non-engineers

Since the target audience of Kronos includes non-engineers, the software should ideally be easily approached. In this regard, the visually oriented patching environments hold an advantage.

A rigorously designed language offers logical cohesion and structure that is often missing from a software package geared towards rapid visual construction of modular ad-hoc solutions. Consistent logic within the environment should ease learning.

The ideal solution should be that the environment allows the casual user to stick to the metaphor of physical interconnected devices, but also offers an avenue of more abstract programming for advanced and theoretically inclined users.

2.2 DSP Development for Professionals

Kronos also aspires to be an environment for professional DSP developers. This imposes two additional design criteria: the language should offer adequately sophisticated features, so that more powerful programming constructs can be used if desired. The resulting audio processors should also exhibit excellent real time performance.

A particularily challenging feature of a musical DSP programming is the inherent multi-rate processing. Not all signals need equally frequent updates. If leveraged, this fact can bring about dramatic performance benefits. Many systems offer a distinction between control rate and audio rate signals, but preferably this forced distinction should be eliminated and a more general solution be offered, inherent to the language.

2.3 An Environment for Learning

If a programming language can be both beginner friendly and advanced, it should appeal to developers with varying levels of competency. It also results in an ideal pedagogical tool, allowing a student to start with relatively abstraction-free environment, resembling a modular synthesizer, progressing towards higher abstraction and efficient programming practices.

2.4 A Future Proof Platform

Computing is undergoing a fundamental shift in the type of hardware commonly available. It is essential that any programming language designed today must be geared towards parallel computation and execution on a range of differing computational hardware.

2.5 Summary of the Design Criteria

Taking into account all of the above, the language should;

- Be designed for visual syntax and graphical user interfaces
- Provide adequate abstraction and advanced programming constructs
- Generate high performance code
- Offer a continuous learning curve from beginner to professional
- Be designed to be parallelizable and portable

3 Designing the Kronos Language

This section will make a brief case for the design choices adapted in Kronos.

3.1 Functional Programming

The functional programming paradigm[Hudak, 1989] is the founding principle in Kronos. Simultaneously fulfilling a number of our criteria, we believe it to be the ideal choice.

Compared to procedural languages, functional languages place less emphasis on the order of statements in the program source. Functional programs are essentially signal flow graphs, formed of processing nodes connected by data flow.

Graphs are straightforward to present visually. The nodes and data flows in such trees are also something most music technologists tend to understand well. Much of their work is based on making extensive audio flow graphs.

Functional programming also offers extensive abstraction and sophisticated programming constructs. These features should appeal to advanced programmers.

Further, the data flow metaphor of programming is ideally suited for parallel processing, as the language can be formally analyzed and

transformed while retaining algorithmic equivalence. This is much harder to do for a procedural language that may rely on a very particular order of execution and hidden dependencies.

Taken together, these factors make a strong case for functional programming for the purposes of Kronos and recommend its adoption. However, the functional paradigm is quite unlike what most programmers are used to. The following sections present some key differences from typical procedural languages.

3.1.1 No state

Functional programs have no state. The output of a program fragment is uniquely determined by its input, regardless of the context in which the fragment is run. Several further features and constraints emerge from this fundamental property.

3.1.2 Bindings Instead of Variables

Since the language is based on data flow instead of a series of actions, there is no concept of a changeable variable. Functional operators can only provide output from input, not change the state of any external entity.

However, symbols still remain useful. They can be used to bind expressions, making code easier to write and read.

3.1.3 Higher Order Functions Instead of Loops

Since the language has no variables, traditional loops are not possible either, as they rely on a loop iteration variable. To accomplish iterative behavior, functional languages employ recursion and higher order functions[Kemp, 2007]. This approach has the added benefit of being easier to depict visually than traditional loop constructs based on textual languages – notoriously hard to describe in a patching environment.

As an example, two higher order functions along with example replies are presented in Listing 1.

Listing 1: Higher order functions with example replies

```
/* Apply the mapping function Sqrt to all elements of a list
    */
Algorithm:Map(Sqrt 1 2 3 4 5) => (1 1.41421 1.73205 2 2.23607)
/* Combine all the elements of a list using a folding
     function, Add */
Algorithm:Fold(Add 1 2 3 4 5) => 15
```

3.1.4 Polymorphism Instead of Flow Control

A typical procedural program contains a considerable amount of branches and logic statements. While logic statements are part of functional programming, flow control often happens via *polymorphism*. Several different forms can be defined for a single function, allowing the compiler to pick an appropriate form based on the argument type.

Polymorphism and form selection is also the mechanism that drives iterative higher order functions. The implementation for one such function, *Fold*, is presented in Listing 2. *Fold* takes as an argument a folding function and a list of numbers.

While the list can be split into two parts, x and xs, the second form is utilized. This form recurs with xs as the list argument. This process continues, element by element, until the list only contains a single unsplittable element. In that boundary case the first form of the function is selected and the recursion terminates.

Listing 2: Fold, a higher order function for reducing lists with example replies.

```
Fold(folding-function x)
{
  Fold = x
}

Fold(folding-function x xs)
{
  Fold = Eval(folding-function x Fold(folding-function xs))
}

/* Add several numbers */
Fold(Add 1 2 3 4) => 10
/* Multiply several numbers */
Fold(Mul 5 6 10) => 300
```

3.2 Generic Programming and Specialization

3.2.1 Generics for Flexibility

Let us examine a scenario where a sum of several signals in differing formats is needed. Let us assume that we have defined data types for mono and stereo samples. In Kronos, we could easily define a summation node that provides mono output when all its inputs are mono, and stereo when at least one input is stereo.

An example implementation is provided in Listing 3. The listing relies on the user defining semantic context by providing types, *Mono* and *Stereo*, and providing a *Coerce* method that can upgrade a *Mono* input to a *Stereo* output.

Listing 3: User-defined coercion of mono into stereo

```
Type Mono
Package Mono{
  Cons(sample) /* wrap a sample in type context 'Mono' */
    {Cons = Make(:Mono sample)}
  Get-Sample(sample)  /* retrieve a sample from 'Mono'
       context */
    {Get-Sample = Break(:Mono sample)}
}
```

```
Type Stereo
Package Stereo{
  Cons(sample) /* wrap  a sample in type context 'Stereo' */
    {Cons = Make(:Stereo sample)}
  L/R(sample) /* provide accessors to assumed Left and Right
        channels */
    {(L R) = Break(:Stereo sample)}
}

Add(a b)
{
  /* How to add 'Mono' samples */
  Add = Mono:Cons(Mono:Get-Sample(a) + Mono:Get-Sample(b))
  /* How to add 'Stereo' samples */
  Add = Stereo:Cons(Stereo:L(a) + Stereo:L(b) Stereo:R(a) +
        Stereo:R(b))
}

Coerce(desired-type smp)
{
  /* Provide type upgrade from mono to stereo by duplicating
        channels */
  Coerce = When(
    Type-Of(desired-type) == Stereo
    Coerce = Stereo:Cons(
      Mono:Get-Sample(smp) Mono:Get-Sample(smp)))
}

/* Provide a mixing function to sum a number of channels */
Mix-Bus(ch)
{
  Mix-Bus = ch
}

Mix-Bus(ch chs)
{
  Mix-Bus = ch + Recur(chs)
}
```

Note that the function *Mix-Bus* in Listing 3 needs to know very little about the type of data passed to it. It is prepared to process a list of channels via recursion, but the only other constraint is that a summation operator must exist that accepts the kind of data passed to it.

We define summation for two mono signals and two stereo signals. When no appropriate form of *Add* can bedirectly located, as will happen when adding a mono and a stereo signal, the system-provided *Add*-function attempts to use *Coerce* to upgrade one of the arguments. Since we have provided a coercion path from mono to stereo, the result is that when adding mono and stereo signals, the mono signal gets upconverted to stereo by *Coerce* followed by a stereo summation.

The great strength of generics is that functions do not explicitly need to be adapted to a variety of incoming types. If the building blocks or primitives of which the function is constructed can handle a type, so can the function. If the complete set of arithmetic and logical primitives would be implemented for the types *Mono* and *Stereo*, then the vast majority of functions, written without any knowledge of these particular types, would be able to transparently handle them.

Generic processing shows great promise once all the possible type permutations present in music DSP are considered. Single or double precision samples? Mono, stereo or multichannel? Real- or complex-valued? With properly designed types, a singular implementation of a signal processor can automatically handle any combination of these.

3.2.2 Type Determinism for Performance

Generic programming offers great expressiveness and power to the programmer. However, typeless or dynamically typed languages have a reputation for producing slower code than statically typed languages, mostly due to the extensive amount of run time type information and reflection required to make them work.

To bring the performance on par with a static language, Kronos adopts a rigorous constraint. The output data type of a processing node may only depend on the input data type. This is the principle of *type determinism*.

As demonstrated in Listing 3, Kronos offers extensive freedom in specifying what is the result type of a function given a certain argument type. However, what is prohibited, based on type determinism, is selecting the result type of a function based on the argument data itself.

Thus it is impossible to define a mixing module that compares two stereo channels, providing a mono output when they are identical and keeping the stereo information when necessary. That is because this decision would be based on *data* itself, not the *type* of said data.

While type determinism could be a crippling deficit in a general programming language, it is less so in the context of music DSP. The example above is quite contrived, and regardless, most musical programming environments similarily prevent changes to channel configuration and routing on the fly.

Adopting the type determinism constraint allows the compiler to statically analyze the entire data flow of the program given just the data type of the initial, caller-provided input. The rationale for this is that a signal processing algorithm is typically used to process large streams of statically typed data. The result of a single analysis pass can then be reused thousands or millions of times.

3.3 Digital Signal Processing and State

A point must be made about the exclusion of stateful programs, explained in Section 3.1.1. This seems at odds with the estabilished body of DSP algorithms, many of which depend on

state or signal memory. Examples of stateful processes are easy to come by. They include processors that clearly have memory, such as echo and reverberation effects, as well as those with recursions like digital IIR filters.

As a functional language, Kronos doesn't allow direct state manipulation. However, given the signal processing focus, operations that hide stateful operations are provided to the programmer. Delay lines are provided as operators; they function exactly like the common mathematical operators. A similar approach is taken by Faust, where delay is provided as a built-in operator and recursion is an integrated language construct.

With a native delay operator it is equally simple to *delay* a signal as it is, for example, to take its square root. Further, the parser and compiler support recursive connections through these operators. The state-hiding operators aim to provide all the necessary stateful operations required to implement the vast majority of known DSP algorithms.

4 Multirate Programming

One of the most critical problems in many signal processing systems is the handling of distinct signal rates. A signal flow in a typical DSP algorithm is conceptually divided into several sections.

One of them might be the set of control signals generated by an user interface or an external control source via a protocol like OSC[Wright et al., 2003]. These signals are mostly stable, changing occasionally when the user adjusts a slider or turns a knob.

Another section could be the internal modulation structure, comprising of low frequency oscillators and envelopes. These signals typically update more frequently than the control signals, but do not need to reach the bandwidth required by audio signals.

Therefore, it is not at all contrived to picture a system containing three different signal families with highly diverging update frequencies.

The naive solution would be to adopt the highest update frequency required for the system and run the entire signal flow graph at that frequency. In practice, this is not acceptable for performance reasons. Control signal optimization is essential for improving the run time performance of audio algorithms.

Another possibility is to leave the signal rate specification to the programmer. This is the case for any programming language not specifically designed for audio. As the programmer has full control and responsibility over the execution path of his program, he must also explicitly state when and how often certain computations need to be performed and where to store those results that may be reused.

Thirdly, the paradigm of functional reactive programming[Nordlander, 1999] can be relied on to automatically determine signal update rates.

4.1 The Functional Reactive Paradigm

The constraints imposed by functional programming also turn out to facilitate automatic signal rate optimization.

Since the output of a functional program fragment depends on nothing but its input, it is obvious that the fragment needs to be executed only when the input changes. Otherwise, the previously computed output can be reused, sparing resources.

This realization leads to the functional reactive paradigm[Nordlander, 1999]. A reactive system is essentially a data flow graph with inputs and outputs. Reactions – responses by outputs to inputs – are inferred, since an output must be recomputed whenever any input changes that is directly reachable by following the data flow upstream.

4.1.1 Reactive Programming in Kronos

Reactive inputs in Kronos are called *springs*. They represent the start of the data flow and a point at which the Kronos program receives input from the outside world. Reactive outputs are called *sinks*, representing the terminals of data flow. The system can deduce which sinks receive an update when a particular input is updated.

Springs and Priority

Reactive programming for audio has some special features that need to be considered. Let us examine the delay operators presented in Section 3.3. Since the delays are specified in computational frames, the delay time of a frame becomes the inter-update interval of whatever reactive inputs the delay is connected to. It is therefore necessary to be able to control this update interval precisely.

A digital low pass filter is shown in Listing 4. It is connected to two springs, an audio signal

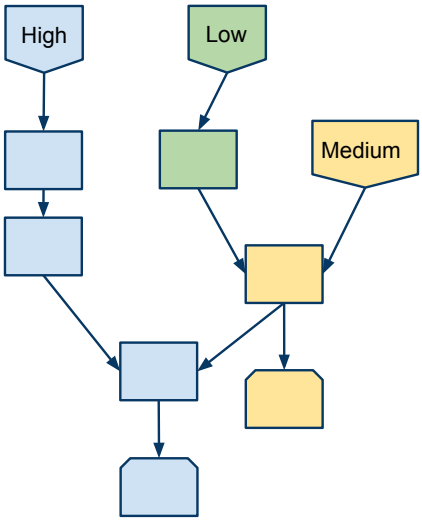

Figure 1: A reactive graph demonstrating spring priority. Processing nodes are color coded according to which spring triggers their update.

provided by the argument *x0* and an user interface control signal via OSC[Wright et al., 2003]. The basic form of reactive processing laid out above would indicate that the unit delays update whenever either the audio input or the user interface is updated.

However, to maintain a steady sample rate, we do not want the user interface to force updates on the unit delay. The output of the filter, as well as the unit delay node, should only react to the audio rate signal produced by the audio signal input.

Listing 4: A Low pass filter controlled by OSC

```
Lowpass(x0)
{
  cutoff = IO:OSC-Input("cutoff")
  y1 = z-1('0 y0)
  y0 = x0 + cutoff * (y1 - x0)
  Lowpass = y0
}
```

As a solution, springs can be given priorities. Whenever there is a graph junction where a node reacts to two springs, the spring priorities are compared. If they differ, an intermediate variable is placed at the junction and any reaction to the lower priority spring is supressed for all nodes and sinks downstream of the junction.

When the springs have equal priority, neither is supressed and both reactions propagate down the data flow. Figure 1 illustrates the reactivity inferral procedure of a graph with several springs of differing priorities.

Typically, priorities are assigned according to the expected update rate so that the highest update rate carries the highest priority.

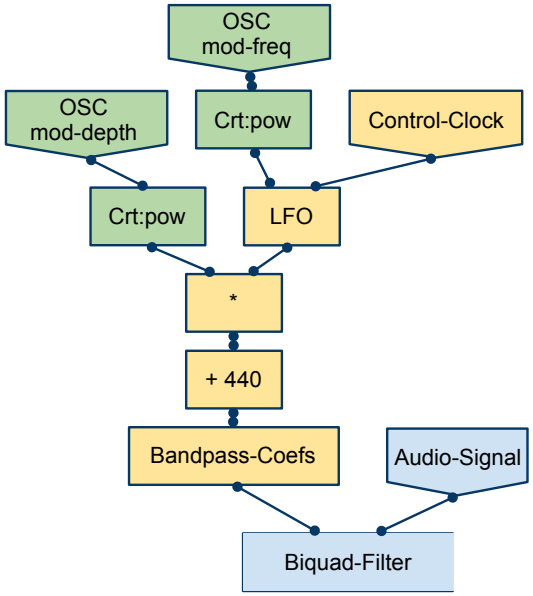

Figure 2: A practical example of a system consisting of user interface signals, coarse control rate processing and audio rate processing.

In the example shown in Listing 5 and Figure 2, an user interface signal adjusts an LFO that in turn controls the corner frequency of a band pass filter.

There are two junctions in the graph where supression occurs. Firstly, the user interface signal is terminated before the LFO computation, since the LFO control clock overrides the user interface. Secondly, the audio spring priority again overrides the control rate priority. The LFO updates propagate into the coefficient computations of the bandpass filter, but do not reach the unit delay nodes or the audio output.

Listing 5: Mixing user interface, control rate and audio rate signals

```
Biquad-Filter(x0 a0 a1 a2 b1 b2)
{
  y1 = z-1('0 y0) y2 = z-1('0 y1) x1 = z-1('0 x0) x2 = z-1('0
       x1)
  y0 = a0 * x0 + a1 * x1 + a2 * x2 - b1 * y1 - b2 * y2
}

Bandpass-Coefs(freq r amp)
{
  (a0 a1 a2) = (Sqrt(r) 0 Neg(Sqrt(r)))
  (b1 b2) = (Neg(2 * Crt:cos(freq) * r)  r * r)
  Bandpass-Coefs = (a0 a1 a2 b1 b2)
}

Vibrato-Reson(sig)
{
  Use IO
  freq = OSC-Input("freq")
  mod-depth = Crt:pow(OSC-Input("mod-depth") 3)
  mod-freq = Crt:pow(OSC-Input("mod-freq") 4)

  Vibrato-Reson = Biquad-Filter(sig
    Bandpass-Coefs(freq + mod-depth * LFO(mod-freq) 0.95
          0.05))
}
```

4.1.2 Explicit Reaction Supression

It is to be expected that the priority system by itself is not sufficient. Suppose we would like to build an envelope follower that converts the envelope of an audio signal into an OSC[Wright et al., 2003] control signal with a lower frequency. Automatic inferral would never allow the lower priority control rate spring to own the OSC output; therefore a manual way to override supression is required.

This introduces a further scheduling complication. In the case of automatic supression, it is guaranteed that nodes reacting to lower priority springs can never depend on the results of a higher priority fragment in the signal flow. This enables the host system to schedule spring updates accordingly so that lower priority springs fire first, followed by higher priority springs.

When a priority inversal occurs, such that a lower priority program fragment is below a higher priority fragment in the signal flow, the dependency rule stated above no longer holds. An undesired unit delay is introduced at the graph junction. To overcome this, the system must split the lower priority spring update into two sections, one of which is evaluated before the suppressed spring, while the latter section is triggered only after the supressed spring has been updated.

Priority inversal is still a topic of active research, as there are several possible implementations, each with its own problems and benefits.

5 Case Studies

5.1 Reverberation

5.1.1 Multi-tap delay

As a precursor to more sophisticated reverberation algorithms, multi-tap delay offers a good showcase for the generic programming capabilities of Kronos.

Listing 6: Multi-tap delay

```
Multi-Tap(sig delays)
{
  Use Algorithm
  Multi-Tap = Reduce(Add Map(Curry(Delay sig) delays))
}
```

The processor described in Listing 6 shows a concise formulation of a highly adaptable bank of delay lines. Higher order functions *Reduce* and *Map* are utilized in place of a loop to produce a number of delay lines without duplicating delay statements.

Another higher order function, *Curry*, is used to construct a new mapping function. *Curry* attaches an argument to a function. In this context, the single signal *sig* shall be fed to all the delay lines. Curry is used to construct a new delay function that is fixed to receive the *curried* signal.

This curried function is then used as a mapping function to the list of delay line lengths, resulting in a bank of delay lines, all of them being fed by the same signal source. The outputs of the delay lines are summed, using *Reduce(Add ...)*. It should be noted that the routine produces an arbitrary number of delay lines, determined by the length of the list passed as the *delays* argument.

5.1.2 Schroeder Reverberator

It is quite easy to expand the multi-tap delay into a proper reverberator. Listing 7 implements the classic Schroeder reverberation[Schroeder, 1969]. Contrasted to the multi-tap delay, a form of the polymorphic *Delay* function that features feedback is utilized.

Listing 7: Classic Schroeder Reverberator

```
Feedback-for-RT60(rt60 delay)
{ Feedback-for-RT60 = Crt:pow(#0.001 delay / rt60) }

Basic(sig rt60)
{
  Use Algorithm
  allpass-params = ((0.7 #221) (0.7 #75))
  delay-times = (#1310 #1636 #1813 #1927)

  feedbacks = Map(
    Curry(Feedback-for-RT60 rt60) delay-times)

  comb-section = Reduce(Add
    Zip-With(
      Curry(Delay sig) feedbacks delay-times))

  Basic = Cascade(Allpass-Comb comb-section allpass-params)
}
```

A third high order function, *Cascade*, is presented, providing means to route a signal through a number of similar stages with differing parameters. Here, the number of allpass comb filters can be controlled by adding or removing entries to the *allpass-params* list.

5.2 Equalization

In this example, a multi-band parametric equalizer is presented. For brevity, the implementation of the function *Biquad-Filter* is not shown. It can be found in Listing 5. The coefficient computation formula is from the widely used Audio EQ Cookbook[Bristow-Johnson, 2011].

Listing 8: Multiband Parametric Equalizer

```
Package EQ{
  Parametric-Coefs(freq dBgain q)
  {
```

```
    A = Sqrt(Crt:pow(10 dbGain / 40))
    w0 = 2 * Pi * freq
    alpha = Crt:sin(w0) / (2 * q)

    (a0 a1 a2) = ((1 + alpha * A)  (-2 * Crt:cos(w0))  (1 -
         alpha * A))
    (b0 b1 b2) = ((1 + alpha / A) (-2 * Crt:cos(w0)) (1 -
         alpha / A))

    Parametric-Coefs = ((a0 / b0) (a1 / b0) (a2 / b0) (b1 /
         b0) (b2 / b0))
  }

  Parametric(sig freqs dBgains qs)
  {
    Parametric = Cascade(Biquad-Filter
      Zip3-With(Parametric-Coefs freqs dBgains qs))
  }
}
```

This parametric EQ features an arbitrary number of bands, depending only on the size of the lists *freqs*, *dBgains* and *qs*. For this example to work, these list lengths must match.

6 Conclusion

This paper presented Kronos, a programming language and a compiler suite designed for musical DSP. Many of the principles discussed could be applied to any signal processing platform.

The language is capable of logically and efficiently representing various signal processing algorithms, as demonstrated in Section 5. As algorithm complexity grows, utilization of advanced language features becomes more advantageous.

While the language specification is practically complete, a lot of implementation work still remains. Previous work by the author on autovectorization and parallelization[Norilo and Laurson, 2009] should be integrated with the new compiler. Emphasis should be placed on parallel processing in the low latency case; a particularily interesting and challenging problem.

In addition to the current JIT Compiler for x86 computers, backends should be added for other compile targets. Being able to generate C code would greatly facilitate using the system for generating signal processing modules to be integrated into another software package. Targeting stream processors and GPUs is an equally interesting opportunity.

Once sufficiently mature, Kronos will be released as a C-callable library. There is also a command line interface. Various licensing options, including a dual commercial/GPL model are being investigated. A development of PWGLSynth[Laurson et al., 2009] based on Kronos is also planned. Meanwhile, progress and releases can be tracked on the Kronos website[Norilo, 2011].

References

J Aycock. 2003. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113.

Robert Bristow-Johnson. 2011. Audio EQ Cookbook (http://musicdsp.org/files/Audio-EQ-Cookbook.txt).

Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411.

Colin John Morris Kemp. 2007. *Theoretical Foundations for Practical Totally Functional Programming*. Ph.D. thesis, University of Queensland.

Mikael Laurson, Mika Kuuskankare, and Vesa Norilo. 2009. An Overview of PWGL, a Visual Programming Environment for Music. *Computer Music Journal*, 33(1):19–31.

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

James Nicholl. 2008. Developing applications in a patch language - A Reaktor Perspective. pages 1–23.

Johan Nordlander. 1999. *Reactive Objects and Functional Programming*. Ph.D. thesis, Chalmers University of Technology, Götebord, Sweden.

Vesa Norilo and Mikael Laurson. 2009. Kronos - a Vectorizing Compiler for Music DSP. In *Proceedings of DAFx*, pages 180–183.

Vesa Norilo. 2011. Kronos Web Resource (http://kronos.vesanorilo.com).

Y Orlarey, D Fober, and S Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632.

M Puckette. 1996. Pure data: another integrated computer music environment. In *Proceedings of the 1996 International Computer Music Conference*, pages 269–272.

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

Matthew Wright, Adrian Freed, and Ali Momeni. 2003. OpenSound Control: State of the Art 2003. *Time*, pages 153–159.