KRONOS AS A VISUAL DEVELOPMENT TOOL FOR MOBILE APPLICATIONS

Vesa Norilo

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

ABSTRACT

Kronos is a programming language and a compiler suite, recently enhanced with a visual front end. It is designed to facilitate programming of digital signal processors for music. Kronos patches can be compiled either for real time playback or into an intermediate language such as C++, for integration in several third party frameworks.

This paper introduces the visual patcher for Kronos along with productivity-boosting aspects of functional programming and metaprogramming that are unique in the visual domain. Higher order functions, closures and captured variables are examined pertaining to visually programmed signal processors. As a case study, a resonator bank synthesizer is built in the Patcher and subsequently exported as a component for iOS SDK.

1. INTRODUCTION

Many creative aspects of music technology involve programming tasks. Instrument and effect design as well as live electronics depend on the ability of the creative professional to fashion custom signal processors.

There are several musical programming environments. Among them, Supercollider[4] is a good example of textual programming harnessed for musical DSP. Pure Data[7] offers a visual front end but is simplistic as a programming language. PWGL[3] features a proper language and visual interface, but the signal processing component is limited.

The aim of Kronos is to provide a powerful, easily approachable language and combine it with a visual user interface. The entire compiler suite is built around optimization algorithms that make it possible to write high-performance DSP programs in a high level language.

The rest of this paper is organized as follows; in Section 2, a brief description of the core language is given. In Section 3 the integration of textual and visual programming is discussed. Section 4 deals with translating user patches into code, with an iOS application as a case study. Finally, Section 5 concludes the paper and presents future avenues of research.

2. LANGUAGE OVERVIEW

The Kronos back end is a server application capable of compiling textual programs. Several compile targets are supported, including real time processing. This section gives a brief overview of the language in order to contextualize the visual patching environment. For an extended discussion, the reader is referred to previous publications[5].

A full-featured functional language is provided. Abstractions such as higher order functions and closures[2] are supported, allowing sophisticated constructs to be used.

2.1. The Functional Reactive Paradigm

A central efficency aspect of signal processing algorithms is the multirate problem. Significant optimization can be achieved by updating parts of the algorithm at lower update rates.

Kronos proposes functional reactive programming[8] as a solution to the multirate problem. Nodes can act as clock sources, causing the graph downstream to update synchronously. A priority system ensures that more important clocks can override less important ones; a user interface slider can be a local clock source, but once the control signal reaches the audio path, the audio clock should take over. The reactive system has previously been discussed in depth[5].

The reactive aspect is designed to be ignored by the beginner and intermediate user. The compiler includes a factorizer capable of optimizing user functions for heterogenous signal rates with no effort on the part of the programmer.

2.2. Type System

Kronos offers an algebraic type system[2] to deal with data structure. In addition, data can be decorated with user types to denote semantical meaning. For example, a vector of two real numbers could equally well stand for a complex number or a stereophonic signal. However, very different behavior is expected in these two cases. When multiplying, we expect the complex number case to follow the laws of complex arithmetic. For a stereophonic signal, a more sensible result is an element-wise multiplication.

By utilizing polymorphism[2], the appropriate multiplication can be selected automatically based on the type of the incoming signal. A function library can be designed so that the end user need not worry about data types, yet the functions behave as expected.

2.3. Generic Programming

Generic programming follows from the capabilities of the type system. At source level, Kronos programs are typeless and generic. For performance reasons they are statically typed during execution.

This is accomplished by a specialization pass reminiscent of the C++ template metacompiler[1]. When compiling a program, the data types of the signal flow are inferred throughout, and appropriate forms of polymorphic functions are selected at each point.

This process can generate versions of – *specialize* – all functions in the system for any given argument. The only requirement is that all the function calls the specialization candidate makes can themselves be specialized. Eventually this requirement will propagate to the leaves of the call tree, where arithmetic and logical primitives tend to be found. Defining those for a certain user type can thus enable previously written libraries to accept the said type.

This is very useful due to the number of permutations encountered in DSP. A single filter function can be used for single or double precision, real or complex, mono or multichannel signals or any combination of these.

3. VISUAL USER INTERFACE

Since Kronos is aimed at musicians and music technologists in addition of programmers, a visual patching environment is provided to lower the barrier to entry. The visual interface acts as a client, connecting to the back end server over OSC [9].

Every effort is made to provide feature parity between textual and visual programming. As certain tasks are more natural to perform in writing while others are better suited for visualization, programmers may choose which domain to work in.

3.1. Integration of Text and Patching

The leading principle of the visual interface is that any expression can be typed into a patch node. Whenever the user starts typing, the node creation widget pops up.

The free text entry guarantees that all language constructs are available to the programmer. However, it is just as confusing for the beginner as writing the entire program in textual form in the first place. This is addressed by the interactive suggestion list displayed during text entry.

Beginners can use the text entry widget like a live search box into a predefined menu hierarchy. Advanced users may wish to employ it more like a code completion aid. Further inlets can be added to the expression by inserting an inlet token prefixed with $, such as *$input*. The token itself is displayed as a tooltip for the inlet.

This scheme makes it simple to enter literals and constants, as the user may just type them in to obtain the desired node. Some examples of nodes are shown in Figure 1.

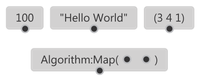

Figure 1. *Constant, literal, tuple and a function node with inlets.*

3.1.1. Patching

Nodes with inlets can be connected to other nodes via patch cords familiar to users of visual environments. In the generated program, connections are represented by variables. The value at the outlet is assigned to a unique symbol. Subsequently, all inlets connected to the outlet are replaced by references to this symbol.

3.2. Embedding

A visual connection is a powerful representation of data flow. However, sometimes the data flow can be so obvious or trivial that visualizing each connection only reduces readability. As an alternative to patch cords, the Kronos Patcher allows user to embed nodes into each other. By dragging a node into an inlet, the contents of that node replace the inlet.

Figure 2 shows an example, where a small subpatch is embedded entirely within a single node. Embedded nodes can be detached with a shake gesture.

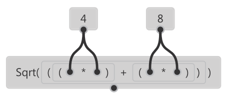

Figure 2. *Embedded nodes as a part of an expression.*

3.2.1. Lambda Arrow and Captured Variables

Higher order functions, anonymous functions and closures are essential to productive programming in the functional paradigm[2]. While well understood in textual format, their application to visual programming may be less obvious.

The most common way to construct anonymous functions in Kronos is to use the *lambda arrow* syntax. It is an infix operator =>, where the left hand side defines function arguments and the right hand side defines the body. A syntax such as $x => Sqrt(x)$ can be read as *x into square root of x*.

This lambda arrow results in an anonymous function that takes the square root of a real number. Such a function can then be passed to a higher order function like *Algorithm:Map*, which applies it over a vector of elements. Similar constructs are used for most iterative tasks.

An essential aspect of anonymous functions is the concept of *captured variables*. Traditionally, anonymous functions can refer to variables out of their scope. As the visual patch lacks the concept of variables, a slightly different method of capturing is required.

In the visual patcher, adding inlets to a lambda arrow expression results in a capture. A capturing inlet is demonstrated in Figure 3. A value from a slider and a lambda transformation form a closure which is then applied over a vector of numbers.

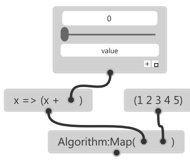

Figure 3. *Example of a captured inlet in an anonymous function*

3.3. Convenience Nodes

To assist in visual programming, several nodes are provided to help in situations where the desired effect is cumbersome to achieve with the visual interface.

3.3.1. Tuple

The Tuple node is created with a single inlet. Whenever something is connected or embedded into the socket, an additional socket is created automatically. A tuple data type is built from all the connected inlets.

3.3.2. Tie

Tie nodes are used to split algebraic types[2]. Created with the identifier *Tie*, it can deconstruct an algebraic type into distinct outlets.

3.3.3. Slider

A slider node allows the user to include an interactive control in the patch. For real time synthesis, the slider connects to the server back end via a dedicated OSC[9] method.

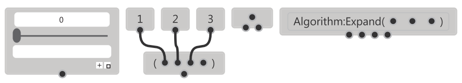

Figure 4. *Slider, Tuple, Tie and a multi-output node embedded in a Tie node*

4. CODE GENERATION

To run the patch, the user may issue a build command. The visual patch is then converted into a textual Kronos program, and sent to the server. The server will perform specialization, type inferral and reactive factorization as described in Section 2. Finally, executable code is generated.

Various code emitters can be added to the Kronos compiler back end. Currently, it supports immediate compilation and execution for the x86 architecture, designed for real time sound synthesis, as well as generation of portable code in C++. This is most useful for integration with third party software. As a case study, the integration of Kronos-generated C++ into a mobile iOS application will be examined.

4.1. Case study: an iOS Application

4.1.1. Designing a Patch

First, the design of a simple resonator bank synthesizer will be examined. The implemented filter is a typical two-pole biquad resonator. The outline of the patch is as follows;

- Compute coefficients from frequency and bandwidth
- Implement a Direct Form II biquad filter
- Use the resonator as a component of a filter bank

A function to compute resonator coefficients is given in Figure 5. Three argument inlets are specified: *signal*, *frequency* and *bandwidth*. The signal is required to obtain the sample rate via *Reactive:Sample-Rate*. Standard resonator coefficients *b1* and *b2* are computed and passed as a tuple to the function root node *Forms*.

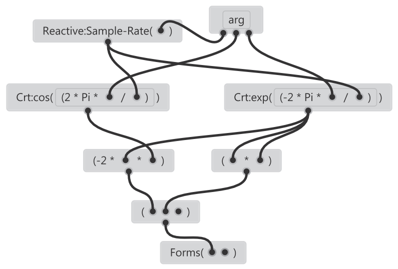

Figure 5. *Computation of resonator coefficients from frequency and bandwidth*

The implementation of a Direct Form II biquad is shown in Figure 6. It makes use of the previously defined coefficient computation routine, embedded in a *Tie* node to provide outlets for both coefficients. The filtering consists of two stages of unit delay into multiply-accumulate and recursion. The feed-forward section and peak gain normalization follow.

Finally, a filter bank is built out of the resonator. One slider will be used to adjust the bandwidth of all the filters, while a frequency control is assigned to each. A pseudo-random noise source will act as a excitation signal.

A lambda transformation is defined where the noise signal along with the bandwidth slider are *captured* (see 3.2.1). These signals partially specify a signal transform, leaving a single lambda argument, *freq*.

The subsequent function, *Algorithm:Map*, receives a vector of frequencies from a set of sliders embedded in a tuple node. The resulting vector is processed with the lambda transform to obtain four channels of resonant noise. The channels are summed by *Algorithm:Reduce*.

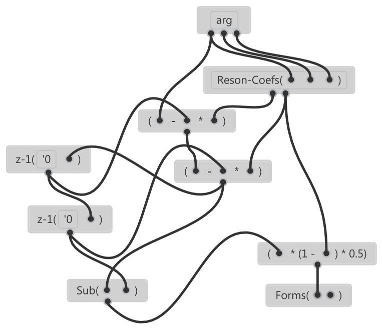

Figure 6. *Direct Form II realization of a biquad resonator*

4.1.2. Integration with XCode

XCode is the tool used to develop iOS applications. The Kronos C++-emitter converts the visual patch to code, and produces an application delegate class containing the signal processor for the iOS target. This way, Kronos can be used to develop signal processing algorithms while the refined user interface design tools of XCode kit can be fully applied.

To provide audio I/O and user interaction, the generated app delegate is annotated with *IBAction* keywords that are recognized by XCode. This allows the user interface elements to be connected visually in XCode Interface Builder. In fact, one can create a fully functional iOS application without writing a single line of code.

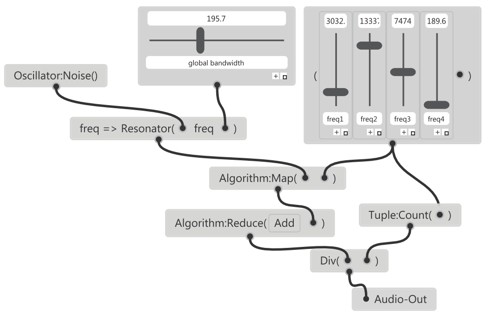

Figure 7. *A resonator bank controllable with sliders*

5. CONCLUSION

This paper has presented the current state of Kronos, a programming environment for musical signal processing. Its key features were discussed, and a visual user interface for comprehensive functional programming was presented. The system was applied to the task of generating a simple mobile audio application.

Many of the capabilities of Kronos are present in various software packages. Supercollider[4] supports a wide range of high level programming constructs. Faust[6] can generate efficient C-code for various frameworks. Pure Data[7] has a visual patching interface. However, none of these are able to combine all the features. Further, the generics and reactivity offered by Kronos are to the author's knowledge unique in the field.

Kronos is being actively developed, with the purpose of providing a capable set of DSP primitives in source form along with the compiler core. Backends are an active area of research, with more supported C++ targets on the way. In the medium term, code generation for massively parallel processors is a future avenue.

The Kronos environment will be released in the spring of 2012 as an open beta. This version enforces the GPL license for any generated code. After the beta period, the core compiler will remain free to use for open source software. The patching frontend and a commercially licensed compiler will also be available.

6. REFERENCES

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, ser. C++ In-Depth, B. Stroustrup, Ed. Addison Wesley, 2005.

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

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

[5] 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.

[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] 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.