Kronos
Reimagining Musical Signal Processing

Vesa Norilo

March 14, 2016

PREFACE

THANKS AND ACKNOWLEDGEMENTS

First of all, I wish to thank my supervisors; Dr. Kalev Tiits, Dr. Marcus Castrén and Dr. Lauri Savioja, for guidance and some necessary goading.

Secondly; this project would never have materialized without the benign influence of Dr Mikael Laurson and Dr Mika Kuuskankare. I learned most of my research skills working as a research assistant in the PWGL project, which I had the good fortune to join at a relatively early age. Very few get such a head start.

Most importantly I want to thank my family, Lotta and Roi, for their love, support and patience. Many thanks to Roi's grandparents as well, who have made it possible for us to juggle an improbable set of props: freelance musician careers, album productions, trips around the world, raising a baby and a couple of theses on the side.

This thesis is typeset in LaTeX with the Ars Classica stylesheet generously shared by Lorenzo Pantieri.

THE APPLIED STUDIES PROGRAM PORTFOLIO

This report is a part of the portfolio required for the Applied Studies Program for the degree of Doctor of Music. It consists of an introductory essay, supporting appendices and six internationally peer reviewed articles.

The portfolio comprises of this report and a software package, *Kronos*. Kronos is a programming language development environment designed for musical signal processing. The contributions of the package include the specification and implementation of a compiler for this language.

Kronos is intended for musicians and music technologists. It aims to facilitate creation of signal processors and digital instruments for use in the musical context. It addresses the research questions that arose during the development of *PWGLSynth*, the synthesis component of *PWGL*, an environment on which the author collaborated with Dr Mikael Laurson and Dr Mika Kuuskankare.

Kronos is available in source and binary form at the following address: `https://bitbucket.org/vnorilo/k3`

CONTENTS

LIST OF PUBLICATIONS

This report consist of an introductory essay, supporting appendices, and the following six articles referred to as *P1–P6*.

P1 Vesa Norilo. Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. *Computer Music Journal*, 39(4), 2015

P2 Vesa Norilo and Mikael Laurson. A Unified Model for Audio and Control Signals in PWGLSynth. In *Proceedings of the International Computer Music Conference*, Belfast, 2008

P3 Vesa Norilo. Introducing Kronos - A Novel Approach to Signal Processing Languages. In Frank Neumann and Victor Lazzarini, editors, *Proceedings of the Linux Audio Conference*, pages 9–16, Maynooth, 2011. NUIM

P4 Vesa Norilo. Designing Synthetic Reverberators in Kronos. In *Proceedings of the International Computer Music Conference*, pages 96–99, Huddersfield, 2011

P5 Digital Audio Effects. Kronos Vst – the Programmable Effect Plugin. In *Proceedings of the International Conference on Digital Audio Effects*, Maynooth, 2013

P6 Vesa Norilo. Recent Developments in the Kronos Programming Language. In *Proceedings of the International Computer Music Conference*, Perth, 2013

Part I

Introductory Essay

1 BACKGROUND

1.1 SIGNAL PROCESSING FOR MUSIC: THE MOTIVATION

Musical signal processing is an avenue of creative expression as well as a realm for commercial innovation. Composers require unheard digital instruments for creative purposes, sound engineers apply novel algorithms to further the recording arts, musicologists leverage exotic mathematics for sophisticated music information retrieval, while designers and engineers contribute exciting products to the vibrant scene of amateurs and autodidacts. Signal processor design is luthiery in the digital age.

Design and realization of signal processors by musicians is a topic that has attracted a lot of research since the seminal MUSIC III [7]. The activity in this field suggests that the related questions are not satisfactorily resolved. A survey of the state of art is given in Section 1.2. In broad terms, this study presents the evolution of musical signal processing as gradual application of ideas from computer science into a programming metaphor, the *unit generator graph*, a digital representation of signal flow in a modular synthesis system. This process is still in ongoing, and a significant body of work in general computer science awaits exploration.

Three research projects are outstandingly influential to this study. SuperCollider by McCartney [8] applies the object oriented programming paradigm to musical programming. It sets a precedent in demonstrating that elevating the level of abstraction in a programming language doesn't necessarily make it more difficult to learn, but certainly facilitates productivity. Faust by Orlaley et al. [9] applies functional programming to low level signal processing, accomplishing this transformative combination by a custom-developed compiler stack. Lastly, the PWGL project, in which the author has collaborated with Laurson and Kuuskankare [10], during which many of the research problems addressed by this study originally arose.

The traditional wisdom states that high performance real time code needs to be written in a low level idiom with a sophisticated optimizing compiler, such as C++ [11]. Improved processing power hasn't changed this: the demand for higher resolution, more intesive algorithms and increasing polyphony and complexity has kept slow, high level languages out of the equation. On the other hand, learning industrial languages such as C is not an enticing proposition for domain experts, such as composers and musicians. The rest of this section presents the rationale and inspiration for designing a signal processing language designed explicitly for them.

1.1.1 Artistic Creativity and Programming

Technical and artistic creativity are widely regarded as separate, if not indeed opposite, qualities. In an extreme, technical engineering can be seen as an optimization process guided by quantifiable utility functions. The act of artistic creation is often discussed in more mystical, ephemeral terms. The object of this study is not to deliberate on this dichotomy. However, the practice of instrument building, including mechanical, electronic and digital, certainly contains aspects of both the former and a latter stereotype. The emergence of digital musicianship [12] involves musicians building

their instruments and developing performative virtuosity in realms like mathematics and computer programming.

A similar trend is observed by Candy in academic research conducted by artists [13, p. 38]; tool building and the development of new technology can both enable and result from artistic research.

Thus, a re-examination of programming tools related to artistic creation could be fruitful. This is programming in the realm of vague or unknown utility functions. Key aspects of the workflow are rapid feedback and iterative development. Regardless of whether the act of programming is performative in and of itself, the interaction between the machine and programmer tends to be conversational [14]. Algorithm development and refinement happen in quickly alterating modify–evaluate cycles. The evaluation is typically perceptual; in the case of music, the code must be *heard* to be assessed. This is a marked contrast to the traditional enterprise software development model of specification, implementation and testing, although similarities to newer methods are apparent.

1.1.2 Ideas From Prototype to Product

Programming languages have diverse goals. The most common such goals include run time and compile time efficiency, correctness and safety, developer productivity and comfort. Sometimes there is synergy between a number of these goals, sometimes they conflict. Efficiency often requires sacrifices in the other categories. Safety can run counter to productivity and efficiency.

Prototypes are often made with tools that are less concerned with efficiency, correctness and safety, and prioritize quick results and effortless programming. Once the need for rapid changes and iteration has passed, final products can be finalized with more work-intensive methods and languages that result in safe, efficient and polished products.

Ideally, the same tools should be fit for both purposes. This is especially the case in musical signal processing, where the act of programming or prototyping is more or less continual, even performative. The signal processor in development must still respond adequately; offer high real time performance, never crash or run out of memory. In a sense, a prototype must share many qualities of a finalized product.

Such a combination of features is more feasible if the language is narrowly targeted. Complex concerns such as safe yet efficient dynamic heap management are central to general purpose language design. A domain language can take a simple stance: in signal processing, dynamic memory semantics should not be required at all. Similarly, if a language enforces a strictly delimited programming model, safety, efficiency and ease of development can all be provided, as the programs stay within predefined constraints. As more complicated requirements, such as parallel execution of programs, arise, strict programming models become ever more important.

The design problem then becomes one of reduction. What can be removed from a programming language in order to make it ideally suited for automated optimizers, safe memory semantics and seemingly typeless source notation? Does reduction also enable something new to emerge? Perhaps a combination of constructs and paradigms that was prohibitively inefficient in more broadly specified languages, can now achieve transformatively high performance via compiler optimization.

1.1.3 Empowering Domain Experts

The value of a code fragment written for artistic purposes can be perceptual and hard to quantify. It is often hard to communicate as well. Collaboration between expert programmers and expert musicians tends to be difficult, especially when new ground is to be broken.

Many musicians opt to build their own digital instruments. Likewise, effects processor design requires the sensibilities of an experienced live sound or studio engineer. Such activity is highly

cross-disciplinary by default. One aim of a domain specific programming environment is to lower the technical barrier to entry. This enables a larger portion of musicians and music technologists to better express their algorithmic sound ideas without the assistance of an expert programmer.

The author of this report suspects that programming tools that promote artistic creativity in programming, more specifically those that employ conversational programming [14], are more suitable and easier to adopt for musicians. A proper scientific examination of this hypothesis is beyond the scope of the present study, but it is nevertheless acknowledged as a part its background motivation.

One can further speculate that such empowerment of domain experts would lead to innovation and furthering of the state of art in musical signal processing. The present study is an attempt to fullfill some of the technical prerequisites to such experiments.

1.2 STATE OF ART

This section presents a brief overview of the evolution and state of art in musical signal processing. A more technically detailed discussion is given in P1.

Many of the aspirations for a musical programming language are for a combination of features from multiple existing languages or environments. Some of them follow from the fact that musical domain specific languages tend to be used by non-programmers. The domain can tolerate a loss of generality if it is accompanied by an improvement in the workflow of accomplishing common musical tasks. A representative example of such a tradeoff is the traditional unit generator paradigm. This paradigm is exceedingly successful in the field, despite common implementations allowing next to no abstraction and only simple composition of nodes that are prebuilt and supplied with the environment. Simplicity can be a strength; the further one taps into computer science, the greater care must be taken to design for music practitioners; complicated abstraction must be presented in a clear and approachable manner.

1.2.1 The Unit Generator Graph

The *unit generator graph* is the most influential programming model in the history of musical signal processing. The MUSICn family by Mathews [15, p. 187] is widely considered [16] to have estabilished this paradigm, which has since appeared in the majority of musical programming environments. Csound [17] is the direct contemporary descendant of the MUSIC series.

Unit generators or *ugens* fulfill a dual role of both the programming model and the implementation strategy for many of these environments. *Ugens* are defined as primitive operations in terms of signal input and output. User programs are composed by interconnecting the inputs and outputs of simple *ugens*. The actual code for the input–output processing itself is typically a "black box", provided as native code component, out of reach of the programmer. The opaqueness of such environments prevents programmers from studying the inner workings of the modules they are using.

A visual representation of an *ugen* connection graph is a dataflow diagram. Since the primary method of programming is composing and connecting *ugens*, a visual programming surface is an easy fit. Max [18] and Pure Data [19] are examples of graphical *ugen* languages.

Ugens also provide a model of program composition. New *ugens* can be defined in terms of the existing ones, by describing their input–output processing as a *ugen* graph. In theory, this method of composition scales from primitive *ugens* to simple signal processors built of them, and finally complicated systems built from the simple processors. Csound [17] provides the ability of defining *opcodes* – the Csound term for *ugens* – built from other opcodes, on multiple levels. Pure Data

does less to encourage such composition, but provides a mechanism to hide *ugen subgraphs* behind abstract graph nodes.

1.2.2 Aspects of Programming Language Theory

Unit generators can be compared to the basic compositional elements in other programming paradigms. In MUSIC III [7] and its descendants, *ugens* and instruments correspond to classes in object oriented programming while *ugen* instances correspond to objects [20].

The Pure Data [19] model corresponds closely to the object model in the Smalltalk tradition [21], where objects send messages to each other. In PD, node inlets can be considered to be *selectors*, with the connector cables describing the messaging flow.

The more advanced aspects of object orientation are out of reach of the visual representation. Delegation, composition and subtyping are not generally achievable in Pure Data [19], although the Odot project [22] provides interesting, if limited, extensions.

SuperCollider [8] takes the object oriented approach further. A high level object language with concepts like higher order functions and dynamic objects is used to construct a *ugen graph* for signal processing. The SuperCollider synthesis graph is interpreted; composed from relatively large hermetic, built-in code blocks, as directed by the front end program. This method is effective, but forces the back end ugens to be considerably less flexible than the front end script idiom due to more stringent performance targets. The related technical details are further explained in P1.

To transcend the limitations of *ugen* interpreters, compilation techniques can be employed. Faust [23] is a prominent example of a bespoke compiler system for musical signal processing. Faust provides first class functions and caters for some functional programming techniques, yet is capable of operating on the sample level, with unit delay recursion. Such a combination is made possible by employing code transformation and optimization passes from source form to natively executable machine code.

1.2.3 The Multirate Problem

A staple of signal processing efficiency is the management of signal update rates. Typical systems, again following the MUSICn tradition, specifically MUSIC 11 [15, p. 187], are divided into audio and control sections. The former always operate at audible bandwidths, while the latter may be roughly as slow as the human event resolution. The required update rates for these sections may differ by an order of magnitude, which has a significant impact on computational efficiency.

Most systems maintain the distinction between control and audio rate. Some compute everything at the audio rate, which is hardly efficient. In SuperCollider [8], most *ugens* can be instantiated at either rate, while Pure Data [19] divides the *ugens* to distinct groups that deal with either control or audio signals. Further, Pure Data represents control data as a series of discrete non-synchronous events that do not coincide with a regular update clock. Some recent systems like multirate Faust [24] and Csound 6 provide the option for several different control rates.

Some signals are endemically composed of discrete events, such as MIDI or OSC [25] messages or user interface interaction. A complicated system might require a large number of update schemes: audio, fine control, coarse control, MIDI events and user interface events.

Most systems deal with audio and control rates that are only globally adjustable. The signal rate boundaries also tend to add to the verbosity and complexity of source code. An interesting alternative solution to the multirate problem is proposed by Wang [26]; the signal processor is defined as a combination of a *ugen graph* and a control script that are co-operatively scheduled, with the control script yielding time explicitly for a well defined sleep period. Thus, the control

script with its flexible processing intervals replaces the control section of the signal processing system. However, the dichotomy between audio and control remains.

1.3 RESEARCH PROBLEM

The research problem in this study is formulated as a design for a programming language and run time for musical signal processing. Firstly, the survey of the state of art is examined to identify open problems in the current practice, which the language aims to address. Secondly, the language design is geared towards enabling technological innovation by domain experts, as motivated in Section 1.1.

The hypothesis is that theory from computer science can be deployed to accomplish the stated goals. Further, by specifying the language as compiled rather than interpreted, underutilized programming paradigms and models can become viable. Compilation enables more significant program transformations to take place, allowing more design freedom to formulate the mapping from a desirable source form to efficient and satisfactory machine code.

To think that a project of such a limited scope as this one could outperform world class programming languages and compilers in the general case is somewhat irrational. The design criteria must therefore be specified as a novel set of tradeoffs that result from the specific characteristics of musical signal processing as a narrowly defined domain.

1.3.1 Open Questions

An expert programmer would likely find most musical programming environments unproductive. Staples of general purpose languages such as code reuse, modularity and abstraction are less developed. Many visual environments struggle to represent basic constructs like loops, leading users to duplicate code manually. There are a number of factors that work against the adoption of helpful abstraction in these environments.

Firstly, the common *ugen interpretation* scheme favours large, monolithic *ugens* that spend as much processor time in their inner processing loop as possible, per dispatch. This is for efficiency reasons. The inner loops are usually opaque to the *ugen interpreter*, having been built in a more capable programming language. The technical limitations derail *ugen* design from simple, modular components to large, monolithic ones. The promise of the *ugen graph* as a compositional model is not realized.

Secondly, the potential of visual programming is often not exploited fully. There is an obvious correspondence between functional data flow prorams and the graphical signal flow diagram. Yet, staples of functional programming, such as polymorphism and higher order functions are largely absent from the existing visual programming surfaces. Perhaps these programming techniques are considered too advanced to incorporate in a domain language for non-programmers, and the omission is by design. However, the theory of functional languages offers a lot of latent synergy for visual programming.

Thirdly, the separation of signal rates should be re-examined. Manual partition of algorithms into distinct update schemes often feels like a premature manual optimization. If this optimization could be delegated to the compiler, the *ugen* vocabulary could conceivably be further reduced by unifying all the clock regimens. The solutions to these open problems are subsequently enumerated as three main topics that this study addresses.

1. **Unified Signal Model**

   The multirate problem is resolved from user perspective by applying unified semantics for all kinds of signals, ranging from user interface events to midi messages as well as control and

audio signals. The technical solution is a compiler capable of producing the typical multirate optimizations automatically, without user guidance.

2. **Composable and Abstractive Ugens**

   The target language offers features that are a superset of a typical *ugen* interpreter. Similar programming models are available, but in addition, the focus is on *ugen* composition rather than a large *ugen* library. Algorithmic routing is provided for increased programmer productivity.

3. **Visual Programming**

   The language is likely more readily adapted by domain experts if a visual programming surface is available. The language should have a syntax that is minimal enough for successful visualization, and the program semantics should be naturally clear in visual form. Nevertheless, expressive abstraction should be supported.

The theoretical and practical methods for addressing these problems are discussed in Chapter 2, *Methodology*.

1.4 ABOUT THE KRONOS PROJECT

The rest of this chapter gives an overview of the activities undertaken during the Kronos project, related publications, the software package and the author's contribution to these.

The result of this project is a portfolio that includes the Kronos Compiler software suite for Windows and Mac OS X operating systems; this report, including six peer-reviewed articles; and supporting appendices that demonstrate aspects of the project via examples and learning materials.

The Kronos Compiler is programmed in C++ [11] and built on the LLVM [27] open source compiler infrastructure. The software architecture consists of the following modules:

1. Parser
2. Code repository
3. Syntax graph representation
4. Syntax graph transformation
5. Reactive analysis and factorization
6. Idiom translator from functional to imperative
7. LLVM IR emitter
8. LLVM compiler

Items 1–7 are exclusively developed by the author of this report. Modules 4–6 represent the central contributions of this study to the field, as detailed in Chapter 2. Item 8 is a large scale open source development, headed by Lattner et al [27].

1.4.1 Academic Activities

An extensive publishing effort has been a part of the Kronos project. 3 journal articles and 12 conference papers have been published in the extended context of study. A number of these are collaborations with Mikael Laurson and Mika Kuuskankare. The author of this report is the first author of one journal article and 10 conference articles. The publications are listed below:

International Scientific Journals

1. Mikael Laurson, Vesa Norilo, and Mika Kuuskankare. PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control. *Computer Music Journal*, 29(3):29–41, 2005

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

3. Vesa Norilo. Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. *Computer Music Journal*, 39(4), 2015

International Conference Articles

1. Vesa Norilo and Mikael Laurson. A Unified Model for Audio and Control Signals in PWGLSynth. In *Proceedings of the International Computer Music Conference*, Belfast, 2008

2. Vesa Norilo and Mikael Laurson. Kronos - a vectorizing compiler for music dsp. In *Proc. Digital Audio Effects (DAFx-10)*, pages 180–183, Lago di Como, 2009

3. Vesa Norilo and Mikael Laurson. A method of generic programming for high performance {DSP}. In *Proc. Digital Audio Effects (DAFx-10)*, pages 65–68, Graz, 2010

4. Vesa Norilo. Designing Synthetic Reverberators in Kronos. In *Proceedings of the International Computer Music Conference*, pages 96–99, Huddersfield, 2011

5. Vesa Norilo. A Grammar for Analyzing and Optimizing Audio Graphs. In Geoffroy Peeters, editor, *Proceedings of International Conference on Digital Audio Effects*, number 1, pages 217–220, Paris, 2011. IRCAM

6. Vesa Norilo. Introducing Kronos - A Novel Approach to Signal Processing Languages. In Frank Neumann and Victor Lazzarini, editors, *Proceedings of the Linux Audio Conference*, pages 9–16, Maynooth, 2011. NUIM

7. V Norilo. Visualization of Signals and Algorithms in Kronos. In *Proceedings of the International Conference on Digital . . .* , pages 15–18, York, 2012

8. Vesa Norilo. Kronos as a Visual Development Tool for Mobile Applications. In *Proceedings of the International Computer Music Conference*, pages 144–147, Ljubljana, 2012

9. Mika Kuuskankare and Vesa Norilo. Rhythm reading exercises with PWGL. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8095 LNCS, pages 165–177, Cyprus, 2013

10. Digital Audio Effects. Kronos Vst – the Programmable Effect Plugin. In *Proceedings of the International Conference on Digital Audio Effects*, Maynooth, 2013

11. Josue Moreno and Vesa Norilo. A Type-based Approach to Generative Parameter Mapping. In *Proceedings of the International Computer Music Conference*, pages 467–470, Perth, 2013

12. Vesa Norilo. Recent Developments in the Kronos Programming Language. In *Proceedings of the International Computer Music Conference*, Perth, 2013

Academic Presentations

Aspects of this study have been the presented by the author in various academic contexts. These presentations are listed below.

CONFERENCE TALKS

1. 2009, Talk at International Computer Music Conference, Belfast
2. 2011, Invited speaker at Linux Audio Conference, Maynooth
3. 2012, Talk at International Computer Music Conference, Ljubljana
4. 2013, Talk at International Computer Music Conference, Perth
5. 2014, Talk at International Conference on Digital Audio Effects, Maynooth

CONFERENCE POSTER PRESENTATIONS

1. 2010, International Conference on Digital Audio Effects, Graz
2. 2011, International Conference on Digital Audio Effects, Paris
3. 2011, International Computer Music Conference, Huddersfield
4. 2012, International Conference on Digital Audio Effects, York

OTHER TALKS

1. 2011, Colloquium at IRCAM, Paris
2. 2012, PRISMA meeting, Arc et Senans
3. 2013, Colloquium at CCRMA, Stanford University
4. 2015, Workshop at National University of Ireland, Maynooth

1.4.2 Contents of This Report

The scope of this report is the design, implementation and applications of the Kronos Compiler. It comprises of three parts: Part I, this introductory essay. Part II, the peer reviewed publications, which constitute the majority of this work. Part III, appendices, where the principles put forward in this essay and the publications are elaborated less rigorously, supported by examples. The Introductory essay refers to the publications and appendices in order to better define or explain a concept. A summary of the peer reviewed publications is given in Section II.

The structure of this essay is as follows. This chapter, *Background*, defined the research problem, motivated the study and provided an overview of the project and the related activities. Chapter 2, *Methodology*, explains the methodology of the study. The chapter is divided in two parts, theory

and implementation. The *Theory*, in Section 2.1, summarizes and collects the theoretical framework this study is based on as well as the novel inventions. The *Implementation*, in Section 2.2, deals with the engineering aspect of writing a compiler, discussing implementation strategies that are too particular to the software in this portfolio to be otherwise published. This section is key for readers who are interested in looking at the source code of the Kronos Compiler. The results of this study in relation to the state of art are discussed in Chapter 3, *Discussion*, followed by the *Conclusion* of this report in Chapter 4.

2 METHODOLOGY

2.1 THEORY

The theoretical framework of the Kronos language and compiler are discussed in this section. Research problems relevant to furthering the state of art in musical signal processing ar identified, and paradigms from the field of general computer science are proposed in order to solve them.

An overview of the three main problems addressed by this study are summarized in Table 1. Firstly, the distinction between audio and control rate, events and signal streams, should be replaced by a **Unified signal model**. Secondly, the requisite vocabulary of unit generator languages should be reduced, replaced by adaptable and **Composable ugens**. Thirdly, the language must be adaptable for **Visual programming**, as it is preferred by many domain experts. All of these should be attainable with high performance real time characteristics. This translates to the generated code executing in deterministic time, reasonably close to the theoretical machine limit.

The main contributions of this study are presented in Sections 2.1.4 and 2.1.5, discussing the unique type system approach, *Simple* F_ω, and the application of *Reactive Factorization* to solve the multirate problem by the application of theory of reactive systems. For an example-driven look at the compiler pipeline, please refer to Appendix C.

2.1.1 Functional Programming

Functional programming is a programming paradigm based on the ideas of lambda calculus [36] [37]. A key feature of this paradigm is the immutability of variables. In other words, variables are constant for the entirety of their lifetime. In addition, functions are treated as first class values, so they can be constructed *ad hoc*, stored in variables and passed to other functions.

Two characteristics of functional programming stand out to make it eminently suitable for a signal processing domain language. Kronos is designed to be applied in the context of visual programming: in this domain, *data flow* is naturally represented. The functional paradigm exhibits data flow programming in its pure form. Secondly, high performance is required of a signal processing system, as discussed in Section 1.3. In a language designed for non-professional programmers, automated rather than manual code optimization is more feasible. Functional programming provides a strong theoretical framework for implementing optimizing compilers. These two aspects are subsequently elaborated.

Table 1: Research Problems and Solutions

Problem	Proposed solution
Unified signal model	Discrete reactive systems
Composable ugens	Functional, generic
Visual programming	Functional, data flow

Data Flow and Visuality

Functional programs focus on the data flow; the composition of functions. Much of the composition apparatus is exposed to the programmer, as functions are *first class values*. This means that programs can assign functions to variables, pass them as parameters to other functions, combine and apply them. New functions can be constructed *ad hoc*.

The difference between functional and the more widely used imperative idiom is best demonstrated via examples. Consider the Listing 1. It shows a routine in C++, written in a typical imperative fashion. First, variables are declared. Then, a loop body is iterated until a terminating condition occurs. Inside the loop, variables from the enclosing scope are mutated to accomplish the final result, which is returned with an explicit control transfer keyword, `return`.

A counterexample is given in Clojure, written without variables or assignment and shown in Listing 2. Instead, the iterative behavior is modeled by a recursive function. This example is for demonstration purposes: it doesn't reflect the best practices due to not being tail recursive.

Listing 1: Function to compute the sum of an array in C++

```
int sum_vector(const std::vector<int>& values) {
  int i = 0, sum = 0;
  for(i; i < values.size(); ++i) {
    sum += values[i];
  }
  return sum;
}
```

Listing 2: Function to compute the sum of an array in Clojure

```
(defn sum-array [values]
  (if (empty? values) 0
    (+ (first values)
       (sum-array (rest values)))))
```

Contrasting the visual depiction of the abstract syntax trees for the algorithms above is instructive. The imperative version, shown in Figure 1, is actually harder to follow when translated from textual to visual form. The visualization de-emphasizes the critically important *chronological* sequence of instructions.

The functional version, shown in Figure 2, exhibits the data flow of the algorithm. The algorithm is stateless and timeless, *topological* rather than *chronological*. The graph captures everything essential about the algorithm well.

The difficulty in understanding the imperative syntax graph results from implicit data flows. Some syntax nodes such as assignment could mutate state upstream from them, implicitly affecting their sibling nodes. This makes the ordering of siblings significant, as the sequence of state mutation defines the behavior of the program. With immutable data, the processing order of sibling nodes is never significant.

Imperative programs resemble recipes or the rules of a board game: they are formulated as a sequence of instructions and flow control. Functional programs are like mathematical formulas or maps: stateless descriptions of how things will happen. This is why graphical syntax trees and visual programming are well suited to represent them.

Please refer to P3 for a discussion on functional replacements for imperative programming staples.

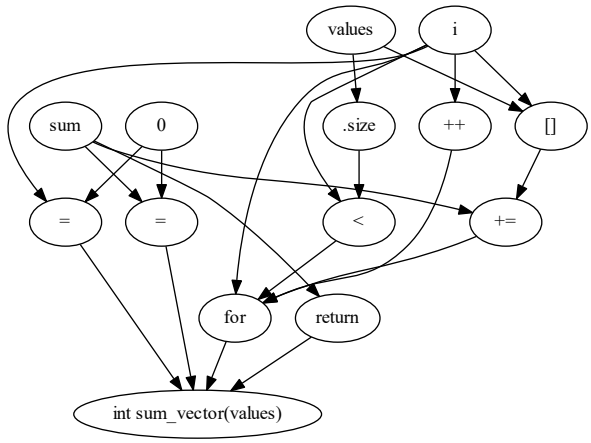

Figure 1: Abstract syntax tree of a C + + function

Efficient High Level Code

High performance is one of the key requisites in a signal processing environment, as discussed in Section 1.3. There are two main methods of designing an efficient language. The language can reach down, "to the metal", to let programmers influence the exact composition of machine instructions that comprise the final program code. Expert programmers can craft routines that are tightly matched with the target machine architecture.

On the other hand, a language can be designed for optimizing compilers. In this case, the compiler deploys algorithms that translate the user code to an efficient machine representation – automating the work of an expert programmer.

The latter option makes sense in a language aimed towards domain experts such as musicians. The Kronos language is designed with very restrictive semantics and a simple memory model. These characteristics support the implementation of an optimizing compiler, as more assumptions can be made about the semantics of the code.

As a result, the machine code emitted by the Kronos compiler may be quite different from the program source. Much of the apparent high level abstraction and data propagation can be elided away. The functional programming model provides the following optimization advantages [36]:

1. **Lambda calculus**, the foundational principle of functional programming, enables mathematical manipulation and reasoning about the syntax tree. Proofs and isomorphic transformations that increase efficiency are easier to come by.

2. **Referential transparency** ensures that expressions can always be replaced with their values without changing program behavior. While trivially true in lambda calculus, this property is hard to prove for imperative program fragments.

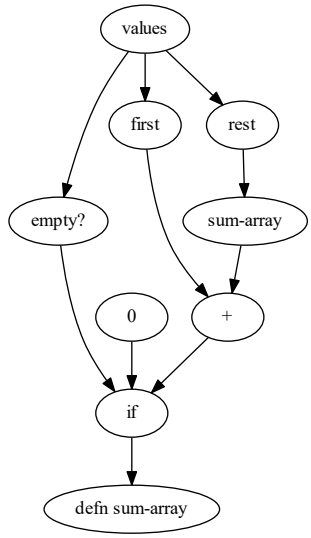

Figure 2: Abstract syntax tree of a Clojure function

3. **Immutability** of data guarantees that the compiler is free to use value or reference semantics at will, which can be utilized to avoid copying memory. Implicit data flows do not exist, making program analysis easier.

There is an inherent risk of a leaky abstraction in a design like the one described above. Leaky abstraction means that the programmer must still know the efficient low level instruction composition, and tweak the source program in order to achieve that composition through the optimization pipeline. This risk is reduced in a language that does not aim for general purpose programming, as it can be designed to exclude source forms that interact badly with the optimizer.

2.1.2 Reactive Systems

According to Van Roy's definition [38], a discrete reactive system is one that responds to input events by producing output events, in such a way that the sequence and timing of the input events uniquely determines the sequence and timing of the output events.

Discrete sequences of events can be viewed as time–value pairs, which constitute a discrete time series. Pure functions [39] operating on such serii can be seen as a trivial subset of discrete reactive systems. The output of a pure function can only change when one or several inputs change. Therefore, the output sequence contains, at most, one event for each input event, synchronously in time.

Both sample streams and event queues can be modeled using time–value pairs, although in the traditional interpreted context, dispatching each audio sample as a timestamped event would likely impose crippling computational overhead. The Kronos language is designed around this

unified signal model, and the reactive factorization technology, described in detail in Section 2.1.5, is designed to eliminate dispatch overhead in the case of high frequency sampled streams.

In addition to pure functions, Kronos offers operators that represent the state of the signal processor. These include unit delays and ring buffers, which are built from more general state arrays with read and write access. Such operators are not pure, as they have an implicit memory. However, the language semantics maintain functional purity from the perspective of the programmer; any mutation of state is only permitted after all its other uses are completed. This allows for referential transparency for the stateful operators, as the state they refer to remains immutable during all computation, to be assigned to only afterwards. A similar method of describing stateful operations in a functional context was adopted earlier in the Faust language [9].

These semantics maintain the discrete reactive model, and the total system output remains a pure function of current and *past inputs*. The implicit state is automatically reified by the compiler in strict accordance with the semantics previously defined. This technique is related to the concept of code weaving in aspect oriented programming [40].

Finally, operators that explicitly control the reactive clock propagation are offered. It is possible to impose the update rate of one signal onto another, to conditionally inhibit clock propagation and to specify clock priorities to automatically make certain signals dominate others. For a summary of the reactive operators currently in the language, please refer to Section A.1.7.

This operator set is sufficient to cover a wide range of signal processors ranging from artificial reverb to voice allocating synthesizers and MIDI filters, all with an efficient, unified signal model, corresponding to the principle of discrete reactive systems. Example applications are shown primarily in P3 and P4.

2.1.3 Generics and Metaprogramming

Type systems are key to generating efficient code. In addition, they enable programming errors to be caught early. However, to beginners, type notation in the source code can seem confusing and redundant.

The Kronos language is designed to require no type notation in the source form. However, the executed program is fully statically typed, to ensure fast, deterministic execution. To accomplish this, a technique inspired by the C++ template metacompiler [41] is utilized. The source program is typeless or *generic*. Upon application, a typed or *specialized* version is generated based on some root types. In the context of signal processing, these root types are the external inputs to the system.

In contrast to more sophisticated automated type systems, such as those in the broad ML family [42], Kronos opts for forward type derivation instead of inference. A similar mechanism is used for the C++11 `auto` keyword [11]. Such derivation is very simple, and never requires any manual type annotations, always resulting in unamibigous static typing. It doesn't scale well for a large scale general purpose language, as the type derivation essentially represents a separate, dynamically typed computation pass over the entire program. Type-specialized functions are generated or *reified* at this time, and the number of distinct reifications can grow quickly. However, these drawbacks matter less in the context of signal processing kernels, and offer great benefits for code optimization.

Ad-hoc Polymorphism and Pattern Matching

One of the fundamental contributions of this study to the field is the furthering of *ugen parametrization*. This idea can be traced back to SuperCollider [8], which offers a channel expansion facility: when a *ugen* receives vectors of parameters, it becomes a vectorized *ugen*. In terms of programming language theory, this can be seen as a form of *polymorphism*; the exact behavior of a *ugen* is dependant on the type of its parameters.

Kronos aims to generalize and further promote such type parametrization. Polymorphic functions serve the role of *ugens*. The Kronos type system supports data types that map to the elementary hardware types, such as 32- and 64-bit floating point and integer scalars. In addition, packed vectors are supported. Constructs such as strings, nil and numeric constants are provided as entities lifted into the type system for metaprogramming purposes. As such, the strings `"Hello"` and `"Hi"` are distinct types. They never constitute a part of a signal flow, but can influence type derivation and *ugen* reification. Tags with unique identity are provided as well.

Algebraic types can be composed of the aforementioned primitive types with the typical composition operators `Pair`, `First` and `Rest`. The parser provides syntactic sugar for structuring and destructuring types from chains of pairs (see A.1.6). In addition, a *nominal* type can be declared by assigning a unique tag to it. The tag can be used to provide semantic information, such as distinguishing between a complex number and a stereophonic sample, both of which are structurally identical.

Polymorphic unit generators can be written for classes of argument types. Type constraints are derived from the destructuring that a function performs, including nominal type tags. In addition, functions inherit constraints from any functions they themselves call. Overload resolution is performed during type derivation, by accepting the first constraint-satisfying form of each function in reverse source order. Polymorphic *ugens* can operate on types that share semantics rather than structure. This is known as *ad-hoc polymorphism* according to Cardelli and Wegner [43].

Ad-hoc polymorphism makes it possible to write a *ugen* that has a different behavior depending on the structural and nominal properties of the argument type. *Ugens* can also choose to ignore types entirely, and rely on their constituent components to maintain correct semantics regardless of type. As an example, most elementary filters only require summation and multiplication semantics to be defined for the type of the signal in order to behave correctly. Such type-agnostic *ugen* implementations are known as *generic ugens*.

It is noteworthy that the inheritance of type constraints from callees enables one to design polymorphic functions without type annotations. However, an explicit type annotation or a homomorphic function is always present at some levels in the call graph. In most cases, these are derived from primitive operators or accessor functions that destructure a nominal type.

2.1.4 Simple F_ω

System F_ω is a concept from Barendregt's [44] lambda cube, which describes lambda calculi with different capabilities. All calculi feature terms that depend on terms – essentially ordinary functions such as those in simple lambda calculus.

The addition of polymorphic functions yields System F, the second order lambda calculus. This system adds terms that depend on types, giving functions the capability of adapt to the argument type. The addition of type operators yields System F_ω. Functions in F_ω can compute on terms as well as types.

F_ω describes the Kronos type system. The notable omission is types depending on terms, or dependent types, from System λ_ω. In practical terms, this means that the result type of a function can not depend on the runtime values of signals.

Since the flow control mechanisms in Kronos are based on type constraints and polymorphism, System F_ω results in a deterministic data- and control flow. This has important implications for both the language semantics and the compiler implementation. Firstly, it renders the computing layer of the language less expressive than the type system itself. Secondly, the determinism allows significant program optimization. The gist of the design is an expressive metaprogramming layer – via types – capable of producing highly optimized static signal processors.

The Kronos type system recognizes no built in collections of any kind. Instead, vectors, lists and maps can be constructed algebraically from pairs. Together with the choice of type derivation instead of inference, the typing pass becomes an algorithmically unremarkable global trace of the program signal flow. However, as an compilation speed optimization, many recursive signal flows are analyzed in closed form. This is an essential feature that enables the type derivation to scale over operations on large collections and deep recursions. The analysis algorithm is presented in [P6].

This simple type derivation system together with the properties of the F_ω calculus is referred to as Simple F_ω, and is to the knowledge of the author unique to this study. Indeed, the benefits only become apparent when combined with a suitable optimizing compiler, applied to a fairly narrow problem domain with limited problem sizes, such as musical signal processing.

Given the deterministic data flow, this system features trivial and deterministic memory allocation behavior in all valid programs. All return values are implemented as side effects on caller-allocated memory. The allocations and side effects can propagate multiple levels in the call hierarchy. This results in very extensive copy elision, and is an important factor in making recursive functions as efficient as possible. For an example, please refer to Section C.5.

2.1.5 Reactive Factorization

The deterministic data and program flow, as described in Section 2.1.4, makes full machine analysis of data dependencies possible for arbitrary program fragments. The Kronos compiler features a global data flow tracer that identifies dependencies between all the inputs to the program and all the syntactic nodes in it, according to the reactive semantics described in Section 2.1.2.

To enable multirate processing, the compiler can insert state memory at the syntactic nodes where update rate boundaries occur. Such state corresponds to the object oriented practice of using member variables that cache intermediate values, but in Kronos they are generated automatically by the compiler.

The data flow analysis further enables the machine code emitter stage to filter machine instructions according to reactivity. For example, an update routine can be emitted for a certain input to the program, filtering out all the machine instructions that do not yield new output upon the input event. Combined with the dead code optimization in the LLVM [27] pipeline, this produces highly efficient update routines.

This system enables automatic factorization of a data flow program for each of its distinct external inputs. The implicit state at signal rate boundaries provides interaction between clock sources, without the user having to manually factor the program source into sections operating at different update rates. It is notable that such automated factorization becomes impossible in the general case if the underlying calculus is permitted to become more expressive than the Simple F_ω, as deterministic data flow is a requirement.

2.2 IMPLEMENTATION

This section discusses the implementation of the Kronos compiler system in the C++ programming language [11]. It is intended as a high level technical document that should be the first step in understanding the source code.

The compiler components are discussed in program life cycle order, starting from Section 2.2.2, *Parser*, discussing the internal representation of programs in Section 2.2.3 and eventual machine code generation in Section 2.2.5. Appendix C elaborates this process via examples.

Relevant type names from the compiler source code are shown boxed in each section. The namespace for the public API is `Kronos`, while the namespace used for the internal implementation is `K3`.

2.2.1 Application Programming Interface

The Kronos API is provided via a private implementation pattern. The client-facing header file specifies abstract interface classes that implement either value or shared pointer reference semantics, and completely inlined wrapper classes that contain the interfaces as private implementation pointers. This technique provides a degree of potential cross-compiler binary compatibility between the library and the client. An overview of the client API is shown in Table 2.

Value and Shared Semantics

All public API classes in the `Kronos` namespace exhibit either value or reference semantics. The classes indicated by "value" semantics in Table 2 can be copied and passed like regular C++ values. The classes with "reference" semantics are smart pointers to an internal implementation. Copied instances of these classes refer to the same underlying object, which is released when the last reference is dropped. Finally, classes with "unique reference" semantics, all of them exception types, appear as C++ constant references to abstract classes: it is not possible to construct these instances.

Kronos Context

`K3::TLS`

`Kronos::Context`

Kronos context represents the state of the compiler and its code repository. Any usage of the compiler is achieved through the context object. The context provides full isolation of all the compiler internals, so multiple independent compilation contexts can exist within the same process. Internally, the context object holds thread local storage for the compiler state, and all the imported source code in parsed form.

2.2.2 Source Language and Units

`namespace K3::Parser`

The Kronos source language is a specification for representing programs in textual form. A comprehensive overview of the language structure is given in Appendix A. The parser is implemented as a simple single pass recursive tokenizer, which builds a package of subpackages, functions and the abstract syntax trees that define them. The resulting package can be sent to the code repository in the generic graph form – see Section 2.2.3.

The source code is structured in units of text with an identifier. The units most commonly correspond to files on disk, where the file path serves as an identifier. Code units can also be provided via the network as a remote resource, where the sender provides an arbitrary identification token for the unit.

The code repository of the current context is defined by a sequence of unit imports. Because source order is significant in overload resolution, the unit order is important. That is why the compiler tracks the units in the repository, and when a unit import is performed with a identification

Table 2: Kronos API classes

typename	semantics	description
InternalError	unique ref	exception: compiler bug detected
RuntimeError	unqiue ref	exception: run time error
SyntaxError	unique ref	exception: syntax error in user code
TypeError	unqiue ref	exception: type error in user code
Type	value	a Kronos type
UserException	unique ref	uncaught user exception
Trigger	value	Compiled instance update callback
Var	value	Compiled instance variable
Instance	shared ref	Compiled instance state
Class	shared ref	Compiled code, constructs instances
Context	shared ref	A compilation context

already in the repository, the changes are applied to the pre-existing point in the import sequence rather than appending it. In effect, the repository is rolled back to the state prior to the changed unit, and rebuilt from that point. The repository maintains a version history of the unit import sequence, each version being stored as a patch on the previous one.

Code units can declare dependencies on other units. Currently, only source code files local to the compile server can be pulled in via explicit dependencies. The unit system maintains a source order where dependencies always precede the unit that depends on them. The exception is circular dependency: if a dependency chain reaches a unit multiple times, only the first dependency is honored, the others ignored. This prevents an infinite recursive unit import chain.

2.2.3 Internal Representation of Programs

The compiler is founded on the concept of graphs and graph transformations. Kronos graphs are lightweight, disposable and immutable. The transforms share the basic mechanism of preserving graph topologies in the case of diamond and cycle shapes. This is accomplished by maintaining source to destination maps for nodes that may be encountered multiple times during a transform pass.

The transformation of a graph node would typically involve calling a node-specific transformation function and passing the transform object to it. Most routines would then use the object to process transformed version of any upstream dependencies and recombine them according them to the node specific logic.

Graph nodes are mutated during graph creation, and immutable afterwards. This affords the pooling of resource allocation on a per-graph basis. Nodes are created with a region allocator [45] associated with the current graph transform. The graph flow is built with unidirectional weak references from downstream to upstream. Since no data is held about any downstream connections, subgraphs can freely share structure. Most graph transforms only require upstream connections to operate. In the remaining cases, per-graph maps are constructed that serve as reverse upstream lookup. Graph objects hold strong references to memory pools that contain the nodes in them.

Many node objects do not require explicit destruction. The nodes that contain non-trivial members or those that require deinitalization must be derived from one of the `Disposable` node types in order to ensure that their destructors are called upon release. The debug build of the compiler contains runtime checks for non-trivial destructors not declared as disposable.

Type

`Kronos::Type`

`K3::Type`

These objects describe the Kronos type notation. No signal data is ever associated with a type object, but it can be used in conjunction with a binary blob to parse or print the blob as text. Without a binary blob, the type can print itself as human readable text. Type objects are also used to describe root level arguments or external inputs to a Kronos program.

This object exposes an API that corresponds to the semantics of the Kronos type system. Internally, homogenic tuples are run length encoded, such that common sequences of a single type are efficiently represented.

Generic Graph

`K3::Nodes::Generic`

The generic graph is the internal representation of a Kronos program that is most closely related to the source code. Each node in the graph corresponds to a construct in the program. Functions are polymorphic and symbols refer to a particular code repository. The generic graph is untyped and not executable.

Typed Graph

`K3::Nodes::Typed`

A typed graph is constructed from a generic graph via a specialization transform pass. This is a reification of the generic program. Specialization starts from a generic graph with an optional argument type. Forward type derivation is performed for the entire graph, with each polymorphic function being resolved to a typed, monomorphic form.

If the program is free of type errors, a reified, typed graph is generated. This graph has full type semantics and corresponds to concrete mathematical operations and the actual data flow of the program.

Further transform passes are used to build a machine code representation of the typed graph via the LLVM code generator.

2.2.4 Compilation Transform Passes

Identity

`K3::Transform::Identity<T>`

Identity transform is the simplest of all graph transforms. The standard identity transform routine for a node makes a shallow copy of the node, and proceeds to replace its upstream connections with recursive transformations. Since Kronos graphs are immutable and share structure, copying them is usually not necessary. The identity transform is mostly useful as a base mechanism for other transforms.

Symbol Resolution

`K3::SymbolResolution`

This transform parses absolute and relative symbol names in a generic graph and resolves them to specific entities in the code repository. The result is a generic graph, where symbols are replaced with a direct reference to the expression they index.

The transform fails if a symbol can not be resolved.

Specialization

`K3::Nodes::SpecializationTransform`

The specialization transform is parametrized by function argument type. It accepts a generic graph and results in a typed graph and a return type.

Any polymorphic function calls are recursively specialized. Generic recurring functions may be compacted into typed function sequences if suitable argument evolution is detected, as explained in P6.

The specialization transform may fail if the program contains a type error. The type error may be hard or soft; the soft `SpecializationFailure` causes a function form to be rejected in overload resolution and the next form to be tried. Any other errors propagate back towards the transform root until the entire transform fails or a suitable exception handler is found.

The specialization transform handles graph cycles by special treatment of delay nodes, which are the only nodes for which the parser generates cycles. These nodes may return an incomplete typed node that will be finalized once the transform is otherwise ready. The deferred processing is accomplished by a per-transform post processing queue. Each delay node adds a post processing step to the transform that finalizes the incomplete cycle. If a nested cycle is encountered, further post processing steps may be added dynamically.

Please see Sections C.2–C.3 for a concrete example of the specialization transform.

Sequence Recognition

To specialize deeply recursive functions in constant time, the specialization transform attempts to recognize recursion and reason about it in closed form. While type inference schemes [42] are inherently capable of this, the simpler type derivation scheme must conceivably trace the entire data flow of the program, since polymorphic ad-hoc recursion could well result in different overload resolutions for each iteration. To prevent compilation times from escalating with recursion depth, Kronos implements a type evolution analysis algorithm for recursive functions. This algorithm is one of the main topics of P6.

The analysis is performed by wrapping the argument type in a special, externally invisible rule generator type. The specialization then proceeds normally. The rule generator records any inquiries about the underlying type and the responses, such as whether the type is an algebraic compound or an integer, or if the `First` of the pair is a floating point number. These inquiries become type rules, which the rule generator *lifts* from the user program. In aggregate, the rules determine the overload resolution behavior.

As a second step, argument evolution in the recursion is examined. The evolution analyzer is capable of detecting recursive iteration such as taking the `Rest` of a list, once or several times, as the sole recursive argument or a portion thereof, plain or as a component of a more complex type. In addition, linearly increasing or decreasing invariant constants are included in the evolution analysis. A recursive argument that consists of invariant types and successfully detected evolutions can be converted to closed form: argument type is a function of the loop induction variable I.

Each rule of the overload resolution rule set is then parametrized by the induction variable. The rule set becomes a group inequality describing the range of I for which the rules remain satisfied.

The iterations of the recursion within that range are thus guaranteed to resolve into the same overload.

For cases that are not tail recursive, the overload resolution may depend on the function return type as well. In these cases the evolution analysis is performed first for the argument – from sequence start to end – and then for the return value, in reverse.

The benefit of sequence recognition is related to the simple type derivation scheme: without the evolution analysis, specialization of deep recursions would be a linear time operation in proportion to the recursion depth – in the case of a collection comprehension operation, to the size of the collection. A successful sequence recognition can specialize such a sequence in constant time, regardless of the depth. Please refer to Section C.3 for an example on sequence recognition.

Code Motion

`K3::Backends::CodeMotionAnalysis` `K3::Backends::CodeMotionPass`

This is an optimization pass for the typed graph that should be run before reactive analysis. It traverses the program graph looking for equivalence classes of subgraphs. If a sufficient number of equivalent expressions is found in the program, they are replaced by a dynamically scoped global variable that is materialized at the outermost scope shared by the occurrences. Dead (unused) function arguments may result from running this pass.

This pass is useful in reducing the amount of state generated by the compiler and hoisting invariant computations out of loops.

Reactive Analysis

`K3::Reactive::Analysis`

This transform accepts a typed graph and produces a typed graph with associated reactivity information. The result graph contains clock source annotation for every typed node, as well as newly inserted boundary nodes at points where state memory is needed to preserve a signal at a clock region boundary.

The reactive analysis handles recursion by reporting an incomplete clock source for recursive connections. This incomplete clock source defers to any other clock source it encounters, while keeping a record of them. When the recursive loop is complete, all the clock sources seen by the loop are resolved and the incomplete clock is replaced by the result.

Please refer to Section C.4 for an example of reactiv analysis and factorization.

Side Effect Translation

`K3::Backends::SideEffectCompiler`

A transform pass that supports the final compiler pass. This pass reconstructs the functional data flow with memory effects and ordering primitives. Signal structuring and destructuring is replaced with pointer arithmetic. The resulting typed graph is no longer purely functional. Cycles are broken and state- and temporary memory buffers are allocated. The output of this pass is intended to be easily lowered to imperative idioms like machine code. Clock source annotations from Reactive Analysis are preserved and relayed to any generated side effects. This allows reactive factorization to filter the side effects as well.

Data flows are constructed bidirectionally. Arguments flow from upstream to downstream in the graph. The basic destructuring operator `First` is a no-op in pointer arithmetic; `Rest` is replaced with pointer arithmetic along with nodes that measure the size of the `First`. Pointer dereferencing

is added on demand, such as for the inputs of a simple mathematical operation that will map onto a single machine instruction.

Return values are implemented by additional pointer parameters. The return value pointer bundle is propagated upstream from the function root. If the return value is an algebraic composite, structuring is handled by inverting the data flow: propagating the side effect up through a `Pair` node is like destructuring the side effect. The left hand side of the node receives the side effect pointer as is, while the right hand side is adjusted accordingly. The side effect propagation stops at nodes like arithmetic and function calls. Memory writes are inserted at elementary operations, while at function call nodes, the side effect pointer is passed as the root return value pointer to the callee. In more complicated cases, where the return value of the callee is destructured and structured, bundles of pointers can be passed as well. This means that the return value can be discontiguous in memory, as required by full copy elision for return values.

Such a comprehensive copy elision scheme is possible due to the fully analyzed static data flow. This often enables the transformation of a recursive function that performs destructuring and structuring into a tail recursive form, which subsequently becomes a simple loop in the optimizer pipeline. For an example, please refer to Section C.5.

2.2.5 LLVM Code Generation

LLVM Module

`K3::Backends::LLVMModule`

`LLVMModule` is responsible for the glue code that enables calling the public API of the generated code. The module connects external inputs and triggers to callback and data slots inside an instance of compiled code.

The most significant implementation detail in the LLVMModule is the vectorization of audio processing. Kronos offers deterministic downsampling and upsampling operators that result in a predictable dispatch pattern between frames that require control signal updates and those that do not. The module code examines the clock rates used by the client code and determines a suitable vectorization stride, such that control frames and audio frames can be interleaved without branches. The module further emits prealignment and remainder code to process arbitrary numbers of samples at once, maintaining a subphase counter relative to the determinstic dispatch pattern fragment. The dispatch pattern is discussed further in P6.

Class

`Kronos::Class`

The client facing `Class` encompasses a compilation unit for which LLVM intermediate representation has been generated and optimized. The class can be queried for a list of instance variables of type `Var` and triggers of type `Trigger`. The class can construct signal processor instances, either by allocating a state blob, or in place with client provided memory. Instance construction compiles the attached LLVM module to machine code, and associates it with a LLVM execution engine.

Alternatively, Classes support static ahead of time compilation. Instance construction and static compilation are exclusive options; either action will make the other option illegal.

The `Class` object manages memory via shared reference semantics. Copies of the object refer to the same class, and resources are freed once no copies exist. The actual code and symbol table are held by this object.

Instance

`Instance` is a state blob associated with a compiled signal processor. Variables and callbacks can be obtained from an instance. The variables can be mutated to supply external data to the instance. The callbacks mutate the state blob and produce output. An instance object retains the class it was constructed from. Instances manage memory via shared reference semantics. Copies of an instance object refer to the same state blob, which will be retained until no copies exist.

3 DISCUSSION

3.1 THE IMPACT OF THE STUDY

This section discusses the results of this study and their relative significance. The attainment of the objectives detailed in Section 1.3.1 is addressed via a supplementary example, shown in the subsequent Section 3.1.1. This example attempts to distill the essential contributions of the Kronos project to a small scale example. To better understand the source code shown, the reader may wish to consult Appendix B, *Tutorial* and Appendix A, *Language Reference*.

Alternate, hypothetical methods of attaining a similar signal processing environment are treated in Section 3.1.3.

3.1.1 Supplementary Example

The example consists of following principal components:

1. a **Generic Feedback Oscillator** defined to generate a waveform at audio rate from two parameters: an initial state, and an anonymous function that describes state evolution between sample frames.

2. a **Sinusoid Oscillator** defined in terms of the generic feedback oscillator and a closure over a complex multiplication coefficient.

3. an **Additive Synthesizer** that derives a set of sinusoids from a fundamental frequency and frequency step between harmonics. The synthesizer is defined as map operation utilizing the sinusoid oscillator as the mapping function.

The sinusoid synthesis method chosen here is based on a recursion derived from Euler's formula: it reduces to a unit-delay recursive complex multiplication. This method has the benefit of not depending on any transcendental functions in audio processing: all the synthesis code shown is directly generated by Kronos. Further, it allows a minimal yet useful demonstration of Kronos' abstractive capability as a specialization of the generic feedback oscillator. Finally, the algorithm provides a combination of good accuracy and computational efficiency – it is perfectly usable in practical signal processing.

The program in source form is given in Listing 3.

Listing 3: Supplementary example: additive synthesis

```
; the Algorithm library contains higher order
; functions like Expand, Map and Reduce

; the Complex library provides complex algebra

; the IO library provides parameter inputs

; the Closure library provides captures for
```

```
; lambdas

; the Math library provides the Pi constant

Import Algorithm
Import Complex
Import IO
Import Closure
Import Math
Import Implicit-Coerce

Generic-Oscillator(seed iterator-func) {
  ; oscillator output is initially 'seed',
  ; otherwise the output is computed by applying
  ; the iterator function to the previous output

  ; the audio clock rate is injected into the loop
  ; with 'Audio:Signal'

  out = z-1(seed iterator-func(Audio:Signal(out)))

  ; z-1 produces an unit delay on its right hand side
  ; argument: the left hand side is used for
  ; initialization

  ; Audio:Signal(sig) resamples 'sig' to audio rate

  Generic-Oscillator = out
}

Sinusoid-Oscillator(freq) {
  ; compute a complex feedback coefficient
  norm = Math:Pi / Rate-of(Audio:Clock)
  feedback-coef = Complex:Unitary(freq * norm)

  ; Complex:Unitary(w) returns a complex number
  ; with argument of 'w' and modulus of 1.

  ; initially, the complex waveform starts from
  ; phase 0
  initial = Complex:Unitary(0)

  ; Haskell-style section; an incomplete binary operator
  ; becomes an anonymous unary function, here closing over
  ; the feedback coefficient
  state-evolution = (* feedback-coef)

  ; the output of the oscillator is the real part of the
  ; complex sinusoid
  Sinusoid-Oscillator = Complex:Real(
    Generic-Oscillator(initial state-evolution))
}

Main() {
  ; receive user interface parameters
  fo = Control:Param("fo" 0)
  fdelta = Control:Param("fdelta" 0)

  ; number of oscillators; must be an invariant constant
  num-sines = #50

  ; generate the frequency spread
  freqs = Algorithm:Expand(num-sines (+ (fdelta + fo)) fo)

  ; apply oscillator algorithm to each frequency
  oscs = Algorithm:Map(Sinusoid-Oscillator freqs)
```

```
  ; sum all the oscillators and normalize
  sig = Algorithm:Reduce((+) oscs) / num-sines

  Main = sig
}
```

For demonstration purposes, the listing was compiled with the Kronos static compiler (see B.3.1), using the `-S` switch to emit symbolic assembly. The examples conform to the Intel syntax, which Kronos defaults to on the Windows platform.

The emitted code is the final result of all the compiler transform passes, including LLVM [27] optimization and code generation. Selected portions of the compiler output are subsequently exhibited and annotated.

Audio rate updates

The `Expand` call that computes the frequencies for all the oscillators is, as desired, absent from the audio path. The first section of the code deals with the sinusoid oscillators and their state. A fragment from the `Map` call is shown in Listing 4. The machine code exhibits four multiplications and two summations, which is the expected amount for complex multiplication. Both factors of the multiplication are read from a state memory buffer, accessed via the register `rax`, where they are laid out contiguously in a cache-friendly manner. The generated state memory results from the delay operation, `z-1`, and the signal rate boundary from control to audio.

Listing 4: Inner loop of `Map`, audio rate

```
.LBB5_3:
  vmovss  xmm1, dword ptr [rax - 20]
  vmovss  dword ptr [rsp + 4*r10], xmm1
  vmovss  xmm2, dword ptr [rax - 4]
  vmovss  xmm3, dword ptr [rax]
  vmulss  xmm4, xmm1, xmm3
  vmulss  xmm5, xmm2, dword ptr [rax - 16]
  vmulss  xmm1, xmm1, xmm2
  vaddss  xmm2, xmm4, xmm5
  vmovss  dword ptr [rax - 16], xmm2
  vmulss  xmm2, xmm3, xmm2
  vsubss  xmm1, xmm1, xmm2
  vmovss  dword ptr [rax - 20], xmm1
  inc r10
  add rax, 24
  cmp r10d, 47
  jne .LBB5_3
```

The `Reduce` call is minimalistic. The compiler has lowered a recursive fold by an anonymous function over a vector of floating point numbers into a completely unrolled loop, which consists of nothing but 49 `vaddss` instructions. An excerpt of the unrolled loop is shown in Listing 5.

Listing 5: Excerpt from `Reduce`, audio rate

```
...
  vaddss  xmm2, xmm2, dword ptr [rsp + 116]
  vaddss  xmm2, xmm2, dword ptr [rsp + 124]
  vaddss  xmm2, xmm2, dword ptr [rsp + 128]
  vaddss  xmm2, xmm2, dword ptr [rsp + 132]
  vaddss  xmm2, xmm2, dword ptr [rsp + 136]
  vaddss  xmm2, xmm2, dword ptr [rsp + 140]
  vaddss  xmm2, xmm2, dword ptr [rsp + 144]
...
```

The control section exhibits nothing but the `Expand` call to compute oscillator frequencies. The expensive transcendental functions required by the `Complex:Unitary` call are confined here, only occurring when the control parameters are updated.

In summary, the example demonstrates several aspects of the Kronos language and compiler, and the lowering of high level abstract code to static, highly efficient idioms:

1. **Abstract state memory** generated by the compiler, enabling the unit delay feedback path in the generic oscillator.

2. **Generic algorithms**, as the feedback oscillator algorithm has no knowledge of the kind of state or iteration function later applied to it.

3. **Ad-hoc Polymorphism**, as the multiplication uses complex number semantics based on the data type.

4. **Higher order functions**, as the mapping function applies a generic function previously defined by the user over a vector of values.

5. **Signal rate factorization**, as the computation of a complex feedback coefficient from a frequency value is automatically moved from the audio path to the control path, despite being deeply intertwined within the audio path. State memory is automatically inserted to enable multirate interaction.

In summary, the author believes that the combination of metaprogramming with reactive factorization and aggressive optimization yields a unique signal processing environment where abstraction that helps programmer productivity and comfort can be deployed with minimal efficiency cost. The extreme composability and flexibility of functions written in the Kronos language is demonstrated; the example contains a model of recursive composition in the form of a generic oscillator, a sinusoid oscillator defined in terms of this generic oscillator and a state evolution closure – and higher order functions that apply such a construct over vectors of values with minimal effort from the programmer.

It is to be noted that the utilized higher order functions `Expand`, `Map` and `Reduce` have no special support from the compiler: they are parsed from a textual source, freely available for the user to inspect and modify. Despite this deeply layered automation, the resulting machine code is quite similar to what a hand-written low level C program would produce.

The counterpoint to these advances is the strict isolation of runtime data from the program control flow; this ensures the determinism necessary for much of the automation in the compiler back end to take place. It's fairly clear that such a model would never be viable for a general purpose programming language. However, the author of this report believes that it represents an interesting and a genuinely new method for signal processor design and implementation.

Comparison to FAUST

FAUST by Orlarey et al. [9] is an earlier compiler research project for musical signal processing, whose results have greatly benefited this study. FAUST demonstrates the feasibility of the functional approach to signal processing and abstracting stateful memory as operators. A comparison of FAUST and Kronos is also given in `P1`.

FAUST is originally a monorate language. Initially, it was designed to have audio signals and parameters, the latter of which are updated once per audio buffer. While optimizations like loop invariant code motion can perform a small subset of the optimizations available in the Kronos reactive factorization, stateful operations such as delays are only possible within the audio path.

Recently, multirate extensions to FAUST have been investigated by Jouvelot and Orlaley [24]. The proposed vectorization and serialization semantics offer a subset of the Kronos deterministic multirate model – in Kronos, such primitives are constructed of ring buffers, decimators, upsamplers and wave table readers. Dynamic clock masking or processing of event-based streams is not available in FAUST as of this writing.

The principal advantage of FAUST is the relative maturity of the system. The compiler can build signal processors for various frameworks automatically [46]. Several large projects have been ported to FAUST, such as the Synthesis Toolkit [47, 48].

The FAUST language is arguably lower level than Kronos, lacking nominal typing and generic functions. A more limited type of polymorphism is available, implemented as pattern matching for arguments. FAUST supports anonymous functions, but at the time this report was written, no closures or captures. The main syntactic contribution of FAUST, the terse block diagram algebra, can be implemented as a domain language in Kronos, by using custom infix functions and closures. An indication of how such a domain lanuage could be designed is given in Section B.2.4.

An interesting avenue of future work could be to add automatic generation of glue code to enable Kronos signal processors to work with FAUST architecture files. In addition, transcompilation between the languages could be possible.

3.1.2 Comparison to Object Oriented Programming

Signal processors implemented in an imperative language tend to follow the basic principles of object orientation, as discussed by Lazzarini [20]. The imperative idiom is chronological, while the functional approach taken by Kronos is topological. The fundamental difference was elaborated in Section 2.1.1. Imperative programs provide an explicit sequence of commands, often moving the signal from one memory location to the next in great detail. By contrast, the functional approach describes the data flow, leaving implementation details to the compiler.

Kronos further pursues the functional approach for the musical signal processing domain. Several assumptions are made about how the data flow should work, especially regarding the processing of multirate signals.

In the case of imperative programming, the multirate problem of is a classic case of cross cutting concerns [49]. The problem of scheduling is interleaved with the actual signal processing. Imperative code must deal with the cross cut explicitly, transferring control to relevant signal processor objects at appropriate times. The scheduling code is difficult to detangle from the signal processing code for all but the most trivial cases.

The Kronos compiler automatically weaves the cross cutting aspects for the particular reactive signal model it supports and enforces. The reactive factorization, discussed in Section 2.1.5, extracts the appropriate code sequences for each external trigger, while adding implicit state to support multirate interaction.

This implicit state is often similar to the private members in an object oriented signal processor. The object would typically expose a high level interface. Consider a low pass filter; this interface could consist of a corner frequency and a quality factor. When these are adjusted, the filter object updates internal coefficients, which will be used in the actual signal processing routine, instead of the high level parameters.

This is a case of manual multirate factorization: since some computations in the filter equation only depend on control parameters, not audio, it is efficient to only recompute them when the control parameters change, and store the intermediate result internally. In the case of a Kronos program, the intermediate results would be generated implicitly, due to the filter containing a signal rate boundary. It is notable that in the case of there being audio rate modulation of the filter parameters, the compiler automatically generates a stateless direct connection. Both cases are

similar to how a manual implementation in the imperative idiom would be written, but entirely derived from the data flow of the program.

Further isomorphisms between the Kronos language and object oriented programming are discussed in P1 .

3.1.3 Alternate Implementation Strategies

An extremely important question in the evaluation of any novel programming language is whether the claimed benefits could have been realized with some tools that already exist. This litmus test is important for several reasons. Firstly, a new language is a formidable commitment for both the author and the end users. If similar results can be had with existing tools, such a commitment is difficult to argue for. Secondly, established languages benefit from a large body of existing work. Tutorials, libraries and tooling is readily available. The situation is unavoidably worse for a niche language; the deficit must be compensated by improvements not available elsewhere.

This section briefly examines some alternatives to reaching results similar to those provided by the Kronos technology described in this report.

C++ Templates

The C++ template metacompiler [41] shares some important features with the Kronos compiler, and has been an important influence in its design. Generic C++ classes and functions are similar to Kronos source programs, in the sense that they are dynamically typed until reified from a set of root types – similar to the process described in Section 2.1.3. Like Kronos, the template compiler performs simple type derivation.

It is conceivable that a domain language implemented in C++, with heavy use of templates, could be designed to offer semantics similar to those presented in this report.

Such an approach would, to the best of the author's knowledge, likely fail. While the C++ template system is expressive enough to model the semantics of the Kronos language, it is not, as of this writing, designed to scale to similar extents. Kronos can effortlessly specialize a typical recursive function to depths of thousands or millions, something not generally recommended for C++ templates. The design complexity of such a library would also likely rival or exceed that of the Kronos compiler.

The second concern relates to error messages. It is not clear how a C++ library with such heavy reliance on templates could be designed to fail with error messages that make sense to the target demographic. This could conceivably change in the future, as concepts are introduced to C++, but would likely still remain problematic.

The third concern is compilation time. Just in time compilation for C++ is rare and cumbersome. The compile cycle is slow, especially in the case of extensive use of templates. Immediate feedback and conversational programming [14] seem out of reach.

In summary, a domain language similar to Kronos, designed as a C++ library, would likely be very hard to design, difficult to use and slow to compile. The possible benefits are not apparent to the author.

Using a Proven Dynamic Language to Generate DSPs

The Kronos programming language can be seen as a two-layered design: a dynamic metaprogramming language (the type system) targeting a low level efficient representation (the reified program). It is conceivable that an estabilished dynamic language could be used as the metaprogramming layer, targeting a suitable low level representation – which could well be the reified stage of a Kronos program.

Sorensen's [50] Impromptu Compiler Runtime comes close to this approach. In ICR, the SCHEME language is intertwined with a more restrictive, statically typed domain language called *xt-lang*. Here, SCHEME corresponds to the generic Kronos program, while *xt-lang* corresponds to the reified, statically typed Kronos program. *Xt-lang* is roughly on the abstractive level of C, and provides similar performance characteristics. Its main advantages are the tight integration with the Impromptu environment for rapid, incremental development, and a syntax more amenable to SCHEME-based metaprogramming.

It should be admitted that the isomorphism between these approaches was not properly understood by the author before the insights afforded by this study. Nevertheless, there are arguments that can be made in retrospect for the design of a bespoke language–compiler stack.

The dichotomy of two languages, as exhibited in ICR, can not be escaped in an approach like this. This is not a criticism of the approach: to expect the power of SCHEME in the context of signal processing is wildly optimistic with current technology.

A similar dichotomy exists in the Kronos type system; the types capable of influencing flow control, including the lifted number and string types, belong to the dynamic layer, while run time variant values belong to the static layer.

The type reification rules in Kronos are simple; defined in terms of the System F_ω. The primitives of the language are defined to make sense in the two-layered construction. Function application, pair construction and destructuring are lowered to the static idiom automatically. To replicate this functionality in a dynamic language, basic primitives would need to be overloaded to treat dynamic and static data differently. Function application would need to perform extensive meta-reflection to determine whether the call should be lowered to the static idiom or applied during the dynamic pass. If the overloading couldn't be done at the level of primitives, as is the case in SCHEME, only functions that understand and respect the dual domains could be safely used in user programs. The benefits of interfacing with existing code bases would be diminished, since they would not be designed for the dual layer approach.

In Kronos, the layer dichotomy is represented cleanly by the Simple F_ω system, as described in Section 2.1.4. The capabilities and limitations of each layer result from simple, well defined rules. It is not possible to write non-compliant programs. The Kronos approach to metaprogramming and productivity is not the only one; indeed, ICR [50] is an interesting, viable method for signal processing. However, advantages provided by the Kronos compiler stack, such as reactive factorization, would be hard to integrate.

3.2 FUTURE WORK

In most signal processors, the majority of signal paths are hard coded. In Kronos, design of the type-deterministic signal graph reflects this assumption. Some dynamic routings can be achieved by signal clock gating and selection operators; however, it should be noted that these facilities are aimed at avoiding wasted computation, rather than true dynamic reconfigurability.

There are cases where the lack of a run time reconfigurable signal graph is truly detrimental, especially as the programs move from low level signal processing kernels to more complicated high level systems, such as programs that represent complete musical works or large parts thereof. Next, a number of problematic areas in the current Kronos compiler and language, as well as avenues for potential improvement, will be discussed.

Auralization of Scores

Consider the auralization of a musical score. One elegant, widely applied mapping is from score object to program object; a signal processor is instantiated per note, containing state that is private to that note, to be disposed of as the note ends. Such semantics are not easily expressible in the Kronos language: it is focused on data flow and signal processing, as opposed to higher-level instantiation and scheduling of signal processors.

This is evident form the fact that the language lacks dynamic memory semantics and mutable signal graphs. Further, the stateful operators are designed to "hide time" from the programmer. Kronos programs are, in their entirety, topologies rather than chronologies. This very fact enables the compiler to treat stateful operators as pure functions – which in turn enables almost all of the novel capabilities it offers. At the same time, this design precludes programmatic access to explicit flow control, as the compiler must have precise control over program execution order.

It could be argued that musical notation is an excessively high level abstraction for signal processor control data. Likewise, the idioms ideal for score processing and signal processing are not necessarily related or even compatible; in the MUSICn [16] tradition, different programming languages are used to address scores, instruments and orchestras.

Perhaps the problem of an all-encompassing musical programming environment would be better solved by providing Kronos bindings to a suitable scripting language in which the score and object logic could be written. This approach has promise, but also a number of pitfalls related to the integration of the idioms and a potential mix-up of semantics such as described in Section 3.1.3.

Dynamic Branches

As noted in Section 2.1.4, the underlying computational structure permitted for Kronos programs is strict and restrictive. The foundation of the design is to make the metaprogramming layer provided by the type system as flexible and expressive as possible, while providing an easily optimizable intermediate representation to the code generator. If the core language featured dynamic branching, it would significantly extend its coverage to a range of new problem domains; working with scores and high level musical representations could be one of them.

A promising approach for implementing dynamic branches is to apply type erasure to closures. They are currently distinct types based on the function body and capture list. Erased closures could be interchangeable at run time, enabling dynamic control flow via selection from an array of closures.

Questions remain: how would the implicit state of the closures be managed and what are the semantics of mutating the signal graph in relation to stateful memory operators in each branch target? How to minimize the performance impact of a dynamic branch, especially as a simple implementation would see it evaluated per sample? An interesting approach could involve the LLVM [27] patchpoint mechanism and self-modifying code.

I/O and Programs with Effects

Functional programs are well suited for expressing data flows. The functional paradigm works equally well in musical signal processing: in addition to the findings in this study, the success of the Faust project [23] is a compelling argument.

However, the functional paradigm is less suited for expressing the effects of the program on the surrounding world. If a program is to be observable and potentially useful, it must, at some point, observe or mutate the state of a human interaction device such as a speaker, video display or a keyboard.

The approach shared by this study and the Faust project [23] is to externalize these concerns. Faust programs produce an audio stream by definition, and it is the responsibility of the surrounding architecture [46], expressed in a different programming language, to bridge the gap between functional signal processors and the stateful devices of interaction.

There are purely functional programming languages that do deal with these problems; Haskell [51] is a prominent example. Haskell programs with I/O appear imperative at a first glance. Monads are employed to construct data flow programs out of sequential imperative programs; finally, a restricted set of external impure functions is utilized to add I/O functionality. The I/O language in Haskell is an *embedded domain language*, used to perform top-level sequencing, while pure functional code is used for processing data.

The wording of *top-level sequencing* evokes musical programming. As Kronos focuses on metaprogramming, an embedded domain language for I/O and effects could well be viable. The I/O commands could include message sending protocols such as OSC [25] or MIDI. Instantiation of Kronos closures as signal processors could be one of the effects. The addition of scheduling code via timed callbacks would enable temporal recursion [50] and open the door to a variety of time-variant programs [52, 53].

A domain language such as the one envisioned in the preceding paragraph would be subject to the criticism presented in Section 3.1.3. This is still a compelling avenue for future work, as the integration between the signal processor idiom and the score level idiom could be more seamless than would be possible otherwise. The score language would in fact be built on the metaprogramming facility that also generates the signal graphs it manages – the scripting language alluded to in Section 3.2 would in fact be directly embedded in the core language.

Automatic Parallelization

Automatic parallelization of user programs is yet another interesting topic. Because current hardware trends emphasize throughput over latency, the parallel work units should consist of vectors of samples, without fine grained synchronization with other work units. A successful implementation is related to the dynamic branch problem mentioned in Section 3.2; it would be beneficial to be able to isolate subgraphs of the user program and vectorize them, for both dynamic and parallel dispatch.

The language semantics afford this well, as both the reactive model and the type derivation processes can be made transparent to whether the subgraphs of a program are compiled into a single signal processor or several interconnected modules. Providing an automatic signal processor modularizer is not unreasonable, and could provide the basis for both dynamic semantics and automatic parallelization.

For most cases, though, parallelization is better left outside the signal processor. Typically, load balancing is best achieved by the host program that runs the processor within its own audio graph. Nested parallelization and synchronization structures are especially inefficient and problematic. However, providing parallelization at the compiler level could expand the domain covered by the Kronos language from low level signal processors towards more complicated standalone music rendering systems, especially in combination with a instance management and scheduling solution.

Automatic Vectorization

Automatic, transparent vectorization for SIMD [54] architectures was studied and prototyped during this project. This work was deemed outside the scope of this report, but preliminary results are promising. Generic types, overloading and metaprogramming work well for constructing instruction level data parallelism from arbitrary algorithms. Typical auto-vectorization occurs late in the

compiler optimizer pipeline. There are a number of benefits in applying it at the type derivation stage: this way, the compiler sees vectorized constructs already when performing early data flow analysis and state reification, allowing it to generate SIMD-friendly constructs naturally.

In future work, carefully designed and tested vectorizing variants of functions like `Map` and `Reduce` could be deployed. As generic programming is ubiquitous in Kronos, most signal processors are amenable to vectorization even when not explicitly designed for it.

Instrumentation, Tooling and Learnability

To realize the pedagogical and conversational [14] potential of the Kronos language and compiler, supporting toolchain must be built. A patcher prototype, deemed outside the scope of this study, exists [32], but has seen limited classroom use as of yet. For a proper evaluation of whether the language can meet the promise to cater to domain experts, further study in this area is required.

Recent trends in learning and deeply understanding programming can be traced back to the seminal work by Papert [55]. His work deals primarily with children, but has been recently applied more generally to multiple kinds of learners. Victor [56] offers a contemporary take on the subject.

The Kronos compiler successfully fulfills an important technical prerequisite to conversational [14], learnable programming: compile times are extremely short. Thus, programmer feedback can be fairly immediate. However, to fully enable the conversational development model, the programmer tools should be developed to enable extensive and immediate visual feedback. A recent example of a study of programmer–environment interaction was done by Lieber et al [57]. A simple code metric displayed as continuous feedback, integrated within the code editor, was found to influence developer workflows significantly.

The musical signal processing domain is conductive to such tooling, as users are widely familiar with standard instrumentation such as level meters, oscilloscopes and transfer functions. An ideal interactive debugger for signal processors would likely offer such instrumentation instead of the traditional textual watch windows. The implementation and study of new programming tools built on the Kronos infrastructure is an important future topic.

Applications and Libraries

Further study of the performance and capability of the Kronos compiler and language depend on a large set of test cases. The domain of applying higher abstraction to the generation of signal processors is also largely unexplored. The metaprogramming facilities of the compiler provide a lot of potential for "magic" and automation. Depending on how well the program code is designed, such magic can seem either helpful and productive, or opaque and confusing. The principles and best practices for striving for the former require experimentation and solidifying.

The examples listed in Appendix B describe how the semantics of the language support constructs such as higher order functions and closures for a flexible system of algorithmic routing. The application and development of such principles in various subdomains of signal processing, including filter design, spectral analysis, circuit modeling and reverberation is an interesting avenue for further research.

The libraries themselves would further enhance the pedagogical potential of the platform, as the processors could readily be supplied in source form, from high level abstractions down to the tiniest of details.

4 CONCLUSION

This study presented Kronos, a signal processing language and a compiler suite built on the LLVM [27] infrastructure. The project consists of this written report, the compiler software, and example programs, all of them openly available to interested parties.

The Kronos language strives to lower the barrier to entry in the fundamentals of signal processing, especially to musicians and composers interested in such activities. The research problem was formulated as a signal processing language that offers the following main features (1.3);

1. **Unified Signal Model**
2. **Composable and Abstractive Ugens**
3. **Visual Programming**

This design addresses some key weaknesses in the state of art. The popular visual programming surface is not conductive to abstraction in the imperative domain, but the problem is resolved with functional programming techniques (2.1.1). The functional paradigm is also synergistic with the idea of increased abstraction and composition of unit generators.

The typical implementation strategy utilized in most signal processing languages, the *ugen interpreter*, also disincentivizes code composition, reuse and abstraction. The abstractive principle – information hiding – should not be automatically understood as something that increases the difficulty and sophistication of programs. Rather, by hiding information that is irrelevant, it should enhance them with clarity and conciseness.

The design challenge of abstract systems is to deem what is relevant and what is not. This distinction is greater in a strictly domain specific language, in comparison to a general purpose language. If a particular programming paradigm is enforced, more assumptions can be made about the programmer intent, and fewer details need explication. Further, the compiler can be built to perform program transformations that would be very hard or impossible to integrate to general purpose languages. In the Kronos language, such transformations include automated reification of abstract state memory (2.1.2), whole-program type derivation (2.1.3), and most importantly, reactive factorization of a single data flow into distinct clock regions (2.1.5).

Such transformations, in aggregate, enable a programming model where source code is simple and minimal, supports abstraction and composability, yet runs very efficiently. The Kronos system can be seen as a dual language: the expressive type system, utilized for metaprogramming purposes, and the less expressive, static, streamlined runtime. The dichotomy of these two is expressed cleanly as *Type Determinism* in terms of the Simple F_ω (2.1.4, P3).

The implementation of Kronos in the C++ programming language [11] and its usage as a library is discussed in Section 2.2, which is a high level technical document of the source code.

The executable result of the Kronos compiler is quite close to that of state of art optimizing compilers for object oriented programs in C++ [11]. Object oriented languages emphasis the explicit transfer of control in code, or the *chronological* sequence of instructions. The Kronos model promotes a *topological* program based on data flow. It is naturally suited for visual representation and

manipulation. Further, the data flow model offers a clearer, more concise representation of typical signal processors. This is especially true in the case of multirate signal processing. On the other hand, the Kronos representation is certainly less flexible: if the proposed semantics for a unified signal model (P1 , P2) do not suit the task at hand, the language has limited capability to adapt.

The principal outcome of the study is a programming model with novel tradeoffs, as discussed above. The combination of an expressive metaprogramming layer with highly restricted computational model enables the use of advanced functional techniques like closures, higher order functions and ad-hoc polymorphism [43]. This is novel in the context of signal processing, traditionally a realm of low level of abstraction. By delimiting the advanced techniques to the metaprogramming layer, the resulting program runs as efficiently as most manually optimized low level programs.

The viability of the programming model for an expanding subset of musical programming task is an important topic of future study. The ability of domain experts to adopt a functional, reactive programming model is interesting from both a pedagogical standpoint as well as an indicator of a successful design. The best practices for designing programs that are clear in intent in the visual domain are as of yet not systematically explored. To maximize the pedagocical and conversational [14] potential of the language, the study should be extended to development tools and the programmer experience. Likewise, further technical advancements, like parallel and vector processing, are interesting topics for a metaprogramming-based approach.

REFERENCES

[1] Vesa Norilo. Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. *Computer Music Journal*, 39(4), 2015.

[2] Vesa Norilo and Mikael Laurson. A Unified Model for Audio and Control Signals in PWGLSynth. In *Proceedings of the International Computer Music Conference*, Belfast, 2008.

[3] Vesa Norilo. Introducing Kronos - A Novel Approach to Signal Processing Languages. In Frank Neumann and Victor Lazzarini, editors, *Proceedings of the Linux Audio Conference*, pages 9–16, Maynooth, 2011. NUIM.

[4] Vesa Norilo. Designing Synthetic Reverberators in Kronos. In *Proceedings of the International Computer Music Conference*, pages 96–99, Huddersfield, 2011.

[5] Digital Audio Effects. Kronos Vst – the Programmable Effect Plugin. In *Proceedings of the International Conference on Digital Audio Effects*, Maynooth, 2013.

[6] Vesa Norilo. Recent Developments in the Kronos Programming Language. In *Proceedings of the International Computer Music Conference*, Perth, 2013.

[7] Max V Mathews. An acoustic compiler for music and psychological stimuli. *Bell System Technical Journal*, 40(3):677–694, 1961.

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

[9] Yann Orlarey, Dominique Fober, and Stephane Letz. FAUST: An Efficient Functional Approach to DSP Programming. In Gérard Assayag and Andrew Gerszo, editors, *New Computational Paradigms for Music*, pages 65–97. Delatour France, IRCAM, Paris, 2009.

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

[11] 14882:2011. *Information technology - Programming languages - C++*. ISO/IEC, 2011.

[12] Andrew Hugill. *The Digital Musician*. Routledge, New York, 2008.

[13] Linda Candy. Research and Creative Practice. In Linda Candy and Ernest Edmonds, editors, *Interacting - Art, Research and the Creative Practitioner*, pages 33–59. Libri Publishing, Faringdon, 2011.

[14] Julian Rohrhuber and Alberto de Campo. Improvising Formalisation: Conversational Programming and Live Coding. In Gérard Assayag and Andrew Gerzso, editors, *New Computational Paradigms for Music*, pages 113–125. Delatour France, IRCAM, 2009.

[15] Peter Manning. *Electronic and computer music*. Oxford University Press, 2013.

[16] Victor Lazzarini. The Development of Computer Music Programming Systems. *Journal of New Music Research*, 42(1):97–110, March 2013.

[17] Richard Boulanger. *The Csound Book*, volume 309. MIT Press, 2000.

[18] Miller Puckette and David Zicarelli. *MAX - An Interactive Graphical Programming Environment*. Opcode Systems, 1990.

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

[20] Victor Lazzarini. Audio Signal Processing and Object-Oriented Systems. *Proceedings of the 2002 International Conference for Digital Audio Effects*, pages 211–216, 2002.

[21] T W Pratt. *Design and Implementation of Programming Languages*. Number PRG-40 in (LNCS 54). Prentice Hall, 2010.

[22] Adrian Freed, John MacCallum, and Andrew Schmeder. Composability for Musical Gesture Signal Processing using new OSC-based Object and Functional Programming Extensions to Max/MSP. In *New Interfaces for Musical Expression*, Oslo, Norway, 2011.

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

[24] Pierre Jouvelot and Yann Orlarey. Dependent vector types for data structuring in multirate Faust. *Computer Languages, Systems and Structures*, 37(3):113–131, 2011.

[25] Matthew Wright, Adrian Freed, and Ali Momeni. OpenSound Control: State of the Art 2003. In *Proceedings of NIME*, pages 153–159, Montreal, 2003.

[26] Ge Wang. The ChucK Audio Programming Language : A Strongly-timed and On-the-fly Environmentality. *PhD Thesis*, (September):175, 2008.

[27] C Lattner and V Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization 2004 CGO 2004.*, 57(c):75–86, 2004.

[28] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare. PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control. *Computer Music Journal*, 29(3):29–41, 2005.

[29] Vesa Norilo and Mikael Laurson. Kronos - a vectorizing compiler for music dsp. In *Proc. Digital Audio Effects (DAFx-10)*, pages 180–183, Lago di Como, 2009.

[30] Vesa Norilo and Mikael Laurson. A method of generic programming for high performance {DSP}. In *Proc. Digital Audio Effects (DAFx-10)*, pages 65–68, Graz, 2010.

[31] Vesa Norilo. A Grammar for Analyzing and Optimizing Audio Graphs. In Geoffroy Peeters, editor, *Proceedings of International Conference on Digital Audio Effects*, number 1, pages 217–220, Paris, 2011. IRCAM.

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

[33] Vesa Norilo. Kronos as a Visual Development Tool for Mobile Applications. In *Proceedings of the International Computer Music Conference*, pages 144–147, Ljubljana, 2012.

[34] Mika Kuuskankare and Vesa Norilo. Rhythm reading exercises with PWGL. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8095 LNCS, pages 165–177, Cyprus, 2013.

[35] Josue Moreno and Vesa Norilo. A Type-based Approach to Generative Parameter Mapping. In *Proceedings of the International Computer Music Conference*, pages 467–470, Perth, 2013.

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

[37] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.

[38] Peter Van Roy. Programming Paradigms for Dummies: What Every Programmer Should Know. In Gérard Assayag and Andrew Gerzso, editors, *New Computational Paradigms for Music*, pages 9–49. Delatour France, IRCAM, Paris, 2009.

[39] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[40] G Kiczales. Aspect-oriented Programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[41] David Abrahams and Aleksy Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth. Addison Wesley, 2005.

[42] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997.

[43] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[44] HP Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):124–154, 1991.

[45] David R Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Practice and Experience*, 20:5–12, 1990.

[46] Dominique Fober, Yann Orlarey, and Stephane Letz. FAUST Architectures Design and OSC Support. In *Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 213–216, 2011.

[47] Gary Scavone and Perry Cook. RtMidi, RtAudio, and a Synthesis ToolKit (STK) update. In *Proceedings of the 2005 International Computer Music Conference*, pages 327–330, Barcelona, 2005.

[48] Romain Michon and Julius O. Smith. FAUST-STK: a set of linear and nonlinear physical models for the FAUST programming language. In Geoffroy Peeters, editor, *Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 199–204. IRCAM, Centre Pompidou, 2011.

[49] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[50] Andrew Sorensen and Henry Gardner. Programming With Time Cyber-physical programming with Impromptu. *Time*, 45:822–834, 2010.

[51] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell. *Proceedings of the third ACM SIGPLAN conference on History of programming languages HOPL III*, pages 12–1–12–55, 2007.

[52] Roger B Dannenberg. Expressing Temporal Behavior Declaratively. In Richard F Rashid, editor, *CMU Computer Science, A 25th Anniversary Commemorative*, pages 47–68. ACM Press, 1991.

[53] Graham Wakefield, Wesley Smith, and Charles Roberts. LuaAV: Extensibility and Heterogeneity for Audiovisual Computing. *Proceedings of the Linux Audio Conference*, 2010.

[54] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[55] Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books Inc, New York, 1980.

[56] Bret Victor. Learnable Programming, 2012.

[57] Tom Lieber, Joel Brandt, and Robert Miller. Addressing Misconceptions About Code with Always-On Programming Visualizations. In *Proceedings of ACM CHI*, Toronto, 2014. ACM.

Part II

Publications

SUMMARY OF THE PUBLICATIONS

P1 KRONOS: A DECLARATIVE METAPROGRAMMING LANGUAGE FOR DIGITAL SIGNAL PROCESSING

P1 provides an up-to-date overview of the Kronos project. The article discusses the prominent programming paradigm, the *unit generator graph*, and several existing programming languages departing or enhancing the model. An in-depth look at Faust [23] is provided and Kronos and Faust are contrasted and compared. The principle of metaprogramming is presented as a highly general solution to signal graph generation. The reactive multirate signal processing scheme is analyzed and isomorphisms to object oriented programming are identified. The author of this report is the sole author of the article, which is included here as a revised manuscript.

P2 A UNIFIED MODEL FOR AUDIO AND CONTROL SIGNALS IN PWGLSYNTH

P2 presents the founding principle of the reactive multirate factorization that is a cornerstone of this work. It develops and generalizes the traditional concept of control and audio rate and its relation to data and demand driven scheduling of computation. These concepts were developed within the context of *PWGLSynth*. The article is written by the author. The PWGLSynth engine discussed in the article has been designed jointly by the author and Dr Mikael Laurson.

P3 INTRODUCING KRONOS — A NOVEL APPROACH TO SIGNAL PROCESSING LANGUAGES

P3 presents an early overview of Kronos and many of its founding principles. It focuses on the application of multirate factorization to various elementary signal processors such as delay, basic reverberation and equalization. The article is written by the author of this report in response to an invitation by Dr Victor Lazzarini on behalf of the Linux Audio Conference, Maynooth, Ireland, 2011.

P4 DESIGNING SYNTHETIC REVERBERATORS IN KRONOS

P4 discusses a specific application of generic programming to artificial reverberators. The application of higher order functions to building complicated signal networks is demonstrated via the case of the feedback delay network. The article is written by the author of this report.

P5 KRONOS VST — THE PROGRAMMABLE EFFECT PLUGIN

P5 introduces a new version (K3) of the Kronos compiler developed during this project. It is demonstrated via integration with VST, a commercial technology for integrating audio processors in various audio workstation software packages. The Kronos compiler is leveraged to provide a programmable VST plugin, compiling user programs on the fly during the use of an audio workstation. The article is written by the author of this report.

P6 RECENT DEVELOPMENTS IN THE KRONOS PROGRAMMING LANGUAGE

P6 presents theoretical work on the K3 version of the Kronos compiler. It elaborates the multirate signal model and presents the novel sequence specialization scheme that improves compile time performance for large data sets. The article was written by the author of this report.

P1 KRONOS: A DECLARATIVE METAPROGRAMMING LANGUAGE FOR DIGITAL SIGNAL PROCESSING

Vesa Norilo. Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. *Computer Music Journal*, 39(4), 2015

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.

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

```
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.

```
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

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 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.

P2 | A UNIFIED MODEL FOR AUDIO AND CONTROL SIGNALS IN PWGLSYNTH

Vesa Norilo and Mikael Laurson. A Unified Model for Audio and Control Signals in PWGLSynth. In *Proceedings of the International Computer Music Conference*, Belfast, 2008

A UNIFIED MODEL FOR AUDIO AND CONTROL SIGNALS IN PWGLSYNTH

Vesa Norilo and Mikael Laurson
Sibelius-Academy
Centre of Music and Technology

ABSTRACT

This paper examines the signal model in the current iteration of our synthesis language PWGLSynth. Some problems are identified and analyzed with a special focus on the needs of audio analysis and music information retrieval. A new signal model is proposed to address the needs of different kinds of signals within a patch, including a variety of control signals and audio signals and transitions from one kind of signal to another. The new model is based on the conceptual tools of state networks and state dependency analysis. The proposed model aims to combine the benefits of data driven and request driven models to accommodate both sparse event signals and regular stream signals.

1. INTRODUCTION

The central problem of a musical synthesis programming environment is to maintain a balance of efficient real time performance, expressiveness, elegance and ease of use. These requirements often seem to contradict. Careful design of the programming environment can help to mitigate the need for tradeoffs.

The original PWGLSynth evaluator [3] is a ugen software that features a visual representation of a signal graph [1]. It was written to guarantee a robust DSP scheduling that is well suited for tasks including physical modelling synthesis. This was accomplished by scheduling the calculations by the means of data dependency - in order to produce the synth output, the system traverses the patch upstream, resolving the dependencies of each box in turn. This signal model is often referred to 'request driven' or 'output driven'. The model has the distinguishing feature of performing computations when needed for the output, and is well suited for processing fixed time interval sample streams.

The opposite model is called 'data driven' or 'input driven'. Calculations are performed on the system input as it becomes available. This model is well suited for sparse data such as a sequence of MIDI events. The input driven model is represented by the MAX environment[6]. The bulk of PWGLSynth scheduling in the current version is output driven. Some benefits of the input driven approach are available via the use of our refresh event scheme, which can override the evaluation model for a particular connection.

The rest of this paper is organized as follows. In the first section, 'PWGLSynth signal model', some problems in the current signal model are examined, focusing on audio analysis and music information retrieval. In the second section 'Optimizing signal processing', a new, more general signal model is presented. The new model combines simplified patch programming with robust timing and efficient computation, allowing the user to combine different kinds of signals transparently. Finally, in the last section, 'Signals in a musical DSP system', the proposed system is examined in the context of audio analysis.

2. THE PWGLSYNTH SIGNAL MODEL

PWGLSynth was designed for the scenario where Lisp-based user interface elements or precalculated sequencer events provide control data for a synthesis patch. We wanted to avoid splitting the system into audio rate and control rate paths, and developed the PWGLSynth refresh scheme which mixes output driven evaluation with data driven control events. In practice, boxes can get notified when their incoming control signal changes.

This notification is called a refresh event. A box can respond to a refresh event by performing some calculations that assist the inner audio loop. When audio outputs are connected to inputs that require refresh, the system generates refresh events at a global control rate. For example, an audio oscillator can be connected to a filter frequency control with the expected behaviour while still avoiding the need to recalculate filter coefficients at audio rate. When considering audio analysis, the scenario changes drastically. Control signals are generated from an audio signal, often in real time.

2.1. Prospects for audio analysis and music information retrieval

Audio analysis essentially involves a mixture of sparse event streams and fixed interval sample streams. Some analysis modules will recognize certain discrete features of an input stream, while others will retrieve some higher level parameter from an input stream, usually at a lower signal rate than that of the audio signal. A buffered FFT analysis might be triggered at a certain sample frame, resulting in a set of high level parameters that require further processing.

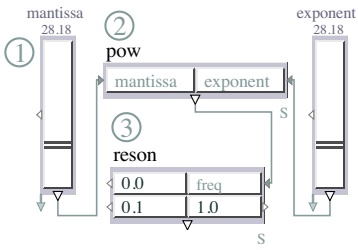

Figure 1. Simple example of cached computation results.

It is also conceivable to extract some high level control parameters from an audio stream and then use them for further audio synthesis. The potential of a system with seamless analysis and synthesis facilities is discussed in [7].

2.2. Towards a general unified signal model in a mixed rate system

The PWGLSynth Refresh scheme could in theory be adapted to suit audio analysis. A FFT box could store audio data internally until a buffer is full, then perform analysis and generate refresh events along with new output data. However, PWGLSynth provides no guarantees on the timing of refresh events generated during synthesis processing, as the scheme was devised for user interface interaction and automatic conversion of audio into control rate signal. The refresh scheme is well suited for simple control signal connections, but is not sufficient for the general case.

Since the refresh calls happen outside the synthesis processing, a unit delay may occur in the transition of the signal from audio to control rate. While not critical for user interface interaction, even a small scheduling uncertainty is not practical for audio analysis, where further processing will often be applied to the control signal. It is important to have well defined timing rules for the case of mixed signal rates.

Why not employ audio rate or the highest required rate for all signals? While this would guarantee robust timing, the very central motive for using a control rate at all is to optimize computation. Efficient handling of control signals can increase the complexity limit of a patch playable in real time, extend polyphony or reduce computation time for an offline patch.

3. OPTIMIZING SIGNAL PROCESSING

3.1. State-dependency analysis

The central theme in DSP optimization is always the same: how to avoid performing unnecessary calculations without degrading the output. The signal model and system scalability are a central design problem of any synthesis programming environment [5].

A simple example patch is given in Figure 1. In this patch, a set of sliders, labeled (1), represent the user interface. An intermediate power function (2) is computed based on the slider values, finally controlling the corner frequency of a filter (3).

When a human observer looks at the patch, it's immediately obvious that some calculations are not necessary to perform at the audio rate. This finding is a result of a dependency analysis, aided by our knowledge of mathematical rules. Updating filter coefficients tends to be computationally expensive, not to speak of the power function. Yet, the coefficients ultimately depend on only the two values represented by sliders. In other words, the power function or the coefficients will not change unless the user moves one or both of the sliders. This can be expected to happen much more rarely than at the audio rate.

This is a very specific case that nevertheless represents the whole control scheme quite generally. The key idea is to note that certain signals do not change often and to avoid calculations whenever they don't. Traditional modular synthesis systems have separated signals into audio rate and control rate signal paths. In this scheme, the programmer is required to explicitly state which signal rate to use for any given connection. Often, separate modules are provided for similar functions, one for each signal rate. A unified signal model on the other hand greatly decreases the time required to learn and use the system, as well as increases patch readability, often resulting in compact and elegant synthesis patches.

3.2. Functional computation scheme

In the previous example, the dependency analysis is easy to carry out because we know the power function very well. Its state only depends on the mantissa and exponent. It is less obvious what would happen if the box would be some other, more obscure PWGLSynth box.

The power function has an important property we might overlook since it is so obvious, yet carries deep theoretical meaning: it is strictly functional, meaning that it has no internal state. All power functions behave exactly the same, given identical input, at all times. This is unlike a recursive digital filter, which has an internal state that influences its output.

It turns out that many of the DSP operations can be carried out with functional operators. These include the common arithmetic and all stateless functions. When we extend the allowed operations by an unit delay, practically any known algorithm is attainable. By formulating the synthesis process as a functional tree expression, dependencies are easy to find. When a value in the patch changes, only the values downstream in the patch from it will need to be recomputed. Functional representation has many benefits, as shown in PWGL, Faust [4] or Fugue [2], all successful musical programming languages.

Traditional digital signal processing relies heavily on internal state, which can be represented by a filter buffer

or a delay line feedback path. For the needs of dependency analysis, full functional rigor is not required. We only need to recognize that DSP modules with internal state will produce different output with identical input, but different moment in time.

By adding a 'time' source upstream of all modules with state we can make sure that by representing the time via a sample clock as a fundamental part of our functional computation tree, we can include modules with state. Correct, strictly functional behavior is ensured by functional dependency analysis once time is recognized as a parameter to all stateful modules.

3.3. State change propagation and synchronic updates

Our proposed system, aimed towards an intuitive yet efficient computation model, mixes aspects of input and output driven models. The patch is modeled with a number of endpoints which are, in effect, the state space of the system. Every point at which an user can tap into the patch is given a state. These states form a network, where each endpoint is connected to other endpoints by computational expressions. By utilizing functional building blocks, we can carry out dependency analysis and determine which states need to be recomputed when a certain state value is changed.

Thus the actual computation process resembles the output driven model, where computations are carried out when output is needed. The distinction is, however, that the output can 'know' when a recomputation is needed since it will be notified of updated input states. In effect, an output driven computation schedule is created for every input state of the system. A change in the input state then triggers the processing that updates its dependent states, in an input driven fashion.

A special case arises when several states must be updated synchronously. If each state update triggers recomputation of the relevant dependent states, and a state depends on several updated states, multiple updates are unnecessarily carried out. In addition, this would break any timing scheme that requires that updates happen regularly with the sample clock, such as a unit delay primitive.

This problem can be solved with update blocks that consist of two steps: first, all states that are about to be updated will be marked as pending. This pending status also propagates downstream in the state network, with each state keeping count of the number of pending states it depends on.

The second step consists of updating the state and releasing the pending flag. During release, all dependent states will decrement their pending counter and when it reaches zero, all states they depend on will have updated their value and the computation can be performed.

For efficiency, a third update mode is introduced. It will ignore the pending counters and just trigger computation and reset the pending counter to one. This mode is what will be used for the most frequently updated state, namely the sample clock. This allows for avoiding branches inside the innermost audio loop while marking all states that depend on the sample clock pending until the next input. Viewed within the entire model, the sample clock is always pending, apart from the moment that the actual audio rate computation is performed. This leaves the portions of the patch that dont deal with audio available for control rate computations outside the inner audio loop.

3.4. Signal model overview

The complete synthesis scheme goes as follows: by default, boxes that depend on the sample clock are by default set to 'pending' with counter 1. Before updating the sample clock, all control event or user interface state changes are updated as a block and released. This triggers all calculations that do not depend on the sample clock. Finally, sample clock is updated. Since dependencies reflect on the whole patch downstream from a given box, the sample clock update is in fact the actual audio computation.

Intermediate states can be automatically inserted at patch points where signal rates mix. These states work as cached results of the lower rate signal process for the higher rate process.

4. SIGNALS IN A MUSICAL DSP SYSTEM

4.1. Coarse and fine time

To further ease box development, more than one clock source can be provided on the global level. Modules that require clock input could connect to either an audio rate or a control rate clock source. In most cases this connection should be made implicit and not visible to the user. A typical example with several different clock source possibilities would be an oscillator that functions as either an audio oscillator or a LFO for some computationally expensive operation.

It is possible to add metadata to the modules in order to automatically choose the most appropriate clock source for a given situation. If a sine oscillator is connected to an input that prefers a coarse control signal, it can automatically revert to a coarse clock and therefore produce an appropriate signal. Clocking could also be made an optional user parameter.

4.2. Signal rate decimation in audio analysis

Considering audio analysis from the perspective of the proposed paradigm, we encounter a further problem. Consider a typical buffered analysis scheme, where high level parameters are retrieved from a block of audio samples. This decimates the signal rate, as only one output value is produced per sample block.

Regardless, there is a functional relation between the audio data and the extracted parameters, implying that within the dependency rules outlined above, the high level analysis results must also be refreshed at audio rate. For modules that produce a control rate signal based on an audio rate signal, an intelligent scheduling scheme is required.

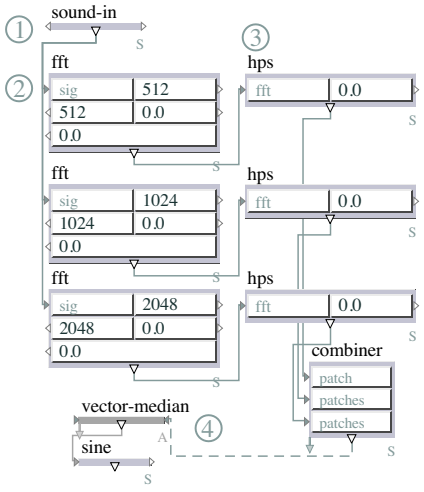

Figure 2. An audio analysis patch.

The system is completed by a sample-and-hold primitive, accessible to box developers. The primitive is able to break a dependency chain. It takes two inputs, updating its output to the first input when and only when the second input changes. This makes it possible for a box designer to instruct the scheduler to ignore an apparent dependency.

This exception to the scheme causes some complications. When resolving the order in which states need to be refreshed when an upstream state changes, all dependencies must be traced through the sample-and-hold, even though the chain is broken for the first input. This is to enforce correct scheduling in the case where the decimated rate signal is merged back into the higher rate stream.

4.3. Redundant f0 estimation - a case study

The example patch in Figure 2 demonstrates the scheme in audio analysis.

Three f0 estimators with different frame size work in parallel for increased pitch detection robustness, driving a simple sine oscillator that follows the pitch of the audio input. The modules that depend on audio rate sample clock are sound-in and sine. The three FFT modules (1) therefore also depend on the sample clock, but break the downstream dependency since their output (the spectrum) changes with a lower rate.

The audio input to each FFT fills a ring buffer, without causing any output state refreshes. Timing is provided by assigning a coarse clock signals for each FFT module. The clock update is timed to provide an update when the ring buffer is full.

Upon a coarse clock update, the FFT computations are performed, f0 estimation is carried out and the median frequency is fed into the sine osc. However, since the sine osc depends on the fine clock, its operation is suspended and separately activated by the fine clock after the analysis step. Thus a delay-free and correctly scheduled audio - analysis - audio signal path is preserved with no waste of computational resources.

5. CONCLUSION

In this paper, we examined a signal graph processing system from a theoretical viewpoint. Some criteria for avoiding wasted computation operations were examined. We proposed a new signal model as a hybrid of two well known signal model schemes, which offers intuitive and robust system response with both sparse event streams and regular sample streams. The system features synchronic updates of input values as well as intelligent refreshing of output values. Finally, the proposed signal model was examined in the context of audio analysis.

6. ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).

7. REFERENCES

[1] R.B Dannenberg and R. Bencina. Design patterns for real-time computer music systems. ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music, 2005.

[2] R.B. Dannenberg, C.L. Fraley, and P.Velikonja. Fugue: a functional language for sound synthesis. *Computer*, 24(7):36–42, 1991.

[3] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare. PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control. *Computer Music Journal*, 29(3):29–41, Fall 2005.

[4] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 2004.

[5] S.T. Pope and R.B. Dannenberg. Models and apis for audio synthesis and processing. ICMC 2007 Panel, 2007.

[6] M. Puckette. Combining event and signal processing in the max graphical programming environment. *Computer Music Journal*, 15(3):68–77, 1991.

[7] G. Wang, R. Fiebrink, and P.R. Cook. Combining analysis and synthesis in the chuck programming language. In *Proceedings of the 2007 International Computer Music Conference*, pages 35–42, Copenhagen, 2007.

P3 | INTRODUCING KRONOS – A NOVEL APPROACH TO SIGNAL PROCESSING LANGUAGES

Vesa Norilo. Introducing Kronos - A Novel Approach to Signal Processing Languages. In Frank Neumann and Victor Lazzarini, editors, *Proceedings of the Linux Audio Conference*, pages 9–16, Maynooth, 2011. NUIM

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

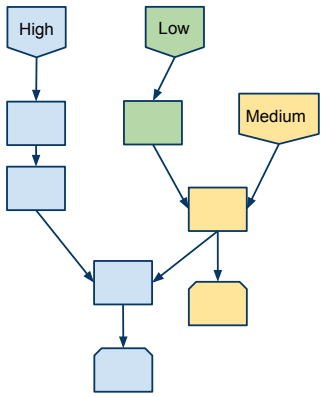

Figure 1: A reactive graph demonstrating spring priority. Processing nodes are color coded according to which spring triggers their update.

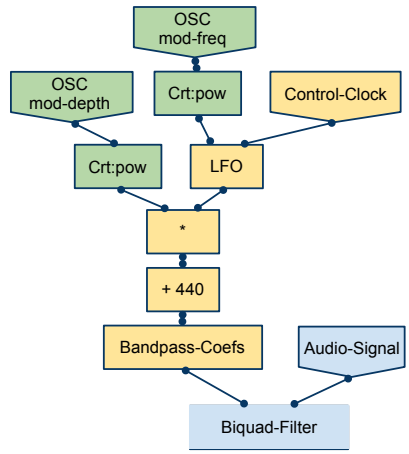

Figure 2: A practical example of a system consisting of user interface signals, coarse control rate processing and audio rate processing.

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.

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 particularly 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.

P4 | DESIGNING SYNTHETIC REVERBERATORS IN KRONOS

Vesa Norilo. Designing Synthetic Reverberators in Kronos. In *Proceedings of the International Computer Music Conference*, pages 96–99, Huddersfield, 2011

DESIGNING SYNTHETIC REVERBERATORS IN KRONOS

Vesa Norilo

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

ABSTRACT

Kronos is a special purpose programming language intended for musical signal processing tasks. The central aim is to provide an approachable development environment that produces industrial grade signal processors.

The system is demonstrated here in the context of designing and building synthetic reverberation algorithms. The classic Schroeder-Moorer algorithm is presented, as well as a feedback delay network, built with the abstraction tools afforded by the language. The resulting signal processors are evaluated both subjectively and in raw performance terms.

1. INTRODUCTION

The Kronos package consists of a language specification as well as an optimizing compiler. The compiler can be paired with several back ends. Currently the main focus is on a just in time compiler for the x86 architecture, while a C-language generator is also planned.

Syntactically Kronos is inspired by high level functional languages[1]. The syntax of functional languages is ideally suited for visualization; this match is demonstrated by another source of inspiration, Faust[6]. Eventually, Kronos aims to combine the ease of use and approachability of graphical environments like Pure Data[7] and PWGL[3] with the abstraction and rigour typical of functional programming languages.

Kronos programs are generic, meaning that signal processing blocks can be written once and used in any number of type configurations. For example, a digital filter could be designed without specifying single or double precision sample resolution or even if the data is numerically real or complex, monophonic or multichannel. The type system used in Kronos is more thoroughly discussed in[5].

When a Kronos patch is connected to a typed source, like an audio input, the patch is *specialized*. The generic algorithm description is type inferred[ref]. Following this, the execution is deterministic, which faciliates drastic compiler optimization[ref]. The system could be summarized as a type-driven code generator that produces highly optimized, statically typed code from high level, functional source code.

The rest of this paper is organized as follows. Section 2, *Implementing a Reverberator*, discusses the implementation of various primitives and algorithms in Kronos. Section 3, *Tuning*, discusses additions and enhancements to the basic algorithms. The results are evaluated in Section 4, *Evaluation*, before the paper's *Conclusion*, Section 5.

2. IMPLEMENTING A REVERBERATOR

Reverberation is a good example case, as the algorithms are straightforward yet complicated enough to stress the development environment and provide opportunities for utilization of several language features. They are also well suited for performance benchmarks.

2.1. Primitives for Reverberation

The example cases start with the primitives that are essential to synthetic reverberation. Delay lines, comb filters, and allpass filters will be examined.

2.1.1. Delay

While the functional programming paradigm intuitively matches the signal flow graph widely used for describing signal processing algorithms, there is an apparent clash. Functional programs do not support program *state*, or memory, a fundamental part of any processor with delay or feedback.

The solution to this problem offered by Kronos is a integrated delay operator. Called *rbuf*, short for a ring buffer, the operator receives three parameters: an initializer function, allowing the user to specify the contents of the ring buffer at the start of processing, the size of the buffer or delay time and finally the signal input. In Listing 1, a simple delay function is presented. This delay line is 10 samples long and is initialized to zero at the beginning.

Listing 1. Simple delay

```
Delay(sig)
{
  Delay = rbuf('0 #10 sig)
}
```

2.1.2. Comb filter

A delay line variant with internal feedback, a comb filter, is also widely used in reverberation algorithms. For this configuration, we must define a recursive connection. The *rbuf* operator allows signal recursion. As in all digital

systems, some delay is required for the recursion to be finitely computable. A delay line with feedback is shown in Listing 2. In this example, the symbol *output* is used as the recursion point.

Listing 2. Delay with feedback

```
Delay(sig fb delay)
{
  delayed = rbuf('0 delay sig + fb * delayed)
  Delay = delayed
}
```

2.1.3. Allpass-Comb

An allpass comb filter is a specially tuned comb filter that has a flat frequency response. An example implementation is shown in Listing 3, similar to the one described by Schroeder[8].

Listing 3. Allpass Comb filter

```
Allpass-Comb(sig fb delay)
{
  delayed = rbuf('0 delay sig - fb * delayed)
  Allpass-Comb = 0.5 * (sig + delayed + fb * delayed)
}
```

2.2. Multi-tap delay

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

Listing 4. Multi-tap delay

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

The processor described in Listing 4 can specialize to feature any number of delay lines. The well known higher order functions *Map* and *Reduce* define the functional language equivalent to a loop[1]. *Map* applies a caller supplied mapping function to all elements of a list. *Reduce* combines the elements of a list using caller-supplied reduction function.

In this example, another higher order function, *Curry*, is used to construct a new mapping function. *Curry* reduces the two argument *Delay* function into an unary function that always receives *sig* as the first argument. Curry is an elementary operator in combinatory logic.

This curried delay 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 finally summed, using *Reduce(Add ...)*. The remarkably short yet highly useful routine is a good example of the power of functional abstraction in Kronos.

A reader familiar with developing real time signal processing code might well be worried that such high level abstraction will adversely affect the performance of the resulting processor. Fortunately this is not the case, as the language is designed around constraints that allow the compiler to simplify all the complexity extremely well. Detailed performance results are shown in Section 4.

2.3. Schroeder Reverberator

Expanding upon the concepts introduced in Section 2.2, the classic diffuse field reverberator described by Schroeder can be implemented. Listing 5 implements the classic Schroeder reverberation[8]. Please refer to Section 4.1 for sound examples.

Listing 5. 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)
}
```

All the tuning parameters are adapted from Schroeder's paper[8]. The allpass parameters are constant regardless of reverberation time, while comb filter feedbacks are calculated according to the specified reverberation time. The comb section is produced similarily to the multi tap delay in Section 2.2. Since the delay function requires an extra feedback parameter, we utilize the *Zip-With* function, which is similar to *Map*, but expects a binary function and two argument lists. The combination of *Curry* and *Zip-With* generates a bank of comb filters, all fed by the same signal, but separately configured by the lists of feedback coefficients and delay times.

The series of allpass filters is realized by the higher order *Cascade* function. This function accepts a parameter cascading function, *Allpass-Comb*, signal input, *sig*, and a list of parameters, *allpass-params*. The signal input is passed to the cascading function along with the first element of the parameter list. The function iterates through the remaining parameters in *allpass-params*, passing the output of the previous cascading function along with the parameter element to each subsequent cascading function. Perhaps more easily grasped than explained, this has the effect of connecting several elements in series.

While the same effect could be produced with two nested calls to *Allpass-Comb*, this formulation allows tuning the allpass section by changing, inserting or removing parameters from the *allpass-params* list, with no further code changes, regardless of how many allpass filters are specified.

2.4. Feedback Delay Network Reverberator

Feedback delay network is a more advanced diffuse field simulator, with the beneficial property of reflection density increasing as a function of time, similar to actual acoustic spaces. The central element of a FDN algorithm is the orthogonal feedback matrix, required for discovering the lossless feedback case and understanding the stability criteria of the network. For a detailed discussion of the theory, the reader is referred to literature[2]f.

Listing 6. Basic Feedback Delay Network reverberator

```
Use Algorithm

Feedback-Mtx(input)
{
  Feedback-Mtx = input

  (even odd) = Split(input)
  even-mtx = Recur(even)
  odd-mtx = Recur(odd)

  Feedback-Mtx = Append(Zip-With(Add even-mtx odd-mtx)
          Zip-With(Sub even-mtx odd-mtx))
}

Basic(sig rt60)
{
  delay-times = (#1310 #1636 #1813 #1927)
  normalize-coef = -1. / Sqrt(Count(delay-times))
  loss-coefs = Map(Curry(Mul normalize-coef)
          Map(Curry(Feedback-for-RT60 rt60) delay-
              times))

  feedback-vector = z-1('(0 0 0 0) Zip-With(Mul loss-
      coefs Feedback-Mtx(delay-vector)))

  delay-vector = Zip-With(Delay Map(Curry(Add sig)
      feedback-vector) delay-times)

  Basic = Reduce(Add Rest(delay-vector))
}
```

In Listing 6, functional recursion is utilized to generate a highly optimized orthogonal feedback matrix, the Householder feedback matrix. The function *Feedback-Mtx* recursively calls itself, splitting the signal vector in two, computing element-wise sums and differences. This results in an optimal number of operations required to compute the Householder matrix multiplication[9].

Note that *Feedback-Mtx* has two return values; one of them simply returning the argument *input*. This is a case of parametric polymorphism[5], where the second, specialized form is used for arguments that can be split in two.

The feedback paths in this example are outside the bank of four delay lines. Instead, a simple unit delay recursion is used to pass the four-channel output of the delay lines through the feedback matrix and back into the delay line inputs. Because all the delay lines are fed back into all the others, the feedback must be handled externally.

The final output is produced by summing the outputs of all the delay lines except the first one, hence *Rest(delay-vector)*. The first delay line is skipped due to very prominent modes resulting from the characteristics of the Householder feedback.

3. TUNING

The implementations in Section 2 do not sound very impressive; they are written for clarity. Further tuning and a greater number of delay lines are required for a modern reverberator. The basic principles of tuning these two algorithms are presented in the following Sections 3.1 and 3.2. The full code and sound examples can be accessed on the related web page[4].

3.1. Tuning the Schroeder-Moorer Reverberator

A multichannel reverberator can be created by combining several monophonic elements in parallel with slightly different tuning parameters. Care must be taken to maintain channel balance, as precedence effect may cause the reverberation to be off-balance if the delays on one side are clearly shorter. Reflection density can be improved by increasing the number of comb filters and allpass filters while maintaining the basic parallel-serial composition. Frequency-dependant decay can be modeled by utilizing loss filters on the comb filter feedback path, and overrall reverberation tone can be altered by filtering and equalization.

The tuned example[4] built for this paper features 16 comb filters and 4 allpass filters for both left and right audio channel. Onepole lowpass filters are applied to the comb filter feedback paths and further to statically adjust the tonal color of reverberation.

3.2. Tuning the Feedback Delay Network Reverberator

Likewise, the number of delay lines connected in the feedback network can be increased. Frequency dependent decay is modelled similarily to the Schroeder-Moorer reverberator. Since a single network produces one decorrelated output channel for each delay line in the network, multichannel sound can be derived by constructing several different sums from the network outputs. Allpass filters can be used to further increase sound diffusion.

The tuned example[4] features 16 delay lines connected in a Householder feedback matrix. Each delay line has a lowpass damping filter as well as an allpass filter in the feedback path to improve the overrall sound. A static tone adjustment is performed on the input side of the delay network.

4. EVALUATION

Firstly, the results of implementing synthetic reverberators in Kronos is evaluated. Evaluation is attempted according to three criteria; how good the resulting reverberator sounds, how well suited was the Kronos language to program it and finally, the real time CPU performance characteristics of the resulting processor. The following abbreviations, in Table 1 are used to refer to the various processors described in this paper.

Key	Explanation
S4	Classic Schroeder reverberator, Section 2.3
S16	Tuned Schroeder reverberator [4]
FDN4	4-dimensional Feedback Delay Network, Section 2.4
FDN16	16-dimensional tuned FDN [4]

Table 1. Keys used for the processors

Key	Delays	Allpass	LPFs	Fmt	Time	CPU
S4	4	2	0	mono	6.9ms	0.12%
S16	32	8	33	stereo	89ms	1.5%
FDN4	4	0	0	mono	7.1ms	0.12%
FDN16	16	20	17	stereo	84ms	1.4%

Table 2. Features and performance of the processors

4.1. Sound

Sound quality is highly subjective measure; therefore, the reader is referred to the actual sound examples[ref]. Some observations by the author are listed here.

Unsuprisingly, *S4* is showing its age; the reverberation is rather sparse and exhibits periodicity. The modes of the four comb filters are also spread out enough that they are perceptible as resonances. However, the sound remains respectable for the computational resources it consumes.

FDN4 is also clearly not sufficient by itself. The diffuse tail is quite an improvement over *s-cl*, although some periodicity is still perceived. The main problem is the lack of diffusion in the early tail. This is audible as a sound resembling flutter echo in the very beginning of the reverberation.

S16 and *FDN16* both sound quite satisfying with the added diffusion, mode density and frequency dependant decay. *FDN16* is preferred by the author, as the mid-tail evolution of the diffuse field sounds more convincing and realistic, probably due to the increasing reflection density.

4.2. Language

It is our contention that all reverberation algorithms can be clearly and concisely represented by Kronos. Abstraction is used to avoid manual repetition such as creating all the delay lines one by one. The delay operator inherent in the language allows the use of higher order functions to create banks and arrays of delay lines and filters. In the case of the feedback delay network, a highly effective recursive definition of the Householder feedback matrix could be used.

4.3. Performance

The code produced by Kronos exhibits excellent performance characteristics. Some key features of the reverberators are listed in Table 2, along with the time it took to process the test audio.

A realtime CPU stress is computed by dividing the processing time with the play time of the audio, 5833 milliseconds in this test case. The processor used for the benchmark is an Intel Core i7 running at 2.8GHz. The CPU load caused by the algorithms presented ranges from 1.2 permil to 1.5 percent.

5. CONCLUSION

This paper presented a novel signal processing language and implementations of synthetic reverberation algorithms in it. The algorithms were then tuned and evaluated by both sound quality and performance criteria.

The presented algorithms could be implemented on a high level, utilizing abstractions of functional programming. Nevertheless, the resulting audio processors exhibit excellent performance characteristics.

Kronos is still in development into a versatile tool, to allow real time processing as well as export to languages such as C. Graphical user interface is forthcoming. Kronos is also going to be used as the next-generation synthesizer for the PWGL[3] environment. Interested parties are invited to contact the author should they be interested in implementing their signal processing algorithms in Kronos.

6. REFERENCES

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

[2] A. Jot Jean-Marc; Chaigne, "Digital Delay Networks for Designing Artificial Reverberators," in *Audio Engineering Society Convention 90*, 1991.

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

[4] V. Norilo, "ICMC2011 Examples," 2011. [Online]. Available: http://www.vesanorilo.com/kronos/icmc2011

[5] V. Norilo and M. Laurson, "A Method of Generic Programming for High Performance DSP," in *DAFx-10 Proceedings*, Graz, Austria, 2010, pp. 65–68.

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

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

[8] M. R. Schroeder, "Digital Simulation of Sound Transmission in Reverberant Spaces," *Journal of the Acoustical Society of America*, vol. 45, no. 1, p. 303, 1969.

[9] J. O. Smith, "A New Approach to Digital Reverberation Using Closed Waveguide Networks," 1985, pp. 47–53.

P5 KRONOS VST – THE PROGRAMMABLE EFFECT PLUGIN

Digital Audio Effects. Kronos Vst – the Programmable Effect Plugin. In *Proceedings of the International Conference on Digital Audio Effects*, Maynooth, 2013

Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-4, 2013

KRONOS VST – THE PROGRAMMABLE EFFECT PLUGIN

Vesa Norilo

Department of Music Technology
Sibelius Academy
Helsinki, Finland
vnorilo@siba.fi

ABSTRACT

This paper introduces Kronos VST, an audio effect plugin conforming to the VST 3 standard that can be programmed on the fly by the user, allowing entire signal processors to be defined in real time. A brief survey of existing programmable plugins or development aids for audio effect plugins is given. Kronos VST includes a functional just in time compiler that produces high performance native machine code from high level source code. The features of the Kronos programming language are briefly covered, followed by the special considerations of integrating user programs into the VST infrastructure. Finally, introductory example programs are provided.

1. INTRODUCTION

There are several callback-architecture oriented standards which allow third parties to extend conformant audio software packages. These extensions are colloquially called *plugins*. The plugin concept was popularized by early standards such as VST by Steinberg. This paper discusses a plugin implementation that conforms to VST 3. Other widely used plugin standards include Microsoft DirectX, Apple Audio Unit and the open source LADSPA. Pure Data[1] extensions could also be considered plugins.

As customizability and varied use cases are always encountered in audio software, it is no suprise that the plugin concept is highly popular. Compared to a complete audio processing software package, developing a plugin requires less resources, allowing small developers to produce specialized signal processors. The same benefit is relevant for academic researchers as well, who often demonstrate a novel signal processing concept in context in the form of a plugin.

The canonical way of developing a plugin is via C or C++. Since musical domain expertise is highly critical in developing digital audio effects, there is often a shortage of developers who have both the requisite skill set and are able to implement audio effects in C++. One way to address this problem is to develop a meta-plugin that implements some of the requisite infrastructure while leaving the actual algorithm to the end user, with the aim of simplifying the development process and bringing it within the reach of domain experts who are not necessarily professional programmers.

This paper presents Kronos VST, an implementation of the programmable plugin concept utilizing the Kronos signal processing language and compiler[2]. The plugin integrates the entire compiler package, and produces native machine code from textual source code while running inside a VST host, without an edit-compile-debug cycle that is required for C/C++ development.

The rest of the paper is organized as follows; Section 2, *Programmable Plugins and Use Cases*, discusses the existing implementations of the concept. Section 3, *Kronos Compiler Technology Overview*, briefly discusses the language supported by the plugin. Section 4, *Interfacing User Code and VST*, discusses the interface between the VST environment and user code. Section 5, *Conclusions*, summarizes and wraps up the paper, while some example programs are shown in Appendix A.

2. PROGRAMMABLE PLUGINS AND USE CASES

2.1. Survey of Programmable Plugins

2.1.1. Modular Synthesizers

Modular synthesizer plugins are arguably programmable, much as their analog predecessors. In this case, the user is presented with a set of synthesis units that can be connected in different configurations. A notable example of such a plugin is the *Arturia Moog Modular*.

Native Instruments Reaktor represents a plugin more flexible and somewhat harder to learn. It offers a selection of modular synthesis components but also ones that resemble programming language constructs rather than analog synthesis modules.

A step further is the Max/MSP environment by Cycling'74 in its various plugin forms. The discontinued *Pluggo* allowed Max/MSP programs to be used as plugins, while *Max for Live* is its contemporary sibling, although available exclusively for the Ableton Live software.

2.1.2. Specialist Programming Environments

In addition to modular synthesizers, several musical programming environments have been adapted for plugins. CSoundVST is a CSound[3] frontend that allows one to embed the entire CSound language into a VST plugin. More recently, Cabbage[4] is a toolset for compiling CSound programs into plugin format.

Faust[5], the functional signal processing language, can be compiled into several plugin formats. It has traditionally relied in part on a C/C++ toolchain, but the recent development of libfaust can potentially remove this dependency and enable a faster development cycle.

Cerny and Menzer report an interesting application of the commercial Simulink signal processing environment to VST Plugin generation[6].

Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-4, 2013

2.2. Use Cases for Programmable Plugins

The main differences between developing a plugin with an external tool versus supplying an user program to the plugin itself boil down to development workflow. The compilation cycle required for a developer to obtain audio feedback from a code change is particularily burdensome in the case of plugin development. In addition to the traditional edit-compile-run cycle, where compilation can take minutes, plugin development often requires the host program to be shut down and restarted, or at least forced to rescan and reload the modified plugin file.

In contrast, if changes can be made on the fly, while the plugin is running, the feedback cycle is almost instantaneous. This is what the KronosVST plugin aims to do. Several use cases motivate such a scheme;

2.2.1. Rapid Prototyping and Development

Rapid prototyping traditionally means that the program is initially developed in a language or an environment that focuses primarily on developer productivity. In traditional software design, this can mean a scripting language that is developer friendly but perhaps not as performant or capable as C/C++. In the case of audio processors, rapid prototyping can take place in, for example, a graphical synthesis environment or a programmable plugin. Once the prototyping is complete, the product can be rewritten in C/C++ for final polish and performance.

2.2.2. Live Coding

Live coding is programming as a performance art. In the audio context, the audience can see the process of programming as well as hear the output in real time. The main technical requirement for successful live coding is that code changes are relatively instantaneous. Also, the environment should be robust to deal with programming errors in a way that doesn't bring the performance to a halt. KronosVST aims to support live coding, although the main focus of this article is rapid development.

2.3. Motivating Kronos VST

As programming languages evolve, it becomes more conceivable that the final rewrite in a low level language like C may no longer be necessary. This is one of the main purposes of the Kronos project. Ideally, the language should strike a correct balance of completeness, capability and performance to eliminate the need to drop down to C++ for any of these reasons.

The benefit of this approach is a radical improvement in developer productivity – but the threat is, as always, that the specialist language may not be good enough for every eventuality and that C++ might still be needed.

The main disincentive for developers to learn a new programming language is the perception that the time invested might not yield sufficient benefits. Kronos VST aims to present the language in a manner where interested parties can quickly evaluate the system and its relative merit, look at example programs and audition them in context.

3. KRONOS TECHNOLOGY OVERVIEW

This section presents a brief overview of the technology behind KronosVST. For detailed discussion on the programming language, the reader is referred to previous work [7] [8] [2].

3.1. Programming Language

Kronos as a programming language is a functional language[9] that deals with signals. From existing systems, Faust[5] is likely the one that it resembles the most. Both systems feature an expressive syntax and compilation to high performance native code. In the recent developments, both have converged on the LLVM[10] backend which provides just in time and optimization capabilities.

As the main differentiators, Kronos aims to offer a type system that extends the metaprogramming capabilities considerably[8]. Also, Kronos offers an unified signal model[7] that allows the user to deal with signals other than audio. Recent developments to Faust enhance its multirate model[**?**], but event-based streams remain second class. Kronos is also designed, from the ground up, for compatibility with visual programming.

On the other hand, Faust is a mature and widely used system, successfully employed in many research projects. In comparison, Kronos is still quite obscure and untested.

3.2. Libraries

The principle behind Kronos is that there are no built-in unit generators. The signal processing library that it comes with is in source form and user editable. By extension, it means that the library components cannot rely on any "magic tricks" with special compiler support. User programs are first class citizens, and can supplant or completely replace the built-in library.

Also due to the nature of the optimizing compiler built into Kronos, the library can remain simpler than most competing solutions. Functional polymorphism is employed so that signal processing components can adapt to their context. It supports generic programming, which enables a single processor implementation to adapt and optimize itself to various channel configurations and sample formats. With a little imagination this mechanism can be used to achieve various sophisticated techniques – facilities such as *currying* and *closures* in the standard library are realized by employing the generic capabilities of the compiler.

As Kronos is relatively early in its development, the standard library is continuously evolving. At the moment it provides functional programming support for the map-reduce paradigm as well as fundamentals such as oscillators, filters, delay elements and interpolators.

3.3. Code Generation

Kronos is a Just in Time compiler[11] that performs the conversion of textual source code to native machine code, to be immediately executed. In the case of a plugin version, the plugin acts as the compiler driver, feeding in the source code entered via the plugin user interface and connecting the resulting native code object to the VST infrasturcture.

3.3.1. Recent Compiler Developments

The standalone Kronos compiler is currently freely available in its beta version. This version features compilation and optimization of source code to native x86 machine code or alternatively translation into C++.

Currently, the compiler is being rewritten, with focus on compile time performance. The major improvement is in the case of

Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-4, 2013

extended multirate DSP, where various buffering techniques can be employed. The language semantics seamlessly support cases where a signal frame is anything from a single sample to a large buffer of sound, but the compile time could become unacceptable as frame size was increased. The major enhancement in the new compiler version is the decoupling of vector size and compilation time, resulting from a novel redundancy algorithm in the polymorphic function specializer.

The design of the compiler is also revised and simplified, aiming to an eventual release of the source code under a free software license. As the code generator backend, the new version relies on LLVM[10] for code generation instead of a custom x86 solution; greatly increasing the number of available compile targets.

3.3.2. Optimization and Signal Rate Factorization

The aim of the Kronos project is to have the source code to look like the way humans think about signal processing, and the generated machine code to perform like that written by a decent developer. This is the goal of most compiler systems, but very hard to accomplish, as in most systems the developer needs to intervene on relatively low level of abstraction to enforce that the generated machine code is close to optimal.

Kronos aims to combine high level source code with high performance native code. The programs should be higher level than C++ to make the language easier for musicians, as well as faster to write. However, if the generated code is significantly slower, a final rewrite in C++ might still be required, defeating the purpose of rapid development.

The proposed solution is to narrow down the capabilities of the language to fulfill the requirements of signal processor development as narrowly as possible. The Kronos language is by design statically typed, strictly side effect free and deterministic, which is well suited for signal processing. This allows the compiler to make a broad range of assumptions about the code, and apply transformations that are far more radical than the ones a C++ compiler can safely do.

A further important example of DSP-specific optimization is the multirate problem. Languages such as C++ require the developer to specify a chronological order in which the program executes. In the case of multirate signal processing, this requires manual and detailed handling of various signal processors that update synchronously or in different orders.

As a result, many frameworks gloss over the multirate problem by offering a certain set of signal rates from which the user may – and has to – choose from. Traditionally, this manifests as similar-but-different processing units geared either for control or audio rate processing, or maybe handling discrete events such as MIDI. This increases the *vocabulary* an user has to learn, and makes signal processing libraries harder to maintain.

Kronos aims to solve the multirate problem, combined with the event handling problem, by defining the user programs as having no chronology. This is inherent to the functional programming model. Instead of time, the programs model data flow; data flow between processing blocks is essentially everything that signal processing boils down to.

Each data flow is semantically synchronous. Updates to the inputs of the system trigger recomputation of the results that depend on them, with the signal graph being updated accordingly. Special delay primitives in the language allow signal flow graphs to connect to previous update frames and provide for recursive loops and delay effects.

Since the inputs and the data flows that depend on them are known, the compiler is able to factorize user programs by their inputs. It can produce update entry points that respond to a certain set of system inputs, and optimize away everything that depends on inputs outside of the chosen set. Each system input then becomes an entry point that activates a certain subset of the user program – essentially, a clock source.

Because the code is generated on the fly, this data flow factorization has no performance impact, which renders it suitable to use at extreme signal rates such as high definition audio as well as sparse event streams such as MIDI. Both signal types become simple entry points that correspond to either an audio sample frame or a MIDI event.

3.3.3. Alternative Integration Strategies

In addition to plugin format, the Kronos compiler is available as a C++-callable library. There is also a command line compile server that responds to OSC[12] commands and is capable of audio i/o. The main purpose for the compile server is to act as a back end for a visual patching environment.

The compiler generates code modules that implement an object oriented interface. The user program is compiled into a code module with a set of C functions, covering allocation and initialization of a signal processor instance, as well as callbacks for plugging data into its external inputs and triggering various update routines. It is also possible to export this module as either LLVM[10] intermediate representation or C-callable object code.

4. INTERFACING USER CODE AND VST

To facilitate easy interaction between user code and the VST host application, various VST inputs and outputs are exposed as user-visible Kronos functions. These functions appear in a special package called *IO*, which the plugin generates according to the current processing context. The user entry point is a function called *Main*, which is called by the base plugin to obtain a frame of audio output.

4.1. Audio I/O

A VST plugin can be used in a variety of different audio I/O contexts. The VST3 standard allows for any number of input and output buses to and from the plugin. Each of these buses is labeled for semantic meaning and can contain an arbitrary number of channels.

The typical use for multiple input buses is to allow for sidechain input to a plugin. Multiple output buses, on the other hand, can be used to inform the host that multiple mixer channels could be allocated for the plugin output. The latter is mostly used in the context of instrument plugins.

The Kronos VST plugin exposes the main input bus as a function called *IO:Audio-In*. The return type of this function is a tuple containing all the input channels to the main bus of the plugin. The sidechain bus is exposed as *IO:Audio-Sidechain*. Both functions act as *external inputs* to the user program, propagating updates at the current VST sample rate.

Currently, only a single output bus is supported. The channel count of the output is automatically inferred from the *Main* function.

Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-4, 2013

4.1.1. Audio Bus Metadata

Programs that need to know the update interval of a given data flow can interrogate it with the Kronos reactive metadata system. The sample rate of the data flow in question becomes another external input to the program with its own update context.

The Kronos VST plugin supplies update interval data for the audio buses to the user program. On sample rate changes, the reactive system can automatically update any computation results that depend on it.

4.1.2. Multichannel Datatype

Polymorphism within the Kronos VST plugin allows user programs to be flexible in their channel configuration. Many signal processors have implementations that do not vary significantly on the channel count. A processor such as an equalizer would just contain a set of identical filter instances to process a bundle of channels.

For such cases, the Kronos VST library comes with a data type that represents a multichannel sample frame. An atom of this type can be constructed from a tuple of samples by a call to *Frame:Cons*, which packages any number of channels into a single frame. These frames have arithmetic with typical vector semantics; operations are carried out for each element pair for matching multichannel frames.

The *Frame* type also has an upgrade coercion semantic. There is a specialization of the *Implicit-Coerce* function that can promote a scalar number into a multichannel duplicate. The Kronos runtime library widely calls the implicit coercion function to resolve type mismatches. This means that the compiler is able to automatically promote a scalar to a multichannel type. For example, whenever a multichannel frame is multiplied by a gain coefficient, each channel of the frame is processed without explicit instructions from the user program.

Because Kronos programs have only implicit state, this extension carries over to filter- and delay-like operations. The compiler sees a delay operation on a multichannel frame and allocates state accordingly for each channel. Therefore, the vast majority of algorithms can operate on both monophonic samples and multichannel frames without any changes to the user code.

4.2. User Interface

The VST user interface is connected to the user program via calls to *IO:Parameter*. This function receives the parameter label and range. The *IO* package constructs external inputs and triggers that are uniquely identified by all the parameter metadata, which allows for the base plugin infrastructure to read back both parameter labels and ranges.

Any external input in the user code that has the correct label and range metadata attached is considered a parameter by the base plugin. At the moment, each parameter is assigned a slider in the graphical user interface. In the future, further metadata may be added to support customizing the user interface with various widgets such as knobs or XY-pads. An example user interface is shown in Figure 1.

The parameters appear as external inputs from the user code perspective, and work just like the audio input, automatically propagating a signal clock that ticks whenever a user interaction or sequencer automation causes the parameter value to be updated.

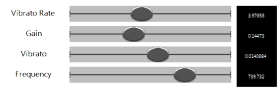

Figure 1: *An Example of a Generated VST Plugin Interface*

However, the parameters are assigned a lower reactive priority than the audio. Any computations that depend on both audio and parameter updates ignore the parameters and lock solely to audio clock.

This prevents parameter updates from causing additional output clock ticks – in effect, the user interface parameters terminate inside the audio processor at the point where their data flow merges with the audio path. This is analogous to how manually factored programs tend to cache intermediate results such as filter coefficients that result from the user interface and are consumed by the audio processor.

4.3. MIDI

MIDI input is expressed as an event stream, with a priority between parameters and audio. Thus, MIDI updates will override parameter updates but submit to audio updates.

The MIDI stream is expressed as a 32-bit integer that packs the three MIDI bytes. Accessor functions *MIDI:Event:Status()*, *MIDI:Event:A()* and *MIDI:Event:B()* can be called to retrieve the relevant MIDI bytes.

MIDI brings up a relevant feature in the Kronos multirate system; dynamic clock. MIDI filtering can be implemented by inhibiting updates that do not conform to the desired MIDI event pattern. The relevant function is *Reactive:Gate(filter sig)* which propagates the signal *sig* updates if and only if *filter* is true. A series of Gates can be used to deploy different signal paths to deal with note on, note off and continuous controller events. Later, the updates can be merged with *Reactive:Merge()*.

5. CONCLUSIONS

This paper presented an usage scenario for *Kronos*, a signal processing language. Recent developments in Kronos include a compiler rewrite from scratch. Kronos VST, a programmable plugin, is the first public release powered by the new version.

The programmable plugin allows an user to deploy and modify a signal processing program inside a digital audio workstation while it is running. It is of interest to programmers and researchers attracted to rapid prototyping or development of audio processor plugins. The instant feedback is also useful to live coders.

The Kronos VST plugin is designed to stimulate interest in the Kronos programming language. As such, it is offered free of charge to interested parties. The plugin can be used as is, or as a development tool – a finished module may be exported as C-callable object code, to be integrated in any development project.

A potential further development is an Apple Audio Unit version. The host compatibility of the plugin will be enhanced in extended field tests. Pertaining to the mainline Kronos Project, the libraries shipped with the plugin as well as the learning materials are under continued development. As the plugin and the compiler technology are very recent, the program examples at this point are

Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland, September 2-4, 2013

introductory. More sophisticated applications are forthcoming, to better demonstrate the capabilities of the compiler.

A. EXAMPLE PROGRAMS

A.1. Tremolo Effect

Listing 1: *Tremolo Source*

```
Saw(freq) {
    inc = IO:Audio-Clock(freq / IO:Audio-Rate())
    next = z-1(0 wrap + inc)
    wrap = next - Floor(next)
    Saw = 2 * wrap - 1
}

Main() {
  freq = IO:Parameter("Tremolo Freq" #0.1 #5 #20)
  (l r) = IO:Audio-In()
  gain  = Abs(Saw(freq))
  Main  = (l * gain r * gain)
}
```

A.2. Parametric EQ

Listing 2: *Parametric EQ Source*

```
/* Coefficient computation routine 'EQ-Coefs' omitted
    for brevity */
EQ-Band(x0 a0 a1 a2 b1 b2) {
  y1 = z-1(init(x0 #0) y0)
  y2 = z-1(init(x0 #0) y1)
  y0 = x0 - b1 * y1 - b2 * y2
  EQ-Band = a0 * y0 + a1 * y1 + a2 * y2
}

EQ-Params(num) {
  EQ-Params = (
    IO:Parameter(String:Concat("Gain " num) #-12 #0 #12)
    IO:Parameter(String:Concat("Freq " num) #20 #2000
        #20000)
    IO:Parameter(String:Concat("Q " num) #0.3 #3 #10))
}

Main() {
  input  = Frame:Cons(IO:Audio-In())
  params = Algorithm:Map(band => EQ-Coefs(EQ-Params(band
      )) [#1 #2 #3 #4])
  Main   = Algorithm:Cascade(Fitler:Biquad input params)
}
```

A.3. Reverberator

Listing 3: *Simple Mono Reverb*

```
RT60-Fb(delay rt60) {
  RT60-Fb = Crt:pow(0.001 delay / rt60)
}

Main() {
  Use Algorithm  /* for Map and Reduce */
  /* simplification: input is mono sum */
  input = Reduce(Add IO:Audio-In())
  rt60 = IO:Parameter("Reverb Time" #0.1 #3 #10) * IO:
      Audio-Rate()
  mix  = IO:Parameter("Mix" #0 #0.5 #1)

  /* settings adapted from the Schroeder paper */
  allpass-params = [(0.7 #221) (0.7 #75)]
  delay-times = [#1310 #1636 #1813 #1927]

  /* compute feedbacks and arrange delay line params */
  delay-params = Map(d => (d RT60-Fb(d rt60))
                     delay-times)

  /* compute parallel comb section */
  comb-sec  = Map((dl fb) => Delay(input dl fb)
                  delay-params)

  /* mono sum comb filters and mix into input */
  sig = (1 - mix) * input +
        mix * Reduce(Add comb-sec) / 4

  Main = (sig sig)
}
```

B. REFERENCES

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

[2] Vesa Norilo, "Introducing Kronos - A Novel Approach to Signal Processing Languages," in *Proceedings of the Linux Audio Conference*, Frank Neumann and Victor Lazzarini, Eds., Maynooth, Ireland, 2011, pp. 9–16, NUIM.

[3] Richard Boulanger, *The Csound Book*, vol. 309, MIT Press, 2000.

[4] Rory Walsh, "Audio Plugin development with Cabbage," in *Proceedings of the Linux Audio Conference*, Maynooth, Ireland, 2011, pp. 47–53, Linuxaudio.org.

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

[6] Robert Cerny and Fritz Menzer, "Convention e-Brief The Audio Plugin Generator: Rapid Prototyping of Audio DSP Algorithms," in *Audio Engineering Society Convention*, 2012, vol. 132, pp. 3–6.

[7] Vesa Norilo and Mikael Laurson, "Unified Model for Audio and Control Signals," in *Proceedings of ICMC*, Belfast, Northern Ireland, 2008.

[8] Vesa Norilo and Mikael Laurson, "A Method of Generic Programming for High Performance DSP," in *DAFx-10 Proceedings*, Graz, Austria, 2010, pp. 65–68.

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

[10] C Lattner and V Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization 2004 CGO 2004*, vol. 57, no. c, pp. 75–86, 2004.

[11] J Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[12] Matthew Wright, Adrian Freed, and Ali Momeni, "OpenSound Control: State of the Art 2003," in *Proceedings of NIME*, Montreal, 2003, pp. 153–159.

P6 | RECENT DEVELOPMENTS IN THE KRONOS PROGRAMMING LANGUAGE

Vesa Norilo. Recent Developments in the Kronos Programming Language. In *Proceedings of the International Computer Music Conference*, Perth, 2013

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

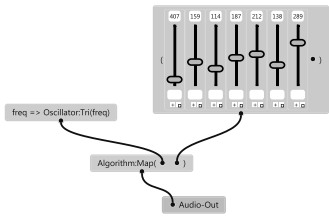

Figure 1. Mapping a set of sliders into an oscillator bank

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.

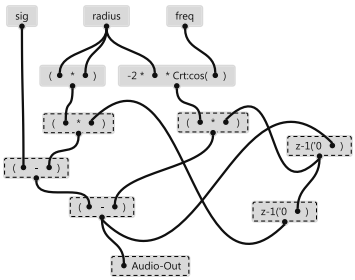

Figure 2. A Filter with ambigious clock sources

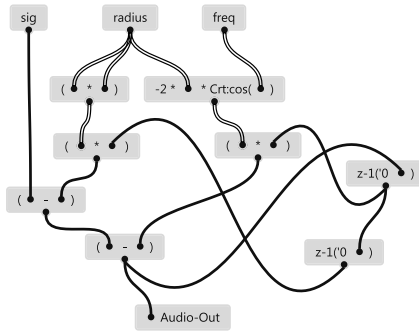

Figure 3. A Filter with clocking ambiguity resolved

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.

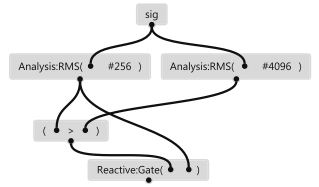

Figure 4. Dynamic clock from a Transient Detector

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 × 3	X				X				X		
Clock × 4	X			X			X			X		

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*

3.3.3. 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.

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 overall 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.

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*

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.

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.

Part III

Appendices

A LANGUAGE REFERENCE

This chapter is intended to enumerate and explain the primitive syntactic constructs in the Kronos language, as well as to cover the functions supplied with the compiler in source form.

A.1 SYNTAX REFERENCE

This section explains the structure and syntax of a program in the Kronos language.

A.1.1 Identifiers and Reserved Words

An identifier is a name for either a *function* or a *symbol*. Kronos identifiers may contain alphabetical characters, numbers and certain punctuation. The first character of an identifier must not be a digit. In most cases it should be an alphabetical character. Identifiers beginning with punctuation are treated as infix functions.

Identifiers are delimited by whitespace, commas or parentheses of any kind. Please note that punctuation does not delimit symbols; as such, `a+b` is a single symbol, rather than three.

Identifiers may be either defined in the source code or be reserved for specific purpose by the language. Please refer to Table 3 for a summary of the reserved words.

A.1.2 Constants and Literals

Number types

Constants are numeric values in the program source. The standard decimal number is interpreted as a 32-bit floating point number. Different number types can be specified with suffixes, as listed in Table 4.

Invariants

In addition, numeric constants can be given as *invariants*. This is a special number that is lifted to the type system. That is, every invariant number has a distinct type. Invariant numbers carry no runtime data. Due to type determinisim, this is the only kind of number that can be used to direct program flow. Invariants are prefixed with the hash tag, such as `#2.71828`.

Invariant Strings

Kronos strings are also lifted to the type system. Each unique string thus has a distinct type. This allows strings to be used to direct program flow. They do not contain runtime data. Strings are written in double quotes, such as `"This is a string"`

Table 3: Reserved Words in the Kronos Parser

word	reserved for
arg	tuple of arguments to current function
Break	remove type tag from data
cbuf	ring buffer, returns buffer content
Let	bind a symbol dynamically
Make	attach type tag to data
Package	Declaring a namespace
rbuf	ring buffer, returns overwritten slot
rcbuf	ring buffer, returns overwritten and buffer
rcsbuf	ring buffer, returns overwritten, index and buffer
Type	Declaring a type tag
Use	Search for symbols in package
When	explicit overload resolution rule
z-1	Unit delay

Table 4: Numeric constant types

example	suffix	description
3i	i	3 as a 32-bit integer
3.1415d	d	3.1415 as a 64-bit floating point number
9q	q	9 as a 64-bit integer

Some special characters can be encoded by escape notation. The escape sequences are listed in Table 5.

A.1.3 Symbols

A symbol is an identifier that refers to some other entity. Symbols are defined by equalities:

```
My-Number = 3
```

```
My-Function = x => x * 10
```

Table 5: String escape sequences

escape sequence	meaning
\n	new line
\t	tabulator
\r	carriage return
\v	backspace
\\	single backslash

Subsequently, there is no difference between invoking the symbol or spelling out the expression assigned to it. Because symbols are immutable, there is no concept of the bound value ever changing.

A.1.4 Functions

Functions can be bound to symbols via the lambda arrow:

```
Test = x => x + 5
```

Or the compound form:

```
Test(x) { Test = x + 5 }
```

The compound form allows multiple definitions of the function body, and these will be treated as polymorphic. Unlike normal symbols, compound definitions of the same function from multiple source code units are merged. This allows code units to extend a function that was defined in a different unit.

The compound form can only appear inside packages, while the lambda arrow is an expression and thus more flexible – it can be used to define nested functions. An example of both function forms is given in Listing 6.

Listing 6: Compound function and Lambda arrow

```
My-Fold(func data) {
  My-Fold = data

  (x xs) = data
  My-Fold = func(x My-Fold(func xs))
}

Main() {
  Main = My-Fold((a b) => (b a)
           1 2 3 4 5)

}
; Outputs ((((5 4) 3) 2) 1)
```

A.1.5 Packages

A Package is an unit of organization that conceptually contains parts of a Kronos program. These parts can be functions or symbols. The packaging system provides a unique, globally defined way to refer to these functions or symbols.

Symbols defined in the global scope of a Kronos program reside in the root namespace of the code repository. They are visible to all scopes in all programs. The Listing 7 illustrates nested packages and name resolution.

Listing 7: Packages and symbol lookup

```
Global-Symbol = 42

Package Outer {
  Package Inner {
    Bar() {
      Bar = Global-Symbol
    }
  }

  Baz() {
```

```
    Baz = Inner:Bar()
  }
}

Quux() {
  Quux = (Outer:Baz() Outer:Inner:Bar())
}
```

Scope

Scope is a context for symbol bindings within the program. By default, symbols are only visible to the expressions within the scope. Only a single binding to any given symbol is permitted within a scope. The compound form of a function is an exception: multiple compound forms are always merged into one, polymorphic definition.

A.1.6 Expressions

Expressions represent computations that have a value. Expressions consist of Symbols (A.1.3), Constants (A.1.2) and Function Calls. The simplest expression is the tuple; an ordered grouping of Expressions.

Tuples

Tuples are used to bind multiple expressions to a single symbol. Tuples are denoted by parentheses, enclosing any number of Expressions. The tuple is encoded as a chain of ordered pairs. This is demonstrated in the Listing 8, in which the expressions `a` and `b` are equal in value.

Listing 8: Tuples and chains of pairs

```
a = (1 2 3 4)
b = Pair(1 Pair(2 Pair(3 4))
; a and b are equivalent
```

Expressions within a tuple can also be tuples. Binding a symbol to multiple values via a tuple is called Structuring.

DESTRUCTURING A TUPLE

Destructuring is the opposite of structuring. Kronos allows a tuple of symbols on the left hand side of a binding expression. Such tuples may only contain Symbols or similar nested tuples. This is demonstrated in Listing 9.

Listing 9: Destructuring a tuple

```
my-tuple = (1 2 3 4 5)
a1 = First(my-tuple)
a2 = First(Rest(my-tuple))
a3 = First(Rest(Rest(my-tuple)))
as = Rest(Rest(Rest(my-tuple)))
(b1 b2 b3 bs) = my-tuple
; b1, b2, b3 and bs are equivalent to a1, a2, a3 and as
```

Destructuring proceeds by splitting the right hand side of the definition recursively, according to the structure of the left hand side. Each symbol on the left hand side is then bound to the corresponding part of the right hand side.

Lists

Kronos lists are tuples that end in the `nil` type. This results in semantics that are similar to many languages that use lists. The parser offers syntactic sugar for structuring lists. Please see Listing 10.

Listing 10: List notation

```
a = (1 2 3 4 nil)
b = [1 2 3 4]
; a and b are equivalent
```

Usage of nil terminators and square brackets is especially advisable when structuring extends to multiple levels. In Listing 11, symbols `a` and `b` are identical, despite different use of parenthesis. Symbols `c` and `d` are not similarly ambiguous. As a rule of thumb, ambiguity is possible whenever several tuples end at the same time. This never arises in list notation, as the lists always end in `nil`.

Listing 11: Disambiguation of Nested Tuples

```
a = ((1 2) (10 20) (100 200))
b = ((1 2) (10 20) 100 200)
; a and b are equivalent; this is confusing
c = [(1 2) (10 20) (100 200)]
d = [(1 2) (10 20) 100 200]
; c and d are not equivalent
```

Function Call

A symbol followed by a tuple, with no intervening whitespace, is considered a function call. The tuple becomes the argument of the function.

The symbols can be either local to the scope of the function call or globally defined in the code repository. A local function call is shown in Listing 12.

Listing 12: Calling a local function

```
add-ten = x => x + 10
y = add-ten(5)
; y is 15
```

Infix Functions

Infix functions represent an alternative function call syntax. They are used to enable standard notation for common binary functions like addition and multiplication. Kronos features only left associative binary infix functions, along with a special ternary operator for pattern matching.

Symbols that start with a punctuation character are considered infix functions by default. Section A.1.6 explains how to change this behavior.

The parser features a set of predefined infix functions that map to a set of binary functions. These infices also have a well defined standard operator precedence. They are listed in Table 6 in order of descending precedence. In addition, it is possible to use arbitrary infix functions: these always have a precedence that is lower than any of the standard infices. Their internal precedence is based on the first character of the operator. They are divided into four groups based on the initial character; the rest of the characters are arbitrary. The initial characters of each group are listed in Table 6.

Table 6: Predefined infix functions

Infix	Calls	Description
/	:Div	arithmetic division
*	:Mul	arithmetic multiplication
+	:Add	arithmetic addition
-	:Sub	arithmetic substraction
==	:Equal	equality test
!=	:Not-Equal	non-equality test
>	:Greater	greater test
<	:Less	less test
>=	:Greater-Equal	greater or equal test
<=	:Less-Equal	less or equal test
&	:And	logical and
||	:Or	logical or
=>		lambda arrow
<<		bind right hand side to _ on left hand side
>>		bind left hand side to _ on right hand side
*/+-		custom infices group 1
?!=<>		custom infices group 2
||&%		custom infices group 3
.: ~^		custom infices group 4

Unary Quote

Prepending an expression with the quote mark `'` causes the expression to become an anonymous function. The undefined symbol `_` within the expression is bound to the function call argument tuple. As an example, $f(x) = 2^x$ can be written as an anonymous function; `'Math:Pow(2 _)`

Section

Parentheses can be used to enforce partial infix expressions, which are called sections. These become partial applications of the infices. For example, `(* 3)` is an anonymous function that multiplies its argument by three. If one side is omitted, the anonymous function is unary. If both sides are omitted, the anonymous function is binary. This syntax is similar to Haskell.

Custom Infices

A symbol that begins with punctuation, but is not any of the predefined infices listed in Table 6, is considered a custom infix operator. It has the lowest precedence. During parsing, such an operator is converted into a function call by prepending "Infix" to the symbol. For example, a custom infix `a +- b` is converted to `Infix+-(a b)`. Note that while the predefined infices refer to symbols in the root namespace, custom infices follow the namespace lookup rules of their enclosing scope. This allows constraining custom infix operators to situations where code is either located in or refers to a particular package, reducing the risk of accidental use and collisions.

Table 7: Delay line operators in Kronos

Operator	Arguments	Returns
z-1	(init sig)	prev-sig
rbuf	(init order sig)	delayed-sig
cbuf	(init order sig)	buffer
rcbuf	(init order sig)	(buffer delayed-sig)
rcsbuf	(init order sig)	(buffer index delayed-sig)

Infixing notation

A normal function can be used as an infix function by enclosing its name in backticks. Such in situ infices always have the lowest precedence. For example, `3 + 4`, `Add(3 4)` and `3 'Add'4` are equivalent apart from precedence considerations.

Delays and Ring Buffers

Delays and ring buffers are operators that represent the state and memory of a signal processor. Their syntax resembles that of a function, but they are not functions – they cannot be assigned to symbols directly. The reason for this is that these operators enable cyclic definitions. That is, the signal argument to a delay or ring buffer operator can refer to symbols that are only defined later.

There are multiple versions of delays for different common situations. They all share some characteristics, such as having two signal paths, one for initialization and the other one for feedback. The initializer path also decides the data type of the delay line.

An overview of the delay line operators is given in Table 7. The `init` argument is evaluated and used to initialize the delay line contents for new signal processor instances. For the higher order delays, the `order` argument is an invariant constant that determines the length of the delay line. The initialization value is replicated to fill out the delay line. The `sig` argument determines the reactivity – clock rate – of the delay operator. This clock rate is propagated to the output of the operator.

Most delay operators output the delayed version of the input signal. The delay amount is fixed at the order of the operator; one sample for the unit delay operator, and `order` for the higher order operators. Variable delays and multiplexing can be accomplished by the delay line operators that output the entire contents of the `buffer`, along with the `index` of the next write.

It is to be noted that all reads from a delay line always happen before the incoming signal overwrites delay line contents.

Select and Select-Wrap

The selection operators provide variable index lookup into a homogenic tuple or list. If the source tuple is not homogenic, in that it contains elements of different types, both selection operators produce a type error. An exception is made for lists; a terminating `nil` type is allowed for a homogenic list, but cannot be selected by the selection operators.

`Select` performs bounds clamping for the index; indices less or equal to zero address the first element, while indices pointing past the end of the tuple will be constrained to the last element. `Select-Wrap` performs modulo arithmetic; the tuple is indexed as an infinite cyclic sequence,

Table 8: Selection operators

Operator	Arguments	Returns
Select	(vector index)	element-at-index
Select-Wrap	(vector index)	element-at-index

Table 9: Reactive operators

Operator	Args	Retns	Description
Tick	(priority id)	nil	provides a reactive root clock with the supplied 'id' and 'priority'
Resample	(sig clock)	sig	'sig' now updates at the rate of 'clock' signal
Gate	(sig gate)	sig	any updates to 'sig' will be inhibited while 'gate' is zero
Merge	tuple	atom	outputs the most recently changed element in homogenic 'tuple'
Upsample	(sig multiplier)	sig	output signal is updated 'multiplier' times for every update of 'sig'
Downsample	(sig divider)	sig	output signal is updated once for every 'divider' updates of 'sig'
Rate	sig	rate	returns the update rate of 'sig' as a floating point value

with the actual data specifying a single period of the cycle. A summary of the selection operators is given in Table 8.

A.1.7 Reactive Primitives

Reactive primitives are operators that are transparent to data and values, and guide the signal clock propagation instead. All the reactive operators behave like regular functions in the Kronos language, but their functionality can't be replicated by user code.

Most primitives take one or more signals, manipulating their clocks in some way. They can be used to override the default clock resolution behavior, where higher priority signal clocks dominate lower priority signal clocks. An overview of all the primitives is given in Table 9.

A.2 LIBRARY REFERENCE

Algorithm

The `Algorithm` package provides functional programming staples: higher order functions that recurse over collections, providing projections and reductions. This package covers most of the functionality that imperative programs would use loops for.

ACCUMULATE `Algorithm:Accumulate(func set...)`

Produces a list where each element is produced by applying 'func' to the previously produced element and an element from 'set'.

CONCAT `Algorithm:Concat(as bs)`

Prepends the list 'as' to the list 'bs'

EVERY `Algorithm:Every(predicate set...)`

return #true if all elements in 'set' are true according to 'predicate'

EXPAND `Algorithm:Expand(count iterator seed)`

Produces a list of 'count' elements, starting with 'seed' and generating the following elements by applying 'iterator' to the previous one.

FILTER `Algorithm:Filter(predicate set)`

Evaluates 'predicate' for each element in 'set', removing those elements for which nil is returned.

FLAT-FIRST `Algorithm:Flat-First(x)`

Returns the first element in 'x' regardless of its algebraic structure.

FOLD `Algorithm:Fold(func set...)`

Folds 'set' by applying 'func' to the first element of 'set' and a recursive fold of the rest of 'set'.

ITERATE `Algorithm:Iterate(n func x)`

Applies a pipeline of 'n' 'func's to 'x'.

MAP `Algorithm:Map(func set...)`

Applies 'func' to each element in 'set', collecting the results.

MULTI-MAP `Algorithm:Multi-Map(func sets...)`

Applies a polyadic 'func' to a tuple of corresponding elements in all of the 'sets'. The resulting set length corresponds to the smalles input set.

REDUCE `Algorithm:Reduce(func set...)`

Applies a binary 'func' to combine the first two elements of a list as long as the list is more than one element long.

SOME `Algorithm:Some(predicate set...)`

return #true if some element in 'set' is true according to 'predicate'

UNZIP `Algorithm:Unzip(set...)`

Produces a pair of lists, by extracting the 'First' and 'Rest' of each element in 'set...'.

ZIP `Algorithm:Zip(as bs)`

Produces a list of pairs, with respective elements from 'as' and 'bs'.

ZIP-WITH `Algorithm:Zip-With(func as bs)`

Applies a binary 'func' to elements pulled from 'as' and 'bs', collecting the results.

Complex

The `Complex` package provides a complex number type in source form, along with type-specific arithmetic operations and overloads for global arithmetic.

ABS `Complex:Abs(c)`

Computes the absolute value of complex 'c'.

ABS-SQUARE `Complex:Abs-Square(c)`

Computes the square of the absolute value of complex 'c'.

ADD `Complex:Add(a b)`

Adds two complex numbers.

CONJUGATE `Complex:Conjugate(c)`

Constructs a complex conjugate of 'c'.

CONS `Complex:Cons(real img)`

Constructs a Complex number from 'real' and 'img'inary parts.

CONS-MAYBE `Complex:Cons-Maybe(real img)`

Constructs a Complex number from 'real' and 'img', provided that they are real numbers.

DIV `Complex:Div(z1 z2)`

Divides complex 'z1' by complex 'z2'.

EQUAL `Complex:Equal(z1 z2)`

Compares the complex numbers 'z1' and 'z2' for equality.

IMG `Complex:Img(c)`

Retrieve Real and/or Imaginary part of a Complex number 'c'.

MAYBE `Complex:Maybe(real img)`

MUL `Complex:Mul(a b)`

Multiplies two complex numbers.

NEG `Complex:Neg(z)`

Negates a complex number 'z'.

POLAR `Complex:Polar(angle radius)`

Constructs a complex number from a polar representation: 'angle' in radians and 'radius'.

REAL `Complex:Real(c)`

Retrieve Real and/or Imaginary part of a Complex number 'c'.

SUB `Complex:Sub(a b)`

Substracts complex 'b' from 'a'.

UNITARY `Complex:Unitary(angle)`

Constructs a unitary complex number at 'angle' in radians.

Dictionary

The `Dictionary` package provides a key-value store mechanism that stores a single value per key, which can be retrieved or replaced. It is primarily intended for small stores as the implementation performs a linear search.

FIND `Dictionary:Find(dict key)`

Finds an entry in key-value list 'dict' whose key matches 'key'; or nil if nothing found.

INSERT `Dictionary:Insert(dict key value)`

Inserts a 'key'-'value' pair into 'dict'; if 'key' already exists in 'dict', the value is replaced. The modified collection is returned.

REMOVE `Dictionary:Remove(dict key)`

Removes an entry in key-value list 'dict' whose key matches 'key'; returns the modified collection.

Gen

The `Gen` package consists of building blocks for signal sources, such as oscillators, along with some predefined audio and control rate oscillators.

PHASOR `Gen:Phasor(clocking init inc)`

Create a periodic ramp within [0,1], increasing by 'inc' every time 'clock' ticks.

SIN `Gen:Sin(clocking freq)`

Sinusoid generator suitable for frequency modulation. Generates its own clock via the 'clocking' function.

WITH-UNIT-DELAY `Gen:With-Unit-Delay(func init)`

Route the output of 'func' back to its argument through a unit delay. Initialize the delay to 'init'.

DPW `Wave:DPW(freq)`

Implements a differentiated parabolic wave algorithm to provide a better quality sawtooth oscillator for audio rates. Updates at the audio rate, oscillating at 'freq' Hz.

SAW `Wave:Saw(freq)`

A Simplistic sawtooth generator without band limiting. Updates at the audio rate, oscillating at 'freq' Hz.

SIN `Wave:Sin(freq)`

Audio rate sinusoid generator suitable for frequency modultaion.

IO

The `IO` module provides routines for querying and defining inputs and clock rates that are external to the program, such as audio, MIDI or control inputs.

FREQUENCY-COEFFICIENT `Frequency-Coefficient(sig freq)`

Compute a frequency coefficient for the sample clock of 'sig'; the frequency is multiplied by the sampling interval and becomes a signal source with the chosen sample rate.

INTERVAL-OF `Interval-of(sig)`

Retrieve the sampling interval of 'sig' in seconds.

RATE-OF `Rate-of(sig)`

Retrieve the sampling rate of 'sig' in Herz.

IN `Audio:In()`

Represents all the audio inputs to the system as a tuple.

SIGNAL `Audio:Signal(sig)`

Treats 'sig' as an audio signal that updates at the audio rate.

PARAM `Control:Param(key init)`

Represents an external control parameter keyed as 'key', with the default value of 'init'.

SIGNAL `Control:Signal(sig)`

Treats 'sig' as a control signal that updates at 1/64 of the audio rate.

SIGNAL-COARSE `Control:Signal-Coarse(sig)`

Treats 'sig' as a control signal that updates at 1/512 of the audio rate.

SIGNAL-FINE `Control:Signal-Fine(sig)`

Treats 'sig' as a control signal that updates at 1/8 of the audio rate.

LinearAlgebra

The `LinearAlgebra` package provides elementary operations on matrices.

CONS `Matrix:Cons(rows)`

Constructs a matrix from a set of 'rows'.

ELEMENT `Matrix:Element(mtx col row)`

Retrieves an element of matrix 'mtx' at 'row' and column 'col', where 'row' and 'col' are zero-based invariant constants.

HADAMARD-PRODUCT `Matrix:Hadamard-Product(a b)`

Computes the Hadamard product of matrices 'a' and 'b'.

MAP `Matrix:Map(func mtx)`

Applies function 'func' to all elements of matrix 'mtx', returning the resulting matrix.

MUL `Matrix:Mul(a b)`

Multiplies matrices 'a' and 'b'.

TRANSPOSE `Matrix:Transpose(matrix)`

Transposes the 'matrix'.

Math

The `Math` package defines mathematical functions, such as the typical transcendental functions used in many programs.

COS `Math:Cos(a)`

Take cosine of an angle in radians.

COSH `Math:Cosh(x)`

Computes the hyperbolic cosine of 'x'.

COTH `Math:Coth(x)`

Computes the hyperbolic cotangent of 'x'.

CSCH `Math:Csch(x)`

Computes the hyperbolic cosecant of 'x'.

EXP `Math:Exp(a)`

Compute exponential function of 'a'.

HORNER-SCHEME `Math:Horner-Scheme(x coefficients)`

Evaluates a polynomial described by the set of 'coefficients' that correspond to powers of 'x' in ascending order. The evaluation is carried out according to the Horner scheme.

LOG `Math:Log(a)`

Compute the natural logarithm of 'a'.

LOG10 `Math:Log10(a)`

Compute the base 10 logarithm of 'a'.

POW `Math:Pow(a b)`

Compute the 'b':th power of 'a', where 'a' and 'b' are real numbers.

SECH `Math:Sech(x)`

Computes the hyperbolic secant of 'x'.

SIN `Math:Sin(a)`

Take sine of an angle in radians.

SINH `Math:Sinh(x)`

Computes the hyperbolic sine of 'x'.

SQRT `Math:Sqrt(a)`

Takes the square root of a floating point number.

TANH

```
Math:Tanh(x)
```

Computes the hyperbolic tangent of 'x'.

B TUTORIAL

This appendix is intended to explain the Kronos language from user perspective to a person with some programming background. The tutorial consists of a set of thoroughly explained program examples, which are intended to demonstrate the unique features of the system, as well as a section on the usage of the command line drivers for the Kronos compiler. This tutorial doesn't cover the syntax: Please refer to Section A.1 for details. The compiler driver shown in all the examples is `krepl`; please see Section B.3 for a brief user guide.

B.1 INTRODUCTION

Programming in Kronos is about functions. Some fundamental functions are provided as a baseline: elementary math, structuring data and function application are built into the system. Everything else builds on these; the provided runtime library composes the fundamental functions into tasks that are more complex, such as evaluating a polynomial, handling richer data structures or producing an oscillating waveform.

`Math:Cos` is an example of a fundamental function that computes the cosine of an angle given in radians. We can call the function from the REPL to see the return value.

```
> Math:Cos(0)
1
> Math:Cos(3.141592)
-1
```

In this example, `Math:` specifies that the function we are looking for is in the Math package. `Cos` is the name of the function, followed by arguments immediately following the name, grouped in parentheses. We can use `Math:Cos` as a building block in a slightly more complicated function that computes the catheti of a right-angled triangle, given the hypotenuse and an angle in radians:

```
> Catheti(w r) {
>  Catheti = (r * Math:Cos(w) r * Math:Sin(angle))
> }
Function
> Catheti(0.6435 5)
4 3
```

Here we specified a function that uses cosine and sine, constructing a *tuple* of two numbers out of them.

In addition to typical function calls that follow the form *package–symbol–arguments*, Kronos supports infix functions. These are the familiar mathematical operators, which are a convenient alternative to common function notation:

```
> 2 + 4 - 5 * 6
-24
> Sub(Add(2 4) Mul(5 6))
-24
```

To build a function that generates a sound from a non-audio source, such as an oscillator that produces a waveform from a frequency parameter, Kronos utilizes *signal memory* or delay. Built-in delay operators allow functions to compute on values of other symbols at some previous moment in time. This can be used to build a phase integrator:

```
> Ig(rate) {
> st = z-1(0 st + Audio:Signal(rate))
> Ig = st
> }
```

The function `Ig` defines a local symbol, `st`. The definition looks odd: `z-1` looks like a function call, but there is a circularity: the value of `st` is used to compute its own definition!

This can work only because `z-1` is a special operator rather than a real function. It takes two parameters, an *initializer* and a *signal*. It returns a delayed version of *signal*, or if there is no past signal yet, the *initializer*. Because of this, the value of `st` in terms of itself can be computed incrementally, sample by sample, the way we typically process audio. The length of the delay is determined by the incoming *signal*. In this case, we explicitly create an `Audio:Signal`, upsampling the `rate` parameter.

A simple way to obtain an oscillation is to put the integrator through a periodic function such as `Math:Sin`. This is a rather bad oscillator, because the ever-increasing phase accumulator interacts badly with floating point arithmetic. As a result, tuning errors can be observed as the oscillator keeps playing.

```
> snd = Math:Sin(Ig(0.02))
0
```

The basic concepts in Kronos include functions, signal memory and circular definitions. Signal processors are composites of these building blocks. In Section B.2, more advanced compositions of functions, signals, types and reactivity are explored.

B.2 EXAMPLES

B.2.1 Higher Order Functions

You can define functions using the lambda arrow syntax: `arg => expr`. The result of the arrow operator is an anonymous function.

```
> MyFunction = x => x + 10
Function
> MyFunction(1)
11
> Algorithm:Map(MyFunction 1 2 3 4 5)
11 12 13 14 15
```

In this example, a simple function that adds 10 to its argument is constructed and passed as a parameter to `Algorithm:Map`. Map is a *higher order function*; it makes use of other functions to perform a type of task. In this instance, the other function is `MyFunction`, and the task is to apply it to all elements in a set. The `Algorithm` package contains many higher order functions that correspond to patterns commonly achieved with loops in imperative languages.

Let us define a higher order function of our own:

```
> MyFold(f tuple) {
>  MyFold = tuple
>  (x xs) = tuple
>  MyFold = f(x MyFold(f xs))
> }
Function Function
> MyFold(Add 1 2 3 4 5)
15
> MyFold(String:Append "Hello" " " "World")
Hello World
```

There are a number of notable points in this small example. The function we have implemented is Fold, a functional programming staple. Fold combines the elements in a set with a binary function: we apply it in the example to do the sum $1 + 2 + 3 + 4 + 5$ as well as concatenate strings.

The way Fold works is *polymorphic*. The return value is defined as either `tuple` or `f(x MyFold(f xs))`. Kronos will use the latter definition whenever possible, as it is defined later in the source file. Let's examine this form in detail.

The expression consist of two function calls, to `f` and `MyFold`. The former is a function parameter, while the latter is the function itself – a recursive call. The use of a function parameter that is itself a function is what makes `MyFold` a *higher order function*.

The remaining symbols in the expression are `x` and `xs`, defined via a *destructuring bind* `(x xs) = tuple`. `x` is bound to the head of the tuple, while `xs` is bound to everything else in

the tuple – the tail. The operating principle of our function is then to call itself recursively, with an identical function parameter (`f`) and the tail of the `tuple` .

Previously, we glossed over the *polymorphic form selection*, stating just that Kronos would use this form of `MyFold` whenever possible. The destructuring bind is key to what is "possible": when the tuple can not be destructured – has no head and tail – this form of `MyFold` fails to satisfy the *type constraints*. Kronos will then fall back to the next available form, in this case `MyFold = tuple` , returning the tuple as is.

The following illustrates the recursive call and the return value of each `MyFold` call.

```
MyFold(Add 1 2 3 4 5)
  MyFold(Add 2 3 4 5)
    MyFold(Add 3 4 5)
      MyFold(Add 4 5)
        MyFold(Add 5) ; can't destructure!
        5
      Add(4 5)
    Add(3 9)
  Add(2 12)
Add(1 14)
15
```

Higher Order Functions and Signal Routing

In the context of signal processing, higher order functions can be considered to construct common signal path topologies. `Algorithm:Map` , the first higher order function demonstrated in this tutorial, corresponds to a parallel topology, such as a parallel filter bank.

`MyFold` is different from `Algorithm:Map` in one essential way. While the parameter functions passed to `Map` all operate on elements of the input set, `Fold` threads the signal path through the parameter function so that the output of each function call is fed to the input of the next one. This can be used to perform serial routing.

Let's use `MyFold` together with an oscillator to create a summing network:

```
> Import Gen
Ok
> add-sin = (freq sig) => sig + Wave:Sin(freq) * 0.1
Function
> snd = MyFold(add-sin 440 550 660 0)
0.0207643
> snd = ()
nil
```

This example should play a major triad with sinusoids. This time, the folding function we provide is more complicated than in the trivial examples above: we expect a frequency as the first argument and a signal bus as the second argument. The function adds a `Wave:Sin` oscillator to the signal bus, scaled by a gain coefficient, and returns the bus. Because of the way `MyFold` was designed, we can pass a list of frequencies followed by an initial signal bus to create an oscillator bank.

Often, it can be easier to split such tasks between several higher order functions. In the following example, we create a parallel oscillator bank usign `Map` and a summing network with `MyFold`. In this configuration, we can pass in library functions directly, and do not need to specify a complicated anonymous function as in the example above.

```
> oscs = Algorithm:Map(Wave:Sin 440 550 660)
0.0540629 0.0692283 0.0843518
> snd = MyFold(Add oscs) * 0.1
0.207643
> snd = ()
```

Folds can be used for a variety of serial routings. In the final example, we are going to use `Algorithm:Expand` and `Algorihm:Reduce` from the runtime library. `Reduce` greatly resembles our `MyFold` – it is just designed to thread the signal from left to right, which can often be more computationally efficient, and easier to read. `Expand` generates a list of elements from a seed element and an iterator function applied a number of times. As a simple example, we generate a list of 10 elements, starting from seed `1`, and iterating with the function `(* 2)` to generate a series of powers of two.

```
> Algorithm:Expand(#10 (* 2) 1)
1 2 4 8 16 32 64 128 256 512 nil
```

```
> f0 = 220
220
> ops = #3
#3
> freqs = Algorithm:Expand(ops (+ 110) f0)
220 330 440 nil
> rfreqs = Algorithm:Reduce((a b) => (b a) freqs)
440 330 220
> fm-op = (sig freq) => Wave:Sin(freq + sig * freq)
Function
> snd = Algorithm:Reduce(fm-op 0 rfreqs)
0.024885
```

This creates a simple FM-synthesizer with three cascaded operators. We generate a series of harmonic frequencies with `Expand`, reverse their order by using `Reduce`, and use another `Reduce` to construct modulator-carrier pairs.

The REPL tracks symbol dependencies. This means that if you make changes that reflect on the definition of `snd`, a new signal processor is constructed for you:

```
> f0 = 220 + Wave:Sin(5.5) * 10
219.938
> ops = #4
#4
> rfreqs = (0.1 330 440 550 165)
0.1 330 440 550 165
```

For additional examples, please see the literature [4] for a set of synthetic reverberators utilizing higher order functions.

B.2.2 Signals and Reactivity

The Kronos signal model is based on reactivity. Each Kronos function is a *pure function of current and past inputs*. Past input is accessed via *delay operators* such as `z-1` (unit delay) and `rbuf` (ring buffer).

The *timing* of input versus output depends on external stimuli, or inputs to the signal processor. These include input streams like audio, possibly at many different update rates at once, and event-based signals like MIDI, OSC or user interface widgets. The Kronos Core is largely oblivious of the details of these input schemes, only providing facilities to declare typed inputs and associated drivers that can act as clock sources for one or more inputs. This scheme enables asynchronous and synchronous *push semantics*.

The reactive propagation described in [3] makes building of input–output signal processors very easy. Such processors, for example digital filters, can be driven by the input signal, which provides the clock for the processor. Most filter implementations can ignore signal clock completely, although some may require the sample rate to compute coefficients. Shown here is a resonator that is inherently capable of adapting to any sample rate:

```
Resonator(sig amp freq bw) {
    sample-rate = Reactive:Rate(sig)

    w = #2 * Math:Pi * freq / sample-rate
    r = Math:Exp(Neg(Math:Pi * bw / sample-rate))
    norm = (#1 - r * r) * amp

    a1 = #-2 * r * Math:Cos(w)
    a2 = r * r

    zero = x0 - x0
```

```
    y1 = z-1(zero y0)
    y2 = z-1(zero y1)
    y0 = sig - y1 * a1 - y2 * a2

    Resonator = norm * (y0 - y2)
}
```

This filter implementation queries the `Reactive:Rate` of the input signal `sig`, which provides the sample rate for whichever clock is driving `sig`. Computations that depend on `sample-rate` will be performed only when the sample rate changes, rather than at the audio rate.

Reactivity is a little more complicated when we want to design signal processors whose output clock is not derived directly from the input. A typical case would be an oscillator: audio oscillators should update at the audio rate, and low frequency oscillators may update at some lower rate. Most of the time, the inputs to these oscillators, such as frequency, waveform or amplitude controls, do not update at the desired output rate.

The Kronos runtime library defines functions that *resample* signals to commonly desired clocks. One such function is `Audio:Signal`, which resamples any signal at the base sample rate for audio.

Many oscillators are defined in terms of unit-delay feedback, which is the way you can add *state* to Kronos signal processors. As the length of the unit delay is determined based on incoming signal clock, we must be careful to send signal to the unit-delay feedback loop at the correct update rate. Below is a simplistic example of an oscillator:

```
Phasor(freq) {
    driver = Audio:Signal(0) ; dummy driver
    sample-rate = Reactive:Rate(driver)
    state = z-1(freq - freq Audio:Signal(wrap))
    next  = state + freq * (#1 / sample-rate)
    wrap  = next - Floor(next)
    Phasor = state
}
```

This oscillator makes use of a faux audio input by constructing one via `Audio:Signal(0)`. It is used to obtain the sample rate of the audio clock. Each sample in the output stream is computed by adding the frequency, converted to *cycles per sample*, to the previous sample and substracting the integral part via `Floor`. This results in a periodic ramp that increases linearly from 0 to 1 once per cycle.

We can listen to the phasor and even make it modulate itself or use it with a higher order function. A word of warning: these sounds are a little harsh.

```
> snd = Phasor(440)
0
```

```
> snd = Phasor(440 + 110 * Phasor(1))
0
> snd = Algorithm:Reduce(
            (sig f) => Phasor(f + sig * f)
            0 1 110 440)
0
> snd = nil
nil
```

The audio phasor function serves as a basis for a number of oscillators, as it generates a non-bandlimited ramp signal that cycles between 0 and 1 at the frequency specified. A set of naive geometric waveforms are easy to come by:

```
> Saw = freq => #2 * Phasor(freq) - #1
Function
> Tri = freq => #2 * Abs(Saw(freq)) - #1
Function
> Pulse = (freq width) => Ternary-Select(
                            Phasor(freq) < width 1 -1)
Function
> snd = Pulse(55 0.5 + 0.5 * Tri(1))
1
```

Many signal processing algorithms can be optimized by lowering the update rate of certain sections. In the literature, this is known as *control rate* processing. It is applied to signals whose bandwidth much lower than the audio band.

We can reformulate our oscillators in terms of a desired update rate and waveshaping function:

```
Phasor(clocking freq) {
    driver = clocking(0) ; dummy driver
    sample-rate = Reactive:Rate(driver)
    state = z-1(freq - freq clocking(wrap))
    next  = state + freq * (#1 / sample-rate)
    wrap  = next - Floor(next)
    Phasor = state
}
> Saw = x => #2 * x - #1
Function
> Tri = x => #2 * Abs(Saw(x)) - #1
Function
> Pulse = width => (x => Ternary-Select(x < width 1 -1))
Function
> Osc = (shape freq) => shape(Phasor(Audio:Signal freq))
```

```
Function
> LFO = (shape freq) => #0.5 + #0.5 *
    shape(Phasor(Control:Signal-Coarse freq))
Function
> snd = Osc(Saw 440)
Function
> snd = Osc(Saw 440 + 40 * LFO(Tri 5))
Function
> snd = Osc(Pulse(LFO(Saw 1)) 110)
Function
```

We use two clocking functions, `Audio:Signal` and `Control:Coarse` to differentiate the update reates of the phasors. Coarse control rate is 512 times slower than audio rate – such an extremely low control rate is used here to better illustrate multirate processing.

Sometimes, sonic artifacts related to multirate processing can arise. A common case is zipper noise, which is easily heard in amplitude modulation:

```
> snd = Osc(Tri 440) * LFO(Tri 1)
Function
```

This noise is caused by sudden changes in the amplitude, because the control signal looks like a staircase waveform from the audio point of view. A form of interpolation is required to smooth over the edges. An example of a linear interpolator is given below:

```
Upsample-Linear(sig to-clock) {
    x0 = sig
    x1 = z-1(sig - sig x0)

    slow-rate = Reactive:Rate(sig)
    fast-rate = Reactive:Rate(to-clock(0))

    inc  = slow-rate / fast-rate
    wrap = x => x - Floor(x)

    state = z-1(0 to-clock(wrap(state + inc)))

    Upsample-Linear = x0 + state * (x1 - x0)
}

> snd = Osc(Tri 440) * Upsample-Linear(LFO(Tri 1) Audio:Signal)
Function
```

Finally, let's examine an OSC event stream. The correct way to use such a signal depends on the application: if instantaneous transitions are desired, the stream can usually be sent directly to the synthesis function. Alternatively, the signal could be resampled with a steady clock, such as the audio or control signal rates, and filtered or smoothed. Please note that smoothing an event stream without first injecting a steady update rate is not advised.

The following is a small example of direct usage, as well as a smoothing filter applied to the control parameter after it being upsampled to control rate `Fine`, which runs at 1/8th of the audio signal rate.

```
> freq = IO:Param("/freq" 440)
440
> snd = Osc(Saw freq)
Function
> lag(sig speed) {
    st = z-1(sig st + speed * (sig - st))
    lag = st
}
Function
> freq = lag(Control:Fine(IO:Param("/freq" 440)) 0.1)
440
```

B.2.3 Type-driven Metaprogramming

The previous tutorial on *higher order functions* demonstrated a *polymorphic* function – a function whose behavior depends on the type of the parameters passed to it. Various forms exist for such functions, and an appropriate one is picked according to the *type constraints* imposed by each form.

`MyFold` used destructuring as a type constraint to govern form selection. The specific behavior of the function depends on whether the data parameter contains a *tuple*, that is, a number of elements. For just one element, the recursion terminates, while several elements cause continued recursion.

In addition to *structural* type features, such as the number of elements bound to a symbol, Kronos also supports *nominal* types. These are semantic metadata – names – that are attached to data. Consider two structurally identical but semantically different datums: a complex number and a stereophonic sample. Both can consist of two floating point numbers. However, a programmer would expect a multiplication of complex numbers to adhere to the mathematical definition, while a product of stereophonic samples would more logically be performed per channel.

Further, if a set of types adhere to common behavior, such as basic arithmetic, we can design functions against that behavior, ignoring the implementation details of the types in question. This can be used in signal processing to write processor templates that can handle various I/O configurations, such as different channel counts and sample resolutions.

Generic Filtering

As an exercise, let us define a simple low pass filter that adapts dynamically to the I/O configuration. The basic implementation of the filter is shown below:

```
MyFilter(sig tone) {
    zero = sig - sig
    y1 = z-1(zero y0)
    y0 = y1 + tone * (sig - y1)
    MyFilter = y0
}
```

This filter requires that the types of `sig` and `tone` have well-defined operators for `Add`, `Sub` and `Mul`. As such, we can use it for single- or double precision numbers.

```
> Import Gen
Ok
> noise = Noise:Pseudo-White(0.499d)
0
> mod = 0.5 + 0.5 * Wave:Sin(1)
0.496607
> snd = MyFilter(noise mod)
0
```

Now, let's make it work for an arbitrary number of channels as well. A straightforward solution would be to define the filtering in terms of `Algorithm:Map`, but with a type-based solution, we can enable multichannel processing for *all the generic filters* in the Kronos system in one fell swoop.

```
Type Multichannel
Package Multichannel {
    Cons(channels) {
        Cons = Make(:Multichannel channels)
    }

    As-Tuple(mc) {
        As-Tuple = Break(:Multichannel mc)
    }

    Binary-Op(op a b) {
        Binary-Op = Multichannel:Cons(
                Algorithm:Zip-With(op
                    Multichannel:As-Tuple(a)
                    Multichannel:As-Tuple(b)))
    }
}
```

```
Add(a b) {
    Add = Multichannel:Binary-Op(Add a b)
}

Sub(a b) {
    Sub = Multichannel:Binary-Op(Sub a b)
}

Mul(a b) {
    Mul = Multichannel:Binary-Op(Mul a b)
}
```

This defines just enough of the `Multichannel` type for it to work in our filter. We can construct multichannel samples out of tuples of real numbers with `Multichannel:Cons`. `As-Tuple` retrieves the original tuple. These functions have very simple definitions: `Cons` attaches a semantic tag, defined by `Type Multichannel` to the data passed to it. `As-Tuple` removes the tag, but has a type constraint: it is a valid function call if and only if the type tag of the parameter was actually `Multichannel`.

We define a helper function, `Binary-Op` that acts like a functional *zip* on two multichannel signals: it applies a binary function pairwise to each element in the multichannel samples. Because `Binary-Op` uses `As-Tuple` on its arguments, it inherits the type constraints: the function is not valid if either `a` or `b` is not a multichannel sample.

This is makes polymorphic extensions to `Add`, `Sub` and `Mul` simple. The compiler is able to perform pattern matching on the arguments based on the fact that these functions call `Binary-Op`, which requires `Multichannel` types.

This is enough to make arithmetic work and also to stop someone from obliviously destructuring a multichannel sample:

```
> Multichannel:Cons(1 2 3) + Multichannel:Cons(20 30 40)
:Multichannel{21 32 43}
> Rest(Multichannel:Cons(1 2 3))
* Program Error E-9995: immediate(0;0); Specialization failed *
| Rest of non-pair
|  no form for Rest :Multichannel{Floatx3}
```

`MyFilter` can also automatically take advantage of `Multichannel`:

```
> stereo-noise = Multichannel:Cons(noise noise)
:Multichannel{0 0}
```

```
> lfo = freq => 0.5 + 0.5 * Wave:Sin(freq)
Function
> mod = Multichannel:Cons(lfo(1) lfo(1.1))
:Multichannel{-0.00678665 -0.00677262}
> snd = MyFilter(stereo-noise mod)
:Multichannel{-0.00678665 -0.00677262}
```

With our current code, `Multichannel` cannot interact with monophonic signals. `MyFilter` only works if both `sig` and `tone` are `Multichannel` signals. It would be useful if we could scale or translate a multichannel signal with a scalar. While we could define additional forms for `Add`, `Mul` and `Div`, the runtime library contains type conversion infrastructure we can make use of. First, let's provide an explicit type conversion from a scalar to `Multichannel`:

```
Package Type-Conversion {
    Explicit(type data) {
        channels = Multichannel:As-Tuple(type)
        Explicit = When(Real?(data)
            Multichannel:Cons(
                Algorithm:Map('data channels)))
    }
}
```

`As-Tuple` serves a dual purpose here: firstly, we obtain the number of channels in the conversion target format. Secondly, we add a type constraint – this explicit conversion applies only when the target `type` can accomodate `Multichannel:As-Tuple`.

Rudimentary type checking is added via a `When` clause, which requires `data` to be a (scalar) real number. This prevents a number of strange things like strings, dictionaries and nested multichannel structures ending up inside this `Multichannel` sample. The final sample is produced by mapping the channels of the target type with a simple funtion returning the source datum.

This allows coercion of real numbers to a corresponding multichannel format; however, arithmetic still fails:

```
> Coerce(Multichannel:Cons(0 0) 1)
:Multichannel{1 1}
> Coerce(Multichannel:Cons(0 0 0 0) 5i)
:Multichannel{5 5 5 5}
> 1 + Multichannel:Cons(1 1)
* Program Error E-9995: immediate(0;2); Specialization failed *
|  no form for Add (Float :Multichannel{Float Float})
```

This is because Kronos will not do type conversions implicitly unless instructed. More specifically, the runtime library is programmed to fall back on a specific mechanism for implicit coercion when binary operators are called with mismatched operands. To make the implicit case work, we need to provide the following function:

```
Multichannel?(mc) {
    Multichannel? = nil
    Multichannel? = Require(Multichannel:As-Tuple(mc) True)
}

Package Type-Conversion {
    Implicit?(from to) {
        Implicit? = When(Real?(from) & Multichannel?(to) True)
    }
}
```

We implement a small reflection function, `Multichannel?` which returns a truth value based on whether the argument is a multichannel sample. `Type-Conversion:Implicit?` is queried by the Kronos runtime library to check if a particular conversion should be done implicitly. In this case, we return `True` when real numbers are to be converted to `Multichannel` samples. Now, we can mix and match reals and multichannel samples.

```
> Multichannel:Cons(1 2) * 3
:Multichannel{3 6}
> 5 + 7 * Multichannel:Cons(1 10)
:Multichannel{12 75}
```

By now, our filter and sample type are pretty flexible:

```
> pan = (sig p) => Multichannel:Cons(sig * (1 - p) sig * p)
Function
> snd = pan(noise lfo(1))
:Multichannel{-0.00200005 4.52995e-08}
> snd = MyFilter(pan(noise lfo(1)) lfo(0.5))
:Multichannel{-0.000794533 1.79956e-08}
```

We specified a small ad hoc function, `pan`, to generate a stereo signal from a monophonic signal. We then pass that to `MyFilter` along with a scalar coefficient. The implicit type conversions result in the filter automatically becoming a 2-channel processor with a single coefficient controlling all the channels. Alternatively, we could pass a multichannel coefficient to control each channel separately.

Notably, we injected all this functionality into `MyFilter` without altering any of its code after the initial, simple incarnation. This is the power of type-based metaprogramming. Further extensions to `Multichannel` type could include format descriptions, such as LCR or 5.1, and automatic, semantically correct signal conversions between such formats.

B.2.4 Domain Specific Language for Block Composition

In this section we develop a small DSL within Kronos, inspired by the Faust [23] block-diagram algebra. First, let's define the elementary composition functions. *Parallel* composition means evaluating two functions side by side, splitting the argument between them.

```
> Parallel = (a b) => ( (c d) => (a(c) b(d)) )
Function
> Eval(Parallel(Add Sub) (10 1) (2 20))
11 -18
```

Next, *Serial* composition in the same vein:

```
> Serial = (a b) => ( x => b(a(x)) )
Function
> Eval(Serial((+ 10) (* 5)) 1)
55
```

Recursive composition completes our DSL. This composition routes its output back to its input via the right hand side function.

```
Recursive(a b) {
    recursively = {
        (sig fn) = arg
        st = z-1(sig upd)
        upd = fn(st)
        upd
    }
    Recursive = in => recursively(in fb => a(b(fb) in))
}
```

This composition allows us to define an oscillator:

```
> Phasor = Serial((/ Audio:Rate())
```

```
            Serial(Audio:Signal
                Recursive(Add Serial(
                    Serial('(_ _)
                        Parallel('_ Floor)) Sub))))
:Closure{Function Function :Closure{Function ...
> snd = Phasor(220)
0
```

At this point it is very unclear why someone would like to compose signal processors in this way, as it seems very cumbersome. The essence of this style is that it is *point-free* – there is never a need to specify a variable. To truly take advantage of this style – as Faust [23] does – the syntax must become less cumbersome. One way to enhance the programming experience is to define *custom infix functions*.

Kronos treats all symbols that begin with a punctuation mark as infix symbols. If the infix isn't one of the recognized ones, the parser will substitute a call to the function `Infix<symbol>`. These custom infices have operator precedence that is lower than the standard infices – see Table 6 for details.

```
> Dup = '(_ _)
Function
> Infix-> = Serial
Function
> Infix~ = Recursive
Function
> Infix-< = (a b) => (a -> Dup -> b)
Function
> Infix|| = Parallel
Function
> Phasor = (/ Audio:Rate()) -> Audio:Signal
         -> (Add ~ ('_ -< ('_ || Floor) -> Sub))
:Closure{Function :Closure{Function :Closure{Function ...
> snd = Phasor(220)
0
> Tri = Phasor -> (- 0.5) -> Abs -> (* 2) -> (- 0.5)
:Closure{Function :Closure{Function :Closure{Function  ...
> snd = Tri(220)
-0.480045
```

However, the point-free style is at its best when combined with a suitable set of general purpose functions written in the regular style;

```
> Fraction = x => x - Floor(x)
Function
```

```
> Phasor = (/ Audio:Rate()) -> Audio:Signal -> (Add ~ Fraction)
:Closure{Function :Closure{Function :Closure{Function  ...
```

If we formulate `Fraction` as a separate function in the normal style, the formulation of the phasor is arguably more elegant than what could have been achieved in either style alone. The pipeline starts by normalizing the frequency in Hertz to cycles per sample: the signal is then resampled to audio rate by `Audio:Signal`, and the actual oscillation is provided by a recursive composition of `Add` and `Fraction`.

A similar composition can also generate a recursive sinusoid oscillator:

```
> SinOsc = (* (Math:Pi / Audio:Rate())) -> Complex:Unitary
         -> Audio:Signal -> (Mul ~ '_) -> Complex:Real
:Closure{Function :Closure{Function :Closure{Function  ...
> snd = SinOsc(440)
0.998036
```

The DSL can accommodate many types of digital filters, for which we define some helper functions:

```
> Convolution(sig coefs) {
    (c cs) = coefs
    z = sig - sig
    Convolution = When(Atom?(coefs) sig * coefs)
    Convolution = sig * c + Convolution(z-1(z sig) cs)
    Convolution = When(Nil?(coefs) 0)
}
Function Function
> Conv = coefs => ('Convolution(_ coefs))
Function
```

`Convolution` is a FIR filter that can accommodate any order, constructed with functional recursion. `Conv` is a wrapper for the point-free style: it can be used to create a convolution stage in our DSL. Note that the wrapper is essentially a partial application: the anonymous function just leaves a "blank" for the pipeline to connect to. Now we can define a biquad filter in the Faust fashion:

```
> Biquad = (a1 a2 b0 b1 b2) => ( ( Add ~ Conv(a1 a2) )
         -> Conv(b0 b1 b2) )
Function
> Resonator(freq bw) {
```

```
    w = #2 * Math:Pi * freq / Audio:Rate()
    r = Math:Exp(Neg(Math:Pi * bw / Audio:Rate()))
    norm = (#1 - r * r)
    Resonator = Biquad(#2 * r * Math:Cos(w)
                       Neg(r * r)
                       norm #0 Neg(norm))
}
Function
> Import Gen
Ok
> noise = 'Noise:Pseudo-White(0.499d)
Function
> snd = Eval( noise -> Resonator(440 10) nil )
3.4538e-6
```

As has been proven by the Faust [23] project, block-diagram algebra can enable very compact, elegant formulations of signal processors. This tutorial has demonstrated the implementation of domain specific language for block-diagram algebra in the Kronos language.

For a proper library structure, the custom infices could be placed in a `Package`, such as `Block-Diagram`. That would cause the infices to be confined to the namespace, being enabled per-scope by the directive `Use Block-Diagram`.

B.3 USING THE COMPILER SUITE

B.3.1 kc: The Static Compiler

`kc` is the static compiler of the compiler suite. It is intended to produce statically compiled versions of Kronos programs, as either LLVM [27] assembly code, machine-dependent assembly code, or binary object code that can be integrated in a C-language compatible project. The compiler can optionally produce a C/C++ header files to facilitate the use of the generated signal processor.

`kc` expects command line arguments that are either Kronos source code files (.k) to be loaded, and named parameters that are conveyed with command line switches. Each switch has short and long forms. The switches are displayed in Table 10.

By default, `kc` produces a binary object in the native format of the host platform. This object provides entry points with C-linkage that correspond to the active external inputs to the signal processor. Alternatively, the `-S` switch generates machine dependant assembly code instead. Using `-S` and `-ll` together produces assembly in the platform-independent LLVM intermediate representation. Please note that this code is not necessarily portable, as the compiler optimizes according to the target architecture settings. Disabling optimization with the `-O0` switch can produce platform-independent LLVM IR.

The target machine can be specified with the `-mcpu` and `-mtriple` switches. The latter is a Linux-style architecture triple, while the former can specify a target CPU that is more specific than that of given by the triple. The available CPU targets can be listed by using `-mcpu help`.

Table 10: Command line parameters for `kc`

Long	Short	Param	Description
–input	-i	<path>	input source file name; '-' for stdin
–output	-o	<path>	output file name, '-' for stdout
–header	-H	<path>	write a C/C++ header for the object to <path>, '-' for stdout
–output-module	-om	<module>	sets -o <module>.obj, -P <module> and -H <module>.h
–main	-m	<expr>	main; expression to compile
–arg	-a	<expr>	Kronos expression that determines the type of the external argument to main.
–assembly	-S		emit symbolic assembly
–prefix	-P	<sym>	prefix; prepend exported code symbols with 'sym'
–emit-llvm	-ll		export symbolic assembly in LLVM IR format
–emit-wavecore	-WC		export symbolic assembly in WaveCore format
–mcpu	-C	<cpu>	engine-specific string describing the target cpu
–mtriple	-T	<triple>	target triple to compile for
–disable-opt	-Oo		disable LLVM code optimization
–quiet	-q		quiet mode; suppress logging to stdout
–diagnostic	-D		dump specialization diagnostic trace as XML
–help	-h		display this user guide

B.3.2 kpipe: The Soundfile Processor

`kpipe` is a soundfile processor that can feed an audio file with an arbitrary number of channels through a Kronos signal processor. The supported formats depend on the host platform: on Windows, `kpipe` supports the formats and containers provided by the Microsoft Media Foundation. On Mac OS X, the format support depends on Core Audio. On Linux, `kpipe` depends on the libsndfile component. A list of the available command line parameters is given in Table 11. The unnamed parameters are interpreted as paths to Kronos source files (.k) to be loaded prior to compilation.

Table 11: Command line parameters for `kpipe`

Long	Short	Param	Description
–input	-i	<path>	input soundfile
–output	-o	<path>	output soundfile
–tail	-t	<samples>	set output file length padding relative to input
–bitdepth	-b		override bit depth for output file
–bitrate	-br		override bitrate for output file
–expr	-e	<expr>	function to apply to the soundfile
–quiet	-q		quiet mode; suppress logging
–help	-h		display this user guide

Table 12: Command line parameters for `kseq`

Long	Short	Param	Description
–input	-i	<path>	input source file name; '-' for stdin
–main	-m	<expr>	main; expression to connect to audio output
–audio-device	-ad	<regex>	audio device; 'default' or a regular expression pattern
–audio-file	-af	<path>	audio file; patch an audio file to audio input
–dump-hex	-DH		write audio output to stdout as a stream of interleaved 16-bit hexadecimals
–quiet	-q		quiet mode; suppress printing to stdout
–length	-len	<samples>	compute audio output for the first <samples> frames
–profile	-P		measure CPU utilization for each signal clock
–log-sequencer	-ls		log processor output for each sequencer input
–help	-h		help; display this user guide

B.3.3 kseq: The JIT Sequencer

`kseq` is a command line sequencer that can compile Kronos programs in real time and apply sequences of control data in the EDN format either in real time or from disk. In addition, the drive can profile the computational performance of the signal processor, factored for audio and control clocks. The unnamed parameters are interpreted as paths to Kronos source files (.k) to be loaded prior to compilation. A control data sequence can be supplied by the `-i` switch, or provided via the standard input.

A list of the available command line switches to `kseq` is shown in Table 12.

B.3.4 krepl: Interactive Command Line

`krepl` is an interactive read-eval-print-loop application that drives the Kronos compiler. It provides a command prompt for expressions, the results of which are immediately displayed. While symbol redefinition is disallowed by the core language, it is specifically allowed in the context of the REPL to facilitate iterative programming. The REPL engine detects modifications to a global-level symbol `snd` and all the symbols it depends on. When such a modification is detected, the REPL patches `snd` to the audio device.

The REPL generates some Kronos symbols to facilitate audio configuration in the `Configuration` package. `Available-Audio-Devices` is a list of audio devices detected by `krepl`. `Audio-Device` is a function that returns the name of the device currently used. It defaults to the first item in `Available-Audio-Devices` and can be modified in the REPL. `Sample-Rate` evaluates to the desired sample rate. When `krepl` interacts with the audio hardware, it uses these symbols to configure it.

Unlike the other drivers, `krepl` interprets unnamed parameters as expressions to be fed to the REPL. Source files can be imported by the `-i` switch or by using the `Import` statement within the REPL.

Table 13: Command line parameters for `krepl`

Long	Short	Param	Description
–audio-device	-ad	<regex>	audio device; 'default' or a regular expression pattern
–osc-udp	-ou	<integer>	UDP port number to listen to for OSC messages
–interactive	-I		Prompt the user for additional expressions to evaluate
–format	-f	<text | edn | xml>	Format REPL responses as...
–import	-i	<module>	Import source file <module>
–help	-h		help; display this user guide

C LIFE CYCLE OF A KRONOS PROGRAM

This appendix demonstrates the implementation of the Kronos compiler pipeline by exhibiting and explaining the internal representation of a user program at various intermediate stages. The program in question is the supplementary example from the introductory essay of this report: first seen in Section 3.1.1. Node graphs of the more interesting aspects of the program, namely `Map`, `Reduce` and `Sinusoid-Oscillator`, are shown in visual form.

C.1 SOURCE CODE

Listing 13: The Supplementary Example

```
; the Algorithm library contains higher order
; functions like Expand, Map and Reduce

; the Complex library provides complex algebra

; the IO library provides parameter inputs

; the Closure library provides captures for
; lambdas

; the Math library provides the Pi constant

Import Algorithm
Import Complex
Import IO
Import Closure
Import Math
Import Implicit-Coerce

Generic-Oscillator(seed iterator-func) {
  ; oscillator output is initially 'seed',
  ; otherwise the output is computed by applying
  ; the iterator function to the previous output

  ; the audio clock rate is injected into the loop
  ; with 'Audio:Signal'

  out = z-1(seed iterator-func(Audio:Signal(out)))

  ; z-1 produces an unit delay on its right hand side
  ; argument: the left hand side is used for
  ; initialization

  ; Audio:Signal(sig) resamples 'sig' to audio rate

  Generic-Oscillator = out
}
```

```
Sinusoid-Oscillator(freq) {
  ; compute a complex feedback coefficient
  norm = Math:Pi / Rate-of(Audio:Clock)
  feedback-coef = Complex:Unitary(freq * norm)

  ; Complex:Unitary(w) returns a complex number
  ; with argument of 'w' and modulus of 1.

  ; initially, the complex waveform starts from
  ; phase 0
  initial = Complex:Unitary(0)

  ; Haskell-style section; an incomplete binary operator
  ; becomes an anonymous unary function, here closing over
  ; the feedback coefficient
  state-evolution = (* feedback-coef)

  ; the output of the oscillator is the real part of the
  ; complex sinusoid
  Sinusoid-Oscillator = Complex:Real(
    Generic-Oscillator(initial state-evolution))
}

Main() {
  ; receive user interface parameters
  fo = Control:Param("fo" 0)
  fdelta = Control:Param("fdelta" 0)

  ; number of oscillators; must be an invariant constant
  num-sines = #50

  ; generate the frequency spread
  freqs = Algorithm:Expand(num-sines (+ (fdelta + fo)) fo)

  ; apply oscillator algorithm to each frequency
  oscs = Algorithm:Map(Sinusoid-Oscillator freqs)

  ; sum all the oscillators and normalize
  sig = Algorithm:Reduce((+) oscs) / num-sines

  Main = sig
}
```

C.2 GENERIC SYNTAX GRAPH

The generic syntax graph is a representation of the format Kronos source is held in memory after parsing.

Figure 3 represents the parsed form of `Generic-Oscillator`. The return value is indicated for clarity only, and does not reflect a node in the parsed syntax tree.

Figure 4 shows `Sinusoid-Oscillator`. The parser performs a significant transformation, implementing lexical closures via partial application (Currying). Specifically, the function bound to `(* feedback-coef)` in the source code has become `Curry(Mul feedback-coef)`. Similar transformation is performed for all anonymous functions close over symbols defined in the parent scope.

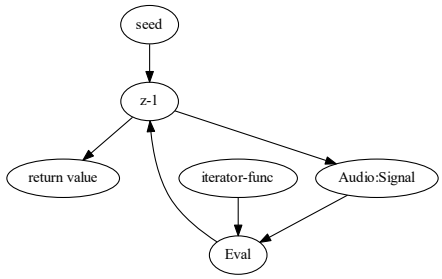

Figure 3: `Generic-Oscillator` abstract syntax tree

C.3 TYPED SYNTAX GRAPH

The typed syntax graph is the result of the specialization pass, described in Section 2.2.4. The typed form of `Sinusoid-Oscillator` is shown in Figure 5. Most notably, type information has been collapsed and is fixed into the nodes of this *typed signal graph*. Function arguments are represented by the `Argument` node, which delivers an algebraic composite of all the arguments for destructuring via `First` and `Rest`.

The invariant constant for `Math:Pi` is absent from the data flow. The division operator has become a call to the function `Div-Fallback`. This is because the division was between different types: an invariant constant (`Math:Pi`) and a floating point constant (output of `Reactive:Rate`). Ordinary division was rejected by type constraints and the fallback form was used instead. It peforms argument coercion to conforming types. Essentially, the library has generated a specialized function for dividing `Math:Pi` by a floating point signal.

As the feedback coefficient is captured by `state-evolution`, the Currying–Closure mechanism has substituted a `Pair` of the function and the curried argument. These are passed to `Generic-Oscillator` together with the `seed` signal, a complex-valued zero. The specialization of `Generic-Oscillator` is shown in Figure 6.

As discussed in `P6`, Kronos performs whole-program type derivation. In a type-variant recursive function such as `Algorithm:Map`, this can be expensive, as it scales linearly with recursion depth. To specialize common recursive sequences in constant time, Kronos contains a recursion solver that can identify *invariant type constraints* for *type-variant recursive functions*. For example, `Algorithm:Map` specializes very similarly for all its iterations. The output of the sequence recognizer is demonstrated in Figure 7. This shows an iteration of the sequence: the entire sequence is stored as a recurring body, repeat count and a terminating case.

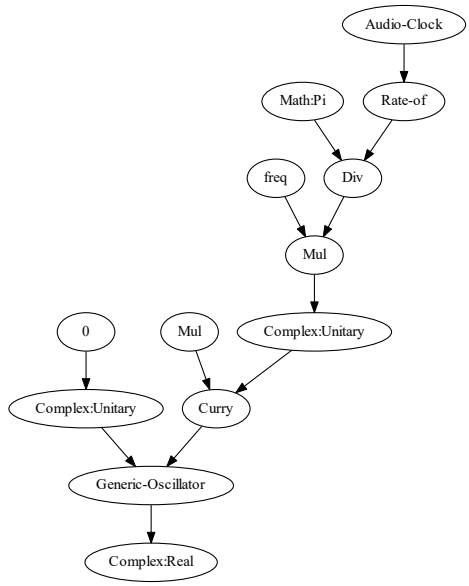

Figure 4: `Sinusoid-Oscillator` abstract syntax tree

C.4 REACTIVE ANALYSIS

The typed programs are subsequently analyzed for reactive data dependencies. To reduce compilation time, *call graph simplification* is done before this analysis. This step *inlines* trivial functions into their callers. Both steps are evident in the analyzed form of `Sinusoid-Oscillator`, shown in Figure 8. The calls to `Generic-Oscillator` and `state-evolution` have been inlined and are present as primitive operators: `Add`, `Mul`, `Sub` and `RingBuffer`.

Complex-numbered arithmetic has been lowered to work on pairs of floating point numbers. Structuring and destructuring are performed via `Pair`, `First` (for the real part) and `Rest` (for the imaginary part). The bulk of the audio rate processing in the example figure is the feedback loop around `RingBuffer[1]`, where the complex-valued content of the buffer is destructured, multiplied with the feedback coefficient, and structured back into the buffer.

The reactive analysis pass incorporates clock information into the typed graph. Clock region boundaries gain a node, shown as the four `Clock` nodes in the example figure. Each boundary node encodes information about which clocks are active up- and downstream of the boundary. For brevity, the example figure only shows the clocks that become active when moving downstream across the boundary. The clock regions themselves are color-coded in the figure. As a clarification, the root node of the graph is indicated by a node labeled as `output`. This node is for visualization purposes only and not present in the internal representation.

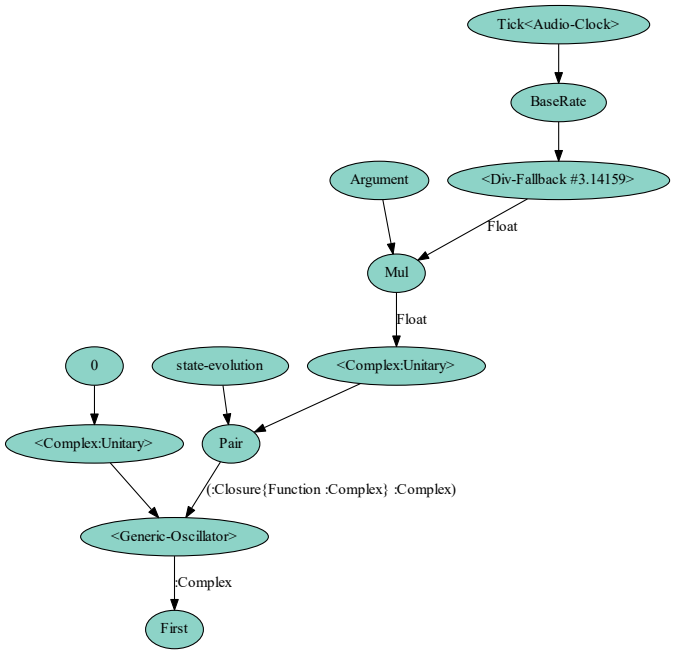

Figure 5: `Sinusoid-Oscillator` specialized form

C.5 SIDE EFFECT TRANSFORM

The Side Effect Transform (please see 2.2.4) facilitates further lowering of the program towards machine code. The abstract structuring and destructuring operations are converted to low-level pointer arithmetic. The recurring case in the call to `Algorithm:Reduce((+) oscs)` is shown in Figure 9, both in simple, typed form and after the Side Effect Transform. The `Recur` node in the figure corresponds with the point of recursion.

The left-hand side data flow is omitted by the Side Effect Transform, as it is a mere abstract passthrough of the reduction function `(+)` and contains no live signal data. The rest of the signal path obtains the two first elements of the argument list, adds them, and constructs a new list by prepending the result to the tail of the list. Kronos encodes the list as a pair of a raw floating point number, `f32` in the LLVM [27] vernacular, and a raw pointer, `i8*`. The new head is formed by `Add`, while the new tail is formed by a simple offset of the previous tail pointer by 4 bytes, which is the size of `f32`.

The third argument to `Recur` is the return value pointer. Side Effect Transform translates return values into side effects – instead of receiving a value from a callee, the caller passes in a memory location where the callee will place the results.

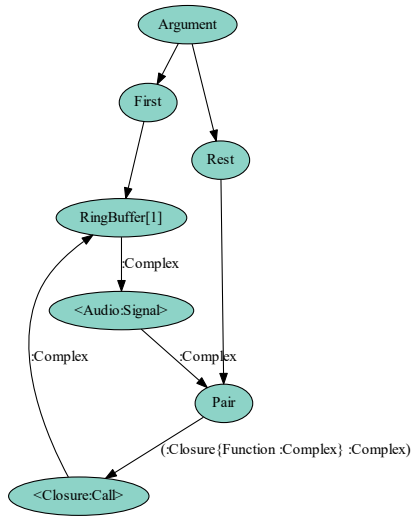

Figure 6: `Generic-Oscillator` specialized form

Kronos performs *copy elision* by propagating the return value pointer upstream from the function root. `Reduce` is tail-recursive, so copy elision passes the return value pointer, `out:i8*`, as is. The terminating case of `Reduce` will eventually write `arg1:f32` to this location.

The copy elision is general enough to perform a technique known as *tail recursion modulo cons*. Any `Pair` nodes at the root of the function will vanish from the side-effectful version, as they become pointer arithmetic on the return value pointers passed via copy elision. Therefore, many recursive functions that generate their output by structuring a recursive call into a list can be compiled in tail-recursive side-effectful form, even if their source code looks like it is not tail recursive.

Notably, the output of the Side Effect Transform is no longer functionally pure or referentially transparent. This means that for program correctness, the execution schedule of the nodes must be constrained. The transform accomplishes this by adding explicit dependencies into the graph. The complexity of the side-effectful graphs can escalate quickly, as is evident from Figure 10. This is the translated version of the recursive case of `Algorithm:Map`, containing the completely inline form of `Sinusoid-Oscillator`.

In addition to return values, cycles in the graph around delay operators are split into feedforward and feedback parts, the latter of which are attached to the root of the function as side effects on a shared memory location. The memory for these operations is reserved from a state pointer, `arg1:i8*`, which is threaded through all the stateful operations and functions in the graph.

Due to the semantics of the delay operators in the Kronos language, the contents of the ring buffer must not be written before all the operations on the previous data are completed. This ensures that the operators remain semantically pure, and that the mutation of state can not be seen

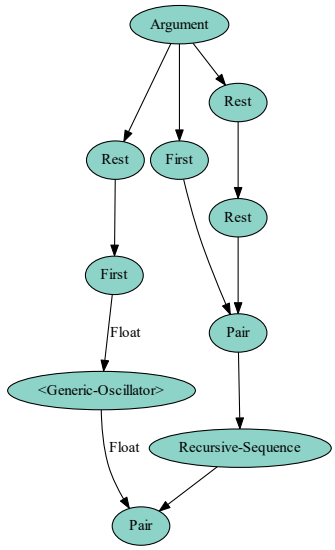

Figure 7: `Algorithm:Map` sequence specialization

by user programs. This is the source of the most complex *data hazard* in the Side Effect Transform – writes must be scheduled after reads. This is accomplished by explicit `Depends` nodes, which guard the write pointers passed to the side effects for the duration of any pure operation that dereferences the same location.

Most of the complexity is Figure 10 is due to data hazard protection. To make the graph more readable, edges that only convey dependencies and program order, and no data, are colored gray and dashed.

C.6 LLVM INTERMEDIATE

The Kronos LLVM [27] emitter produces assembly language in the LLVM IR format that closely matches the signal graph produced by the Side Effect Transform. The significant responsibility of the IR emitter is to schedule the computations: minimize the duration of buffer lifetimes and the number of live signals to aid the LLVM optimizer – as the most significant case, schedule potential function calls so that they can be tail-call optimized.

Listing 14: `Map(Sinusoid-Oscillator ...)` as LLVM IR

```
; Function Attrs: nounwind argmemonly
define private fastcc i8* @"Tickaudio_frame_0::Algorithm:Map_seq"(i8* noalias nocapture %self, i8
    * noalias nocapture %state, i8* noalias nocapture %p1, i8* noalias nocapture %p2, i32 %p3)
    unnamed_addr #0 {
Top:
```

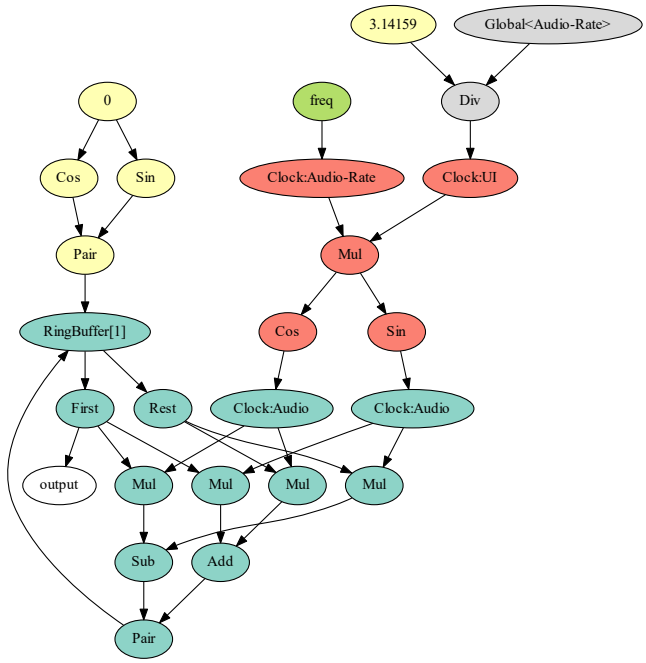

Figure 8: `Sinusoid-Oscillator` after reactive analysis

```
; offsets to the state buffer for local side effects
%0 = getelementptr i8, i8* %state, i64 8
%1 = getelementptr i8, i8* %0, i64 4
%2 = getelementptr i8, i8* %1, i64 4
%3 = getelementptr i8, i8* %2, i64 4
%4 = getelementptr i8, i8* %3, i64 4

; 'Rest' of input and output arrays
%5 = getelementptr i8, i8* %p2, i64 4
%6 = getelementptr i8, i8* %p1, i64 4

%7 = getelementptr i8, i8* %state, i64 4
%8 = getelementptr i8, i8* %state, i64 4
%9 = getelementptr i8, i8* %state, i64 4

; initializer signal for the RingBuffer
%10 = call float @llvm.sin.f32(float 0.000000e+00)
%11 = call float @llvm.cos.f32(float 0.000000e+00)

; destructure and load the complex number parts
; from state memory — both RingBuffer and clock
; boundary
%12 = bitcast i8* %8 to float*
%13 = load float, float* %12, align 4
```

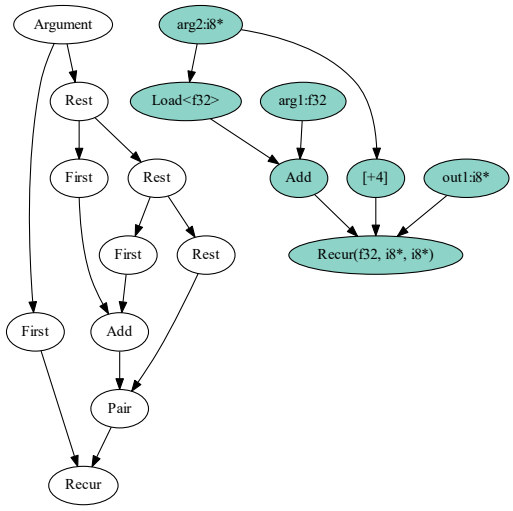

Figure 9: Side-effect transform on the `Reduce(Add ...)` sequence

```
%14 = bitcast i8* %8 to float*
%15 = load float, float* %14, align 4
%16 = bitcast i8* %state to float*
%17 = load float, float* %16, align 4
%18 = bitcast i8* %state to float*
%19 = load float, float* %18, align 4
%20 = getelementptr i8, i8* %state, i64 8
%21 = getelementptr i8, i8* %20, i64 4
%22 = bitcast i8* %21 to float*
%23 = load float, float* %22, align 4
%24 = getelementptr i8, i8* %21, i64 4
%25 = bitcast i8* %24 to float*
%26 = load float, float* %25, align 4
%27 = getelementptr i8, i8* %24, i64 4
%28 = bitcast i8* %27 to float*
%29 = load float, float* %28, align 4
%30 = bitcast i8* %20 to float*
%31 = load float, float* %30, align 4

; access the global variable for audio sampling rate
%32 = getelementptr i8, i8* %self, i64 mul (i64 ptrtoint (i1** getelementptr (i1*, i1** null,
     i32 1) to i64), i64 2)
%33 = bitcast i8* %32 to i8**
%34 = load i8*, i8** %33, align 4
%35 = bitcast i8* %34 to float*
%36 = load float, float* %35, align 4

; complex-number multiplication
%37 = fmul float %17, %29
%38 = fmul float %15, %29
%39 = fmul float %13, %26
%40 = fmul float %19, %26
```

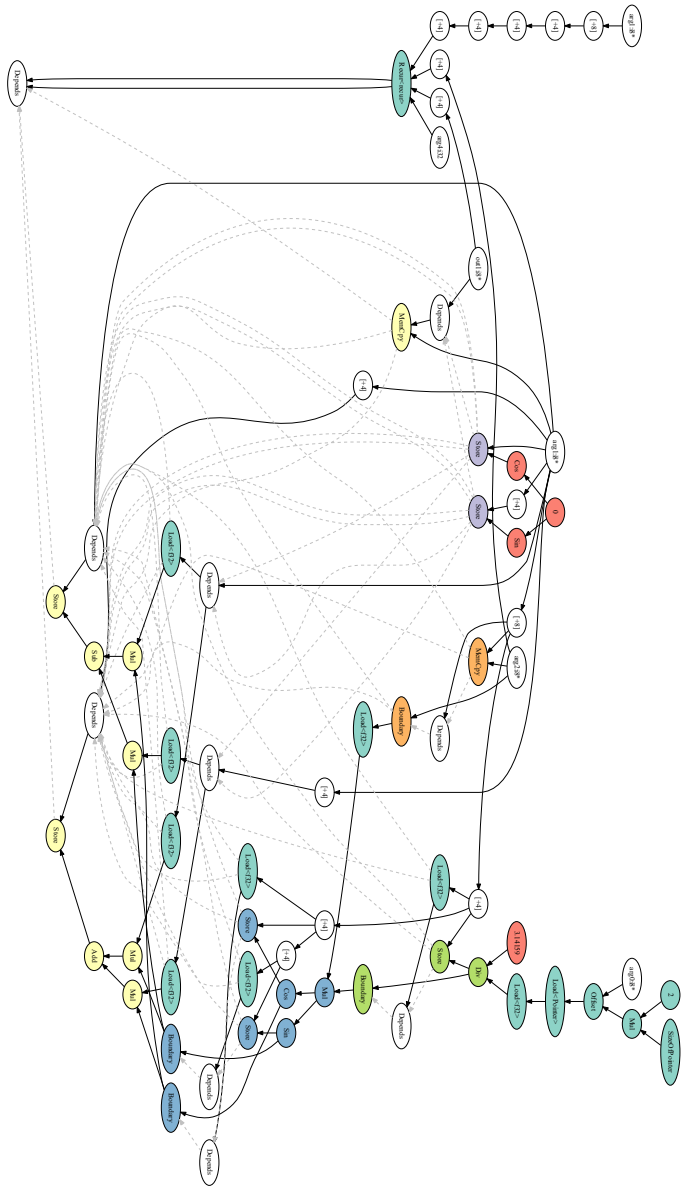

Figure 10: Side-effect transform on the `Map(Sinusoid-Oscillator ...)` sequence

```
  %41 = fadd float %37, %39
  %42 = fsub float %40, %38

  ; return value: real part from the ringbuffer memory
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %p2, i8* %state, i64 4, i32 4, i1 false)

  ; side effects: write the multiplication result to
  ; ringbuffer memory - after all the other operations
  %43 = bitcast i8* %7 to float*
  store float %41, float* %43, align 4
  %44 = bitcast i8* %state to float*
  store float %42, float* %44, align 4

  ; decrement loop counter and branch
  %45 = sub i32 %p3, 1
  %46 = icmp eq i32 %45, 0
  br i1 %46, label %RecursionEnds, label %Recursion

RecursionEnds:                                    ; preds = %Top
  %47 = tail call fastcc i8* @"Tickaudio_frame_0::tail"(i8* %self, i8* %4, i8* %6, i8* %5)
  br label %Merge

Recursion:                                        ; preds = %Top
  %48 = tail call fastcc i8* @"Tickaudio_frame_0::Algorithm::Map_seq"(i8* %self, i8* %4, i8* %6,
        i8* %5, i32 %45)
  br label %Merge

Merge:                                            ; preds = %Recursion, %RecursionEnds
  %49 = phi i8* [ %48, %Recursion ], [ %47, %RecursionEnds ]
  ret i8* %49
}
```

C.7 LLVM OPTIMIZED X64 MACHINE CODE

Finally, the LLVM [27] intermediate representation is passed to the LLVM backend for optimization and machine code generation. LLVM can generate machine code for various hardware targets via techniques such as pattern matching instruction selection and type legalization. Some optimization passes are general and reduce the likely computational complexity of the IR, while others depend on the features of the selected target architecture.

As an example, the 64-bit Intel architecture code generated for the vectored audio update routine for the Listing 13 is shown in symbolic assembly, in Listing 15. The oscillator bank loop starts at the label `.LBB5_3`, which contains several unrolled iterations of the recursive-complex multiplication. One oscillator update consists of four memory reads, three memory writes (of 32-bit floating point values each) and 7 other instructions. Additional four looping instructions are employed once for every three oscillators.

In total, the lifecycle demonstrates the complicated program transformations that Kronos employs in order to keep the overhead of high level, abstract programming to a bare minimum.

Listing 15: Audio update routine for the supplementary example

```
Tickaudio:
  sub rsp, 200
  test  r8d, r8d
  je   .LBB5_5
  lea r9, [rcx + 524]
  movss xmm0, dword ptr [rip + __real@3ca3d70a]
  .align  16, 0x90
.LBB5_2:
  movss xmm1, dword ptr [rcx + 408]
```

```
  movss xmm2, dword ptr [rcx + 412]
  movss xmm3, dword ptr [rcx + 424]
  movss xmm4, dword ptr [rcx + 428]
  movaps  xmm5, xmm1
  mulss xmm5, xmm4
  mulss xmm4, xmm2
  mulss xmm2, xmm3
  mulss xmm3, xmm1
  addss xmm2, xmm5
  subss xmm3, xmm4
  movss dword ptr [rsp], xmm1
  movss dword ptr [rcx + 412], xmm2
  movss dword ptr [rcx + 408], xmm3
  movss xmm1, dword ptr [rcx + 432]
  movss xmm2, dword ptr [rcx + 436]
  movss xmm3, dword ptr [rcx + 448]
  movss xmm4, dword ptr [rcx + 452]
  movaps  xmm5, xmm1
  mulss xmm5, xmm4
  mulss xmm4, xmm2
  mulss xmm2, xmm3
  mulss xmm3, xmm1
  addss xmm2, xmm5
  subss xmm3, xmm4
  movss dword ptr [rsp + 4], xmm1
  movss dword ptr [rcx + 436], xmm2
  movss dword ptr [rcx + 432], xmm3
  mov rax, r9
  mov r10d, 4
  .align  16, 0x90
.LBB5_3:
  movss xmm1, dword ptr [rax - 68]
  movss xmm2, dword ptr [rax - 64]
  movss xmm3, dword ptr [rax - 52]
  movss xmm4, dword ptr [rax - 48]
  movaps  xmm5, xmm1
  mulss xmm5, xmm4
  mulss xmm4, xmm2
  mulss xmm2, xmm3
  mulss xmm3, xmm1
  addss xmm2, xmm5
  subss xmm3, xmm4
  movss dword ptr [rsp + 4*r10 - 8], xmm1
  movss dword ptr [rax - 64], xmm2
  movss dword ptr [rax - 68], xmm3
  movss xmm1, dword ptr [rax - 44]
  movss xmm2, dword ptr [rax - 40]
  movss xmm3, dword ptr [rax - 28]
  movss xmm4, dword ptr [rax - 24]
  movaps  xmm5, xmm1
  mulss xmm5, xmm4
  mulss xmm4, xmm2
  mulss xmm2, xmm3
  mulss xmm3, xmm1
  addss xmm2, xmm5
  subss xmm3, xmm4
  movss dword ptr [rsp + 4*r10 - 4], xmm1
  movss dword ptr [rax - 40], xmm2
  movss dword ptr [rax - 44], xmm3
  movss xmm1, dword ptr [rax - 20]
  movss xmm2, dword ptr [rax - 16]
  movss xmm3, dword ptr [rax - 4]
  movss xmm4, dword ptr [rax]
  movaps  xmm5, xmm1
  mulss xmm5, xmm4
  mulss xmm4, xmm2
  mulss xmm2, xmm3
```

```
mulss xmm3, xmm1
addss xmm2, xmm5
subss xmm3, xmm4
movss dword ptr [rsp + 4*r10], xmm1
movss dword ptr [rax - 16], xmm2
movss dword ptr [rax - 20], xmm3
add r10, 3
add rax, 72
cmp r10d, 49
jne .LBB5_3
mov eax, dword ptr [rcx + 1544]
mov dword ptr [rsp + 188], eax
movss xmm1, dword ptr [rcx + 1548]
movd  xmm2, eax
movss xmm3, dword ptr [rcx + 1560]
mulss xmm3, xmm2
movss xmm4, dword ptr [rcx + 1564]
mulss xmm4, xmm1
addss xmm4, xmm3
mulss xmm2, dword ptr [rcx + 1552]
mulss xmm1, dword ptr [rcx + 1556]
subss xmm2, xmm1
movss dword ptr [rcx + 1544], xmm2
movss dword ptr [rcx + 1548], xmm4
mov eax, dword ptr [rcx + 1576]
mov dword ptr [rsp + 192], eax
movss xmm1, dword ptr [rcx + 1580]
movd  xmm2, eax
movss xmm3, dword ptr [rcx + 1592]
mulss xmm3, xmm2
movss xmm4, dword ptr [rcx + 1596]
mulss xmm4, xmm1
addss xmm4, xmm3
mulss xmm2, dword ptr [rcx + 1584]
mulss xmm1, dword ptr [rcx + 1588]
subss xmm2, xmm1
movss dword ptr [rcx + 1576], xmm2
movss dword ptr [rcx + 1580], xmm4
mov eax, dword ptr [rcx + 1608]
mov dword ptr [rsp + 196], eax
movss xmm1, dword ptr [rcx + 1612]
movd  xmm2, eax
movss xmm3, dword ptr [rcx + 1624]
mulss xmm3, xmm2
movss xmm4, dword ptr [rcx + 1628]
mulss xmm4, xmm1
addss xmm4, xmm3
mulss xmm2, dword ptr [rcx + 1616]
mulss xmm1, dword ptr [rcx + 1620]
subss xmm2, xmm1
movss dword ptr [rcx + 1608], xmm2
movss dword ptr [rcx + 1612], xmm4
movss xmm1, dword ptr [rsp]
addss xmm1, dword ptr [rsp + 4]
movss xmm2, dword ptr [rsp + 12]
addss xmm2, dword ptr [rsp + 8]
addss xmm2, dword ptr [rsp + 16]
addss xmm2, dword ptr [rsp + 20]
addss xmm2, dword ptr [rsp + 24]
addss xmm2, dword ptr [rsp + 28]
addss xmm2, dword ptr [rsp + 32]
addss xmm2, dword ptr [rsp + 36]
addss xmm2, dword ptr [rsp + 40]
addss xmm2, dword ptr [rsp + 44]
addss xmm2, dword ptr [rsp + 48]
addss xmm2, dword ptr [rsp + 52]
addss xmm2, dword ptr [rsp + 56]
```

```
  addss xmm2, dword ptr [rsp + 60]
  addss xmm2, dword ptr [rsp + 64]
  addss xmm2, dword ptr [rsp + 68]
  addss xmm2, dword ptr [rsp + 72]
  addss xmm2, dword ptr [rsp + 76]
  addss xmm2, dword ptr [rsp + 80]
  addss xmm2, dword ptr [rsp + 84]
  addss xmm2, dword ptr [rsp + 88]
  addss xmm2, dword ptr [rsp + 92]
  addss xmm2, dword ptr [rsp + 96]
  addss xmm2, dword ptr [rsp + 100]
  addss xmm2, dword ptr [rsp + 104]
  addss xmm2, dword ptr [rsp + 108]
  addss xmm2, dword ptr [rsp + 112]
  addss xmm2, dword ptr [rsp + 116]
  addss xmm2, dword ptr [rsp + 120]
  addss xmm2, dword ptr [rsp + 124]
  addss xmm2, dword ptr [rsp + 128]
  addss xmm2, dword ptr [rsp + 132]
  addss xmm2, dword ptr [rsp + 136]
  addss xmm2, dword ptr [rsp + 140]
  addss xmm2, dword ptr [rsp + 144]
  addss xmm2, dword ptr [rsp + 148]
  addss xmm2, dword ptr [rsp + 152]
  addss xmm2, dword ptr [rsp + 156]
  addss xmm2, dword ptr [rsp + 160]
  addss xmm2, dword ptr [rsp + 164]
  addss xmm2, dword ptr [rsp + 168]
  addss xmm2, dword ptr [rsp + 172]
  addss xmm2, dword ptr [rsp + 176]
  addss xmm2, dword ptr [rsp + 180]
  addss xmm2, dword ptr [rsp + 184]
  addss xmm2, dword ptr [rsp + 188]
  addss xmm2, dword ptr [rsp + 192]
  addss xmm2, dword ptr [rsp + 196]
  addss xmm2, xmm1
  mulss xmm2, xmm0
  movss dword ptr [rdx], xmm2
  add rdx, 4
  dec r8d
  jne .LBB5_2
.LBB5_5:
  add rsp, 200
  ret
```