RECENT DEVELOPMENTS IN THE KRONOS PROGRAMMING LANGUAGE

Vesa Norilo

Sibelius Academy
Centre for Music & Technology, Helsinki, Finland
`mailto:vnorilo@siba.fi`

ABSTRACT

Kronos is a reactive-functional programming environment for musical signal processing. It is designed for musicians and music technologists who seek custom signal processing solutions, as well as developers of audio components.

The chief contributions of the environment include a type-based polymorphic system which allows for processing modules to automatically adapt to incoming signal types. An unified signal model provides a programming paradigm that works identically on audio, MIDI, OSC and user interface control signals. Together, these features enable a more compact software library, as user-facing primitives are less numerous and able to function as expected based on the program context. This reduces the vocabulary required to learn programming.

This paper describes the main algorithmic contributions to the field, as well as recent research into improving compile performance when dealing with block-based processes and massive vectors.

1. INTRODUCTION

Kronos is a functional reactive programming language[8] for signal processing tasks. It aims to be able to model musical signal processors with simple, expressive syntax and very high performance. It consists of a programming language specification and a reference implementation that contains a just in time compiler along with a signal I/O layer supporting audio, OSC[9] and MIDI.

The founding principle of this research project is to reduce the *vocabulary* of a musical programming language by promoting signal processor design patterns to integrated language features. For example, the environment automates signal update rates, eradicating the need for similar but separate processors for audio and control rate tasks.

Further, signals can have associated type semantics. This allows an audio processor to configure itself to suit an incoming signal, such as mono or multichannel, or varying sample formats. Together, these language features serve to make processors more flexible, thus requiring a smaller set of them.

This paper describes the state of the Kronos compiler suite as it nears production maturity. The state of the freely available beta implementation is discussed, along with issues that needed to be addressed in recent development work – specifically dealing with support for massive vectors and their interaction with heterogenous signal rates.

As its main contribution, this paper presents an algorithm for *reactive factorization* of arbitrary signal processors. The algorithm is able to perform automatic signal rate optimizations without user intervention or effort, handling audio, MIDI and OSC signals with a unified set of semantics. The method is demonstrated via Kronos, but is applicable to any programming language or a system where data dependencies can be reliably reasoned about. Secondly, this method is discussed in the context of heterogenous signal rates in large vector processing, such as those that arise when connecting huge sensor arrays to wide ugen banks.

This paper is organized as follows; in Section 2, *Kronos Language Overview*, the proposed language and compiler are briefly discussed for context. Section 3 describes an algorithm that can perform intelligent signal rate factorization on arbitrary algorithms. Section 4, *Novel Features*, discusses in detail the most recent developments. Finally, the conclusions are presented in Section 5.

2. KRONOS LANGUAGE OVERVIEW

Kronos programs can be constructed as either textual source code files or graphical patches. The functional model is well suited for both representations, as functional programs are essentially data flow graphs.

2.1. Functional Programming for Audio

Most of a Kronos program consists of function definitions, as is to be expected from a functional programming language. Functions are compositions of other functions, and each function models a signal processing stage. Per usual, functions are first class and can be passed as inputs to other, higher order functions.

This allows traditional functional programming staples such as map, demonstrated in Figure 1. In the example, a higher order function called *Algorithm:Map* receives from the right hand side a set of control signals, and applies a transformation specified on the left hand side, where each frequency value becomes an oscillator

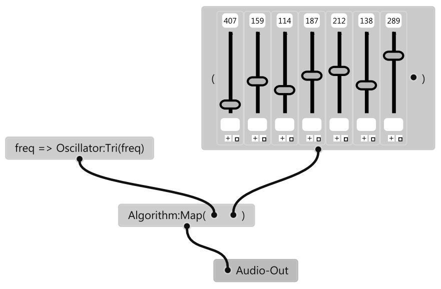

Figure 1. Mapping a set of sliders into an oscillator bank

at that frequency. For a thorough discussion, the reader is referred to previous work[3].

2.2. Types and Polymorphism as Graph Generation

Kronos allows functions to attach type semantics to signals. Therefore the system can differentiate between, say, a stereo audio signal and a stream of complex numbers. In each case, a data element consists of two real numbers, but the semantic meaning is different. This is accomplished by Types. A type annotation is essentially a semantic tag attached to a signal of arbitrary composition.

Library and user functions can then be overloaded based on argument types. Signal processors can be made to react to the semantics of the signal they receive. Polymorphic functions have powerful implications for musical applications; consider, for example, a parameter mapping strategy where data connections carry information on parameter ranges to which the receiving processors can automatically adjust to.

2.2.1. Type Determinism

Kronos aims to be as expressive as possible at the source level, yet as fast as possible during signal processing. That is why the source programs are *type generic*, yet the runtime programs are *statically typed*. This means that whenever a Kronos program is launched, all the signal path types are deduced from the context. For performance reasons, they are fixed for the duration of a processing run, which allows polymorphic overload resolution to happen at compile time.

This fixing is accomplished by a mechanism called *Type Determinism*. It means that the result type of a function is uniquely determined by its argument types. In other words, type can affect data, but not vice versa. This leads to a scheme where performant, statically typed signal graphs can be generated from a type generic source code. For details, the reader is referred to previous work[6].

2.3. Multirate Processing

Kronos models heterogenous signal rates as discrete update events within continuous “staircase” signals. This allows the system to handle sampled audio streams and sparse event streams with an unified[5] signal model. The entire signal graph is synchronous and the reactive update model imposes so little overhead that it is entirely suitable to be used at audio rates.

This is accomplished by defining certain *active* external inputs to a signal graph. The compiler analyzes the data flow in order to determine a combination of active inputs or *springs* that drive a particular node in the graph.

Subsequently, different activity states can be modeled from the graph by only considering those nodes that are driven by a particular set of springs. This allows for generating a computation graph for any desired set of external inputs, leaving out any operations whose output signal is unchanged during the activation state.

For example, user interface elements can drive filter coefficient computations, while the audio clock drives the actual signal processing. However, there’s no need to separate these sections in the user program. The signal flow can be kept intact, and the distinction between audio and control rate becomes an optimization issue, handled by the compiler, as opposed to a defining the structure of the entire user program.

3. REACTIVE FUNCTIONAL AS THE UNIVERSAL SIGNAL MODEL

3.1. Dataflow Analysis

Given an arbitrary user program, all signal data flows should be able to be reliably detected. For functional programming languages such as Kronos or Faust[7], this is trivial, as all data flows are explicit. The presence of any implicit data flows, such as the global buses in systems like SuperCollider[1] can pose problems for the data flow analysis.

3.2. Reactive Clock Propagation

The general assumption is that a node is active whenever any of its upstream nodes are active. This is because logical and arithmetic operations will need to be recomputed whenever any of their inputs change.

However, this is not true of all nodes. If an operation merely combines unchanged signals into a vectored signal, it is apposite to maintain separate clocking records for the components of the vectored signal rather than have all the component clocks drive the entire vector. When the vector is unpacked later, subsequent operations will only join the activation states of the component signals they access.

Similar logic applies to function calls. Since many processors manifest naturally as functions that contain mixed rate signal paths, all function inputs should preferably have distinct activation states.

3.3. Stateful Operations and Clock

The logic outlined in section 3.2 works well for strictly functional nodes – all operations whose output is uniquely determined by their inputs rather than any state or memory.

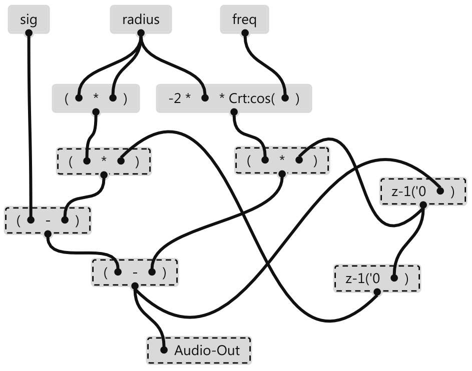

Figure 2. A Filter with ambigious clock sources

However, state and memory are important for many DSP algorithms such as filters and delays. Like Faust[7], Kronos deals with them by promoting them to language primitives. Unit delays and ring buffers can be used to connect to a time-delayed version of the signal graph. This yields an elegant syntax for delay operations while maintaining strict functional style *within* each update frame.

For strictly functional nodes, activation is merely an optimization. For stateful operations such as delays, it becomes a question of algorithmic correctness. Therefore it is important that stateful nodes are not activated by any springs other than the ones that define their desired clock rate. For example, the unit delays in a filter should not be activated by the user interface elements that control their coefficients to avoid having the signal clock disrupted by additional update frames from the user interface.

A resonator filter with a *signal* input and two control parameters *freq* and *radius* is shown in Figure 2. The nodes that see several clock sources in their upstream are indicated with a dashed border. Since these include the two unit delay primitives, it is unclear which clock should determine the length of the unit delay.

3.3.1. Clock Priority

The clocking ambiguities can be resolved by assigning priorities to the springs that drive the signal graph. This means that whenever a node is activated by multiple springs, some springs can preclude others.

The priority can be implemented by a strict-weak ordering criteria, where individual spring pairs can either have an ordered or an unordered relation. Ordered pairs will only keep the dominant spring, while unordered springs can coexist and both activate a node. The priority system is shown in Figure 3. The audio clock dominates the control signal clocks. Wires that carry control signals are shown hollow, while audio signal wires are shown solid black. This allows the audio clock to control the unit delays over sources of lesser priority.

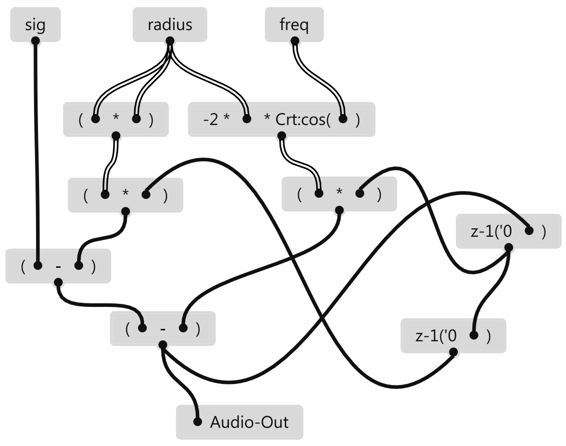

Figure 3. A Filter with clocking ambiguity resolved

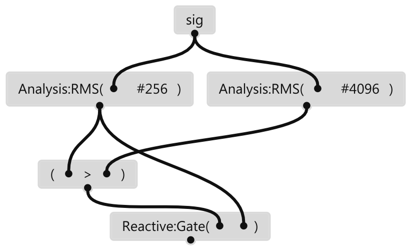

Figure 4. Dynamic clock from a Transient Detector

3.3.2. Dynamic Clocking and Event Streams

The default reactivity scheme with appropriate spring priorities will result in sensible clocking behavior in most situations. However, sometimes it may be necessary to override the default clock propagation rules.

As an example, consider an audio analyzer such as a simple transient detector. This processor has an audio input and an event stream output. The output is activated by the input, but only sometimes; depending on whether the algorithm decides a transient occurred during that particular activation.

This can be implemented by a clock gate primitive, which allows a conditional inhibition of activation. With such dynamic activation, the reactive system can be used to model event streams – signals that do not have a regular update interval. This accomplishes many tasks that are handled with branching in prodecural languages, and in the end results in similar machine code. A simple example is shown in Figure 4. The *Reactive* : *Gate* primitive takes a truth value and a signal, inhibiting any clock updates from the signal when the truth value is false. This allows an analysis algorithm to produce an event stream from features detected from an audio stream.

Table 1. Activation State Matrix

Clock	X											
Clock $\times$ 3	X				X				X			
Clock $\times$ 4	X			X			X			X		

3.3.3. Upsampling and Decimation

For triggering several activations from a single external activation, an upsampling mechanism is needed. A special purpose reactive node can be inserted in the signal graph to multiply the incoming clock rate by a rational fraction. This allows for both up- and downsampling of the incoming signal by a constant factor. For reactive priority resolution, clock multipliers sourced from the same external clock are considered unordered.

To synchronously schedule a number of different rational multiplies of an external clock, it is necessary to construct a super-clock that ticks whenever any of the multiplier clocks might tick. This means that the super-clock multiplier must be divisible by all multiplier numerators, yet be as small as possible. This can be accomplished by combining the numerators one by one into a reduction variable S with the formula in Equation (1)

$$f(a,b) = \frac{ab}{\gcd(a,b)} \tag{1}$$

To construct an activation sequence from an upsampled external clock, let us consider the sequence of S super-clock ticks it triggers. Consider the super-clock multiplier of S and a multiplier clock $\frac{N}{M}$. In terms of the super-clock period, the multiplier ticks at $\frac{N}{SM}$. This is guaranteed to simplify to $\frac{1}{P}$, where P is an integer – the period of the multiplier clock in super-clock ticks.

Within a period of S super-clock ticks, the multiplier clock could potentially activate once every $\gcd(S,P)$ ticks. In the case of $P = \gcd(S,P)$ the activation pattern is deterministic. Otherwise, the activation pattern is different for every tick of the external clock, and counters must be utilized to determine which ticks are genuine activations to maintain the period P. An activation pattern is demonstrated in Table 1.

This system guarantees exact and synchronous timing for all rational fraction multipliers of a signal clock. For performance reasons, some clock jitter can be permitted to reduce the number of required activation states. This can be done by merging a number of adjacent super-clock ticks. As long as the merge width is less than the smallest P in the clock system, the clocks maintain a correct average tick frequency with small momentary fluctuations. An example of an activation state matrix is shown in Figure 1. This table shows a clock and its multiplies by three and four, and the resulting activation combinations per super-clock tick.

Table 2. Compilation passes performed by Kronos Beta

Pass	***Description***
1. Specialization	*Generic functions to typed functions and overload resolution*
2. Reactivity	*Reactive analysis and splitting of typed functions to different activation states*
3. Side Effects	*Functional data flows to pointer side effects*
4. Codegen	*Selection and scheduling of x86 machine instructions*

3.3.4. Multiplexing and Demultiplexing

The synchronous multirate clock system can be leveraged to provide oversampled or subsampled signal paths, but also several less intuitive applications.

To implement a multiplexing or a buffering stage, a ring buffer can be combined with a signal rate divider. If the ring buffer contents are output at a signal rate divided by the length of the buffer, a buffering with no overlap is created. Dividing the signal clock by half of the buffer length yields a 50% overlap, and so on.

The opposite can be achieved by multiplying the clock of a vectored signal and indexing the vector with a ramp that has a period of a non-multiplied tick. This can be used for de-buffering a signal or canonical insert-zero upsampling.

3.4. Current Implementation in Kronos

The reactive system is currently implemented in Kronos Beta as an extra pass between type specialization and machine code generation. An overview of the compilation process is described in Table 2.

The reactive analysis happens relatively early in the compiler pipeline, which results in some added complexity. For example, when a function is factored into several activation states, the factorizer must change some of the types inferred by the specialization pass to maintain graph consistency when splitting user functions to different activation states.

Further, the complexity of all the passes depends heavily on the data. During the specialization pass, a typed function is generated for each different argument type. For reacursive call sequences, this means each iteration of the recursion. While the code generator is able to fold these back into loops, compilation time grows quickly as vector sizes increase. This hardly matters for the original purpose of the compiler, as most of the vector sizes were in orders of tens or hundreds, representing parallel ugen banks.

However, the multirate processing and multiplexing detailed in Section 3.3.4 are well suited for block processes, such as FFT, which naturally need vector sizes from several thousand to orders of magnitude upwards. Such processes can currently cause compilation times from tens of seconds to minutes, which is not desirable for a quick development cycle and immediate feedback. The newest developments on Kronos focus on, amongst other things, decoupling compilation time from data complexity. The relationship of these optimizations to reactive factorization is explored in the following Section 4.

4. NEW DEVELOPMENTS

Before Kronos reaches production maturity, a final rewrite is underway to simplify the overrall design, improve the features and optimize performance. This section discusses the improvements over the beta implementation.

4.1. Sequence Recognition

Instead of specializing a recursive function separately for every iteration, it is desirable to detect such sequences as early as possible. The new version of the Kronos compiler has a dedicated analysis algorithm for such sequences.

In the case of a recursion, the evolution of the induction variables is analyzed. Because Kronos is type deterministic, as explained in Section 2.2.1, the overload resolution is uniquely determined by the types of the induction variables.

In the simple case, an induction variable retains the same type between recursions. In such a case, the overload resolution is *invariant* with regard to the variable. In addition, the analyzer can handle homogenous vectors that grow or shrink and compile time constants with simple arithmetic evolutions. Detected evolution rules are *lifted* or separated from the user program. The analyzer then attempts to convert these into recurrence relations, which can be solved in closed form. Successful analysis means that a sequence will have identical overload resolutions for N iterations, enabling the internal representation of the program to encode this efficiently.

Recognized sequences are thus compiled in constant time, independent from the size of data vectors involved. This is in contrast to Kronos Beta, which compiled vectors in linear time. In practice, the analyzer works for functions that iterate over vectors of homogenous values as well as simple induction variables. It is enough to efficiently detect and encode common functional idioms such as *map*, *reduce*, $unfold$ and *zip*, provided their argument lists are homogenous.

4.2. New LLVM Backend

As a part of Kronos redesign, a decision was made to push the reactive factorization further back in the compilation pipeline. Instead of operating in typed Kronos functions, it would operate on a low level code representation, merely removing code that was irrelevant for the activation state at hand.

This requires some optimization passes *after* factorization, as well as an intermediate representation between Kronos syntax trees and machine code. Both of these are readily provided by the widely used *LLVM*, a compiler component capable of abstracting various low level instruction sets. LLVM includes both a well designed intermediate representation as well as industry strength optimization passes. As an extra benefit, it can target a number of machine architectures without additional development effort.

In short, the refactored compiler includes more compilation passes than the beta version, but each pass is simpler. In addition, the LLVM project provides several of them. The passes are detailed in Table 3, contrasted to Table 2.

Table 3. Compilation passes performed by Kronos Final

Pass	***Description***
1. Specialization	*Generic functions to typed functions and overload resolution* *Sequence recognition and encoding*
2. Reactive analysis	*Reactive analysis*
3. Copy Elision	*Dataflow analysis and copy elision*
4. Side Effects	*Functional data flows to pointer side effects*
5. LLVM Codegen	*Generating LLVM IR with a specific activation state*
6. LLVM Optimization	*Optimizing LLVM IR*
7. Native Codegen	*Selection and scheduling of x86 machine instructions*

4.3. Reactive Factoring of Sequences

The newly developed sequence recognition creates some new challenges for reactive factorization. The basic functions of the two passes are opposed; the sequence analysis combines several user functions into a compact representation for compile time performance reasons. The reactive factorization, in contrast, splits user functions in order to improve run time performance.

A typical optimization opportunity that requires cooperation between reactive analysis and sequence recognition would be a bank of filters controlled by a number of different control sources. Ideally, we want to maintain an efficient sequence representation of the audio section of those filters, while only recomputing coefficients when there is input from one of the control sources.

If a global control clock is defined that is shared between the control sources, no special actions are needed. Since all iterations of the sequence see identical clocks at the input side, they will be identically factored. Thus, the sequence iteration can be analyzed once, and the analysis is valid for all the iterations. The LLVM Codegen sees a loop, and depending on the activation state it will filter out different parts of the loop and provide the plumbing between clock regions.

Forcing all control signals to tick at a global control rate could make the patches easier to compile efficiently. However, this breaks the unified signal model. A central motivation of the reactive model is to treat event-based and streaming signals in the same way. If a global control clock is mandated, signal models such as MIDI streams could no longer maintain the natural relationship between an incoming event and a clock tick. Therefore, event streams such as the user interface and external control interfaces should be considered when designing the sequence factorizer.

4.3.1. Heterogenous Clock Rates in Sequences

Consider a case where each control signal is associated with a different clock source. We would still like to maintain the audio section as a sequence, but this is no longer possible for the control section, as each iteration responds to a different activation state.

In this case, the reactive factorization must compute a distinct activation state for each iteration of the sequence. If there is a section of the iteration with an invariant activation state, this section can be factored into a sequence of its own.

Such sequence factorization can be achieved via *hylopmorphism*, which is the generalization of recursive sequences. The theory is beyond the scope of this article, but based on the methods in literature[2], any sequence can be split into a series of two or more sequences. In audio context, this can be leveraged so that as much activation-invariant code as possible can be separated into a sequence that can be maintained throughout the compilation pipeline. The activation-variant sections must then be wholly unrolled. This allows the codegen to produce highly efficient machine code.

5. CONCLUSIONS

This paper presented an overview of Kronos, a musical signal processing language, as well as the design of its reactive signal model. Kronos is designed to increase the flexibility and generality of signal processing primitives, limiting the vocabulary that is requisite for programming. This is accomplished chiefly via the type system and the polymorphic programming method as well as the unified signal model.

The reactive factorization algorithm presented in this paper can remove the distinction between events, messages, control signals and audio signals. Each signal type can be handled with the same set of primitives, yet the code generator is able to leverage automatically deduced signal metadata to optimize the resulting program.

The concepts described in this paper are implemented in a prototype version of the Kronos compiler which is freely available along with a visual, patching interface[4]. For a final version, the compiler is currently being redesigned, scheduled to be released by the summer of 2013. The compiler will be available with either a GPL3 or a commercial license.

Some new developments of a redesigned compiler were detailed, including strategies for handling massive vectors. This is required for a radical improvement in compilation times for applications that involve block processing, FFTs and massive ugen banks. As Kronos aims to be an environment where compilation should respond as quickly as a play button, this is critical for the feasibility of these applications.

As the compiler technology is reaching maturity, further research will be focused on building extensive, adaptable and learnable libraries of signal processing primitives for the system. Interaction with various software platforms is planned. This takes the form of OSC communication as well as code generation – Kronos can be used to build binary format extensions, which can be used as plugins or extensions to other systems. LLVM integration opens up the possibility of code generation for DSP and embedded devices. Finally, the visual programming interface will be pursued further.

6. REFERENCES

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

[2] S.-C. Mu and R. Bird, “Theory and applications of inverting functions as folds,” *Science of Computer Programming*, vol. 51, no. 12, pp. 87–116, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642304000140

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

[4] ——, “Visualization of Signals and Algorithms in Kronos,” in *Proceedings of the International Conference on Digital Audio Effects*, York, United Kingdom, 2012.

[5] V. Norilo and M. Laurson, “Unified Model for Audio and Control Signals,” in *Proceedings of ICMC*, Belfast, Northern Ireland, 2008.

[6] ——, “A Method of Generic Programming for High Performance DSP,” in *DAFx-10 Proceedings*, Graz, Austria, 2010, pp. 65–68.

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

[8] Z. Wan and P. Hudak, “Functional reactive programming from first principles,” in *Proceedings of the ACM SIGPLAN 2000*, ser. PLDI ’00. ACM, 2000, pp. 242–252.

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