Published as: Roger B. Dannenberg, "UGG: A Unit Generator Generator," in *Proceedings of the 2018 International Computer Music Conference,* International Computer Music Association, August 2018.

UGG: A Unit Generator Generator

Roger B. Dannenberg
Carnegie Mellon University
rbd@cs.cmu.edu

ABSTRACT

Unit generators are primary building blocks of music audio software. Unit generators aim to be both efficient and flexible, but these goals are often in opposition. As designers trade off efficiency against flexibility, many designs emerge, leading to a multitude of incompatible implementations. Thus, there are many incompatible unit generator libraries, each representing substantial effort. The present work suggests that unit generators can be written in a functional style using a conventional language with operator overloading, and an easily modifiable "back end" can generate efficient code. A prototype of this method, the Unit Generator Generator (UGG) system can be tailored quickly to target many unit generator designs. Computations can be shared across unit generators by defining simple functions, leading to an even more compact and expressive notation.

1. INTRODUCTION

In any computer music system for audio processing, most computation time is spent processing audio. Typically, audio processing is performed in a collection of *unit generators* that generate, filter, combine, and modify streams of audio samples. Since unit generators are critical to efficiency, they tend to have specialized and carefully designed lightweight interfaces so that the overhead of moving samples and control from one unit generator to the next is minimized.

Unfortunately, there is little or no compatibility between audio systems and the way they interface with unit generators. If efficiency were not so important, one could consider "compatibility layers" to adapt one set of conventions to another, but in practice, copying samples from one container or format to another can add substantial overhead. Instead, we see the development of many different libraries of unit generators, each optimized for a slightly different purpose.

One promising development has been high-level languages for describing audio signal processing. Typically, these languages use a functional programming style and treat signals as infinite sequences that are defined recursively. If a powerful, functional notation can be compiled to C or C++, then an optimizing compiler can finish the job of producing highly efficient code.

However, the functional language approach has been used mainly for monolithic signal processing programs such as audio plugins or complete synthesizers rather than smaller unit generators, and the "back end" or code generation part is not designed to target multiple different unit generator coding conventions.

The Unit Generator Generator (UGG) was developed to explore a new approach to high-level unit generator description. In UGG, one writes programs to create graphs that describe signal-processing algorithms to be implemented as unit generators. Operator overloading is used so that signal-processing graphs can be written intuitively and more-or-less as mathematical expressions. For example, the sum of signals A and B can be written A+B. In the UGG implementation, A, B and A+B would all be represented by objects, and operator overloading is used to define what it means to apply "+" to two objects. Thus UGG can be embedded in an existing language (Python), which reduces the need for users to learn a special language or syntax. The objects contain methods to generate code. Representing signals as objects allows users to customize code generation by extending or replacing code-generating methods. This keeps UGG lightweight, easy to understand, and adaptable so that one DSP algorithm expression can be used to generate any number of different styles of unit generators. There is no particular limit on the size of algorithms, so one could use UGG to build complete audio processing systems as well as simple unit generators.

We envision a typical use case to be a custom unit generator for an existing computer music language (e.g. Max or csound). The user first describes the algorithm in UGG and automatically generates code and a stand-alone test program. When testing is complete, the user automatically generates an "external" or library compatible with the preferred composition environment. If the unit generator is of general interest, the user can generate extensions for other systems such as Supercollider, Nyquist, Pd, and Chuck.

2. RELATED WORK

Most unit generators are coded directly in C or C++. Unit generator libraries can be found in Csound [1], Supercollider [2], and Pd [3][4], among others. As mentioned, languages use different conventions for unit generators. For example, Csound and Supercollider unit generators have block rate (k-rate) and audio rate (a-rate) versions whereas Pd does not.

Many unit generator libraries exist outside of specific language implementations. Putnam describes Gamma, a C++ library for unit generators [5]. This article also describes a number of other libraries including CLAM [6], CSL [7], ICST [8], and STK [9]. Putnam identifies three "main distinctions between the implementations of unit generators in these libraries: (1) processing granularity (single-sample and/or block-based), (2) support for processing generic types, and (3) ability to run at multiple sample rates," which accounts for some of the diversity and incompatibilities we see.

RATL [10] uses C++ templates to write unit generators in an object-oriented. RATL is motivated by trying to overcome incompatibilities between different unit generator-based systems. Nyquist [11] and Aura [12] describe unit generators in terms of parameters, types, various properties, and use a preprocessor to generate C implementatons. The idea behind these systems is to factor out commonalities so that only algorithm-specific code is required for each unit generator.

FAUST [13][14] and Kronos [15] are high-level languages based on functional programming that describe signal processing algorithms. In both cases, programs are expanded into C or LLVM to be compiled by an optimizing compiler. (FAUST can also generate code in a number of languages including Java and Javascript, and it can produce unit generators for several languages.) These systems may calculate audio in blocks at the top level, but do not support the creation of unit generators with a mix of audio rate and block rate signals.

Functional approaches often compile to functions that do the work of many unit generators. This is efficient, but then any changes to algorithms require recompilation of code, whereas unit generator systems (as produced by UGG) can be reconfigured without recompilation.

3. THE FUNCTIONAL APPROACH

Suppose we want a signal that is initially 0 and increases by 1 every 1000 samples. In procedural programming, we can simply write x ← x + 0.001 (using "←" to indicate assignment), and arrange to perform this assignment every sample period. However, at the risk of oversimplifying things, functional language descriptions of DSP algorithms use equations rather than sequential steps to describe computation, so we must somehow express how signals change over time without assignment statements. The "trick" to avoiding state changes is to describe signals as a *sequence* of values. While the syntax may vary from one language to the next, signals are described as a first element followed by elements that depend on previous elements. For example, we can say $x_0 = 0$ and $x_n = x_{n-1} + 0.001$.

This approach is not limited to a single variable. We can have any number of "state" variables, and the only restriction for the programmer is to describe the DSP algorithm as a set of equations (that more-or-less look like assignment statements) for the values of the next samples in terms of the values of previous samples.

One advantage of functional programs is that all data dependencies and thus order of execution is implicit in the equations. We can use this property to generate code in a very flexible manner.

Often, music signal processing combines signals running at different rates. In particular, if unit generators compute a block of 64 audio samples per activation at sample rate *SR*, it is fairly simple to introduce *control-rate* signals at *SR/*64 that control filter cutoff frequencies, amplitude envelopes, vibrato, and many other time-varying parameters.

Multi-rate computation is not simple in most functional programming languages and usually involves explicit programming to implement signal decimation or interpolation. Since we often want control-rate and audio-rate versions of unit generators, UGG handles mixed sample rate computations automatically, and just one specification can generate multiple unit generators. UGG considers all variables and signals to have one of three *rates*: audio rate (AR) denotes one value per audio sample. Block rate (BR) denotes a sequence of values where *one* new value is computed each time the unit generator's `run` method is called. Constant rate (CR) denotes a scalar value that remains constant over time. However, constant values can be updated between calls to the unit generator's `run` method.

4. CODING IN UGG

The Unit Generator Generator (UGG) uses objects to represent variables, values, and operators. There is no strong attempt to hide the implementation, and we will see that exposing the implementation gives UGG some interesting power.

Unit generator descriptions in UGG begin with a call to `ugg_begin("osc")`, where `"osc"` is the base name of the unit generator. One can then define variables by writing `Var("y",` *expr*`)`, where "`y`" is the variable name, and *expr* is an expression used to compute the value. Related to variables are parameters: `Param("x")` creates a parameter named `"x"`, which represents an input to the unit generator. `First("s", 0)` creates a signal named "`s`" with initial value 0. This must be paired with another declaration, `Next(s,` *expr*`)`, where *expr* describes the *next* value of signal "`s`" in terms of the *current* values of this an other signals. Various functions and operators are available. E.g. `A + B` denotes the sum of `A` and `B`, which can be scalar values or signals. Conditional execution can be defined functionally using `Cond(`*test*`,` *a*`,` *b*`)`, which returns value *a* or *b* depending on *test.* Unit generator code is produced by calling `ugg_write([`*r1*`,`*r2*`,…],` `[`*p1*`,`*p2*`,…],` *expr*`,` *rate*`)`, where *r1, r2,* ... are rates, *p1, p2,* ... are parameters, *expr* describes the output signal of the unit generator, and *rate* is the rate of the output. Rates for parameters and output can be either AR, BR, or CR as described in the previous section.

4.1 The Simplest Unit Generator

As a simple example, the following describes a unit generator that multiplies two audio rate signals:

```
ugg_begin("Mult")
pa = Param("a")
pb = Param("b")
ugg_write([AR,AR], [pa,pb], pa * pb, AR)
```

Notice the use of variables pa and pb. These are bound to parameter *objects* created by Param. In the expression pa*pb, "*" is overloaded; rather than performing any kind of arithmetic, it constructs a tree representation of the product of two parameters. This tree is then used to generate code. Due to space limitations, we will show just the generated constructor and run methods:

```
void Mult_aa_a(Ugen a, Ugen b) {
   this->b = b;
   this->a = a; }
void run() {
   float *b_samps = b->get_outs();
   float *a_samps = a->get_outs();
   for (int i = 0; i < BL; i++) {
      out[i] = (a_samps[i] * b_samps[i]); }}
```

4.2 Recursively Defined Signals as "State"

The next example illustrates how unit generators that retain and modify state can be expressed and generated using the functional style of UGG:

```
ugg_begin("Phasor")
freq = Param("freq")
phase = First("phase", 0, "double")
phase.use_output_rate()
Next(phase, phase + freq / SR)
ugg_write([CR], [freq], phase, AR)
```

Note that phase is created by First with an optional type declaration (the default is float). Since phase depends only on CR variables (freq and SR), UGG will assume that it is a block rate "control" variable. phase.use_output_rate() tells UGG to use the same rate as the unit generator's output, which in this case is audio rate. Then, we complete the recursive definition in Next: each *next* value of phase is computed from the given expression in terms of the *current* value of phase (and other variables). An abbreviated listing of the generated code follows:

```
void Phasor_c_a(float freq) {
 t10 = (freq / SR);
 this->freq = freq;
 phase = 0; }
void run() {
 for (int i = 0; i < BL; i++) {
  out[i] = phase;
  double phase_next = fmod((phase + t10), 1);
  phase = phase_next; }}
```

Because freq / SR depends only on constant terms (CR) but is used in an audio rate computation (to compute phase), the expression is assigned to a new variable (t10), which moves the division out of the inner loop. It may also seem curious that the next value of phase is stored in phase_next and then assigned to phase. This style of code generation can compute multiple "next" values for these recursively-defined signals before the current values are overwritten. In this case, the assignment to phase_next is unnecessary, and the compiler will remove it.

4.3 Alternate Versions for Different Rates

UGG can generate different implementations based on the rates of input parameters and the output signal. For example, we can use our multiplier definition (Section 4.1) to multiply an audio rate by a block rate parameter:

```
ugg_write([AR,BR], [pa,pb], pa * pb, AR)
```

Notice that there are no changes to the DSP algorithm itself. We merely indicate that the second parameter is BR instead of AR. UGG will detect that the block rate parameter pb is used in an AR expression (pa*pb) and automatically introduces in-lined code to linearly interpolate the block rate parameter to audio rate. (Interpolation can be removed with the use_non_interpolated method.) Here is the gist of the generated code:

```
void Mult_ab_a(Ugen a, Ugen b) {
   b_arate = 0;
   this->b = b;
   this->a = a; }
void run() {
   float *a_samps = a->get_outs();
   float b_step =
      (b->get_out() - b_arate) * BL_RECIP;
   for (int i = 0; i < BL; i++) {
      out[i] = a_samps[i] * b_arate;
      b_arate += b_step; }}
```

In this version, a single sample is fetched from b before the inner loop (thus at the block rate), and the inner loop interpolation is a simple increment by b_step. Note also that the base name, Mult, has been "decorated" with rates to indicate parameter rates and output rate, forming the class name Mult_ab_a. In a similar way, we can generate Phasor_c_b, a phasor that runs at the block rate.

5. CODE GENERATION

UGG generates code in three steps. First, the UGG program (expressed in Python) is evaluated, creating a dataflow graph that describes the DSP algorithm. Next, the graph is copied and altered to introduce new variables and operators to implement changes in sample rate and to avoid audio rate computation when control rate will do. Third, the graph is walked multiple times to generate code snipits to fill in an overall unit generator code template.

5.1 Building the Dataflow Graph

The basic elements of UGG DSP algorithms are objects. The constructors Param, Var and First are designed to look like declarations, but they actually create objects. Operators such as +, -, * and / are overloaded to create objects of class UBinary. Thus, our phasor unit generator (Section 4.2) produces the graph shown in **Figure 1**.

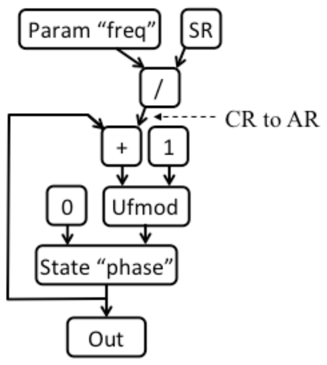

Figure 1. Dataflow graph for the phasor unit generator.

5.2 Introducing New Variables and Operators

In the next code generation step, the expression graph is topologically sorted and traversed from sources to sinks, and each node is labeled with a rate derived from its inputs. An exception in this case is that the method `use_output_rate` forces `phase` to be treated as AR.

When a rate change occurs, such as the one labeled "CR to AR" in **Figure 1**, a new node is inserted: either a variable is created to allow the value to be computed outside and reused inside the inner loop, or a node of class `Upsample` is added to interpolate from block rate to audio rate. In this case, the variable `t10` is created (see the generated code in Section 4.2).

5.3 Template-based Generation

Finally, code is generated by traversing the graph, again in order of sources to sinks so that values are generated before they are used in the sequential C++ code. Generation follows a code template such as the one shown below, which is used for audio-rate output unit generators:

```
class basename_decorations : Ugen_outa {
    declarations
    void classname(parameters) {
        constructor_stmts }
    void run() {
        brate_code
        upsample_prep
        for (int i = 0; i < BL; i++) {
            arate_code
            out[i] = output_expression_code;
            next_arate_state_code
            upsample_update
            update_arate_state }  }
```

In general, each point in the template, e.g. *constructor_stmts*, is generated by a full pass through the graph, invoking a method for each node. In this case, the method is `gen_constructor()`, which is inherited and does nothing in most classes, but, for example, a `Param` node, if declared to have rate CR, will generate code such as:

```
this->freq = freq;
```

Expressions such as `fmod((phase + t10), 1)` are generated from multiple objects by recursively constructing and combining sub-expressions. For example, the `UBinary` object's (slightly simplified) `gen_code` method (here "+" denotes string concatenation) is:

```
def gen_code(self):
  return self.left.gen_code() + self.op +
         self.right.gen_code()
```

Allowing each class to generate class-specific code, and using a different method to generate each code snipit in the template results in a simple architecture that can be modified or extended to meet different requirements.

6. CUSTOM CODE TARGETS

One of the drawbacks of existing libraries is that architectural decisions can affect the code of every unit generator. It is difficult for a library or even a language like FAUST to cover every desirable feature. Therefore, there is some value in a simple, very adaptable approach.

For example, some unit generator systems construct a unit generator graph, topologically sort the unit generators according to dependencies, and then compile and save an execution sequence. Unit generators can then assume without checking that their input samples are valid each time their "run" method is called. Alternatively, unit generators can check for valid inputs and call run methods (recursively) as needed, effectively performing a topological sort as each block is computed. This approach might make sense if unit generator graphs can be modified on the fly.

As an experiment, we modified our UGG code generator to insert checks and eliminate the need to call each unit generator explicitly. A single call to the "output" unit generator's run method brings all inputs up to date. Only 14 lines of code were added or modified to make this change. Admittedly, this *should* be a small change. What about a *large* change such as targeting Javascript instead of C++? To give some idea of the magnitude of this change, there are only 44 lines of code in UGG that explicitly depend upon C++. This number might be much larger if many more built-in functions and operators were added to UGG, but these added declarations would be very simple and repetitive.

7. POLYMORPHISM

UGG generates a unit generator specialized to each possible combination of parameter sample rates. E.g. we might create multipliers to multiply AR×AR, AR×BR, AR×CR, BR×BR, and BR×CR (or even more if we ignore the commutative property or make interpolation an option). One could always up-sample parameters to audio rate before connecting them to an audio rate unit generator, but that is much less efficient. One of the goals of UGG is to generate a polymorphic "front end" so that users can write something simple, such as `mult(a, b)`, and let `mult` find the optimal unit generator based on the types of `a` and `b`.

The details depend on the language in which unit generators are embedded. For example, if one is combining unit generators directly in C++, then function overloading can be used, and the compiler itself can select unit generators. On the other hand, if UGG unit generators are linked into a dynamically typed language, it is up to the caller to inspect parameter types and select from a number of implementations. As with variants of unit generators, this code is tedious to write, so automation can save a lot of work and minimize errors.

As a proof of concept, we imagined a synthesis system based on Python, extended with unit generators created by UGG. (One can imagine doing something similar with Pd [4], Supercollider [2], ChucK [16], or other languages.) Unit generators are represented by objects in Python, and we want polymorphic functions so that we can write generic expressions such as `Mult(Osc(…), Env(…))` rather than type-specific expressions such as `Mult_ab_a(Osc_c_a(…), Env_cccc_b(…))`.

Since the required code is simple and repetitive, we extended UGG with a function that creates a family of unit generators with different parameter rates. The new function: (1) computes permutations of parameter rates (AR, BR, CR), (2) calls `ugg_write` to generate a unit generator for each permutation, (3) writes a polymorphic unit generator (in Python) where the parameters can be any combination of AR, BR, and CR. The function finds and creates an actual unit generator implementation with matching parameter types. Thus, we can use UGG not only for unit generators but also to generate a convenient API to use them. Keeping UGG small and simple makes this sort of customization a very reasonable task compared to writing interfaces by hand.

8. ABSTRACTION IN UGG

One if the interesting possibilities of UGG arises from its embedding in an "ordinary" language (Python). While one might criticize UGG for being just a set of objects and coding conventions rather than a true language, we view this as a feature rather than a shortcoming. The previous section described how UGG supports targeting different systems merely by making simple changes to UGG itself, leaving DSP algorithms untouched.

Another possibility is using functions in Python to represent DSP algorithms in a more abstract form. For example, we presented a *phasor* algorithm that simply ramps from 0 to 1 at a given frequency. In practical cases, we might use a phasor in combination with a table lookup to generate a tone, or use multiple phasors in an FM algorithm. Can we abstract the notion of *phasor* without making a full-blown unit generator?

In fact, this is quit simple using ordinary Python functions and variables in a direct manor. Consider this function definition (in Python):

```
def phasor(freq):
    phase = First("phase", 0, "double")
    phase.use_output_rate()
    Next(phase, phase + freq / SR)
    return phase
```

Now we have a function that constructs and returns the phasor computation. To use this alone as a unit generator, we can write:

```
ugg_begin("Phasor")
freq = Param("freq")
ugg_write([CR], [freq], phasor(freq), AR)
```

More interestingly, we can combine `phasor` with other abstractions. For example, we could define `sinetone` to take a phasor as input and return an interpolated table lookup (as a signal). Then, we could define `lowpass` to filter the `freq` parameter so that sudden frequency changes produce slow frequency sweeps in the phasor and oscillator. The whole construction could then be written as:

```
ugg_begin("SweepOsc")
freq = Param("freq")
swr = Param("sweeprate")
ugg_write([BR, BR], [freq, swr],
   sinetone(phasor(lowpass(freq),swr)), AR))
```

In this example, the `freq` and `sweeprate` parameters are now block rate (`BR`) to make the point that this form of abstraction does not "lock in" rates, and code generation still performs a global resolution and optimization of rates, necessary variables and interpolation operations.

One potential problem is that programs written in this way can "blow up" into large blocks of in-lined code. To some extent, this is a feature that reduces overhead at runtime. On the other hand, because UGG generates readable high-level code and the code generation is easy to modify, we believe users can alter code generation if necessary to control code size.

9. FUTURE WORK

UGG already seems useful as is, but future work might consider a number of extensions. One missing optimization from UGG is common sub-expression elimination. If an operator graph is used in two places, it is possible that the graph will be converted into two or more copies of the same code. Wherever a graph node is referenced more than once and is not a variable or parameter, the computation should be stored in a variable and computed just once. If the graph contains actual copies of subgraphs (not just multiple references to the same node), finding them is more difficult, but the same optimization applies.

Spectral processing is a challenge for computer music languages. One approach is to introduce synchronous processing at some fraction of the block rate, i.e. the FFT analysis/synthesis rate. It seems possible to extend UGG with a "spectral rate" below the block rate. Both ChucK and LC have introduced support for operating at arbitrary hop sizes on vectors of samples, supporting granular synthesis, frequency domain processing with overlapping windows, and other interesting algorithms [17][18]. It would be interesting to explore this approach using sample synchronous unit generators (as in ChucK and LC). Alternatively, one could investigate how to combine this approach with block-based unit generators.

Similarly, asynchronous output should be supported, such as sending a message when a peak is detected. This can be implemented by defining operators that conditionally call a side-effect-producing function.

FAUST has introduced vector-based processing to take advantage of modern vector instructions [19]. This is not always optimal because of sample-to-sample dependencies, but the UGG representation of algorithms should allow a similar approach to vector code generation.

UGG should have "back-end" methods to generate code for a wide range of languages and systems. We also plan to develop a small but extensible UGG-based sound server for games, installations, and other applications where sound synthesis is an embedded component of a larger system.

10. CONCLUSIONS

UGG offers a way to describe unit generators in terms of equations rather than low-level coding in C++. Equations are represented as graphs of objects, leading to a very compact system for automatic code generation. UGG is capable of dealing with multiple sample rates for efficiency and multiple unit generator designs for flexibility. Unit generators can be optimized for combinations of constant, block-rate and audio-rate inputs and outputs, and UGG can generate polymorphic functions that select efficient implementations from among the options. In principle, UGG can target a variety of unit generator libraries, standards, and language implementations, allowing DSP code to be shared, maintained, and extended more easily.

Acknowledgments

The author would like to thank Vesa Norilo for interesting conversations and insights. Also thanks to the authors of many computer music systems for creating the design space from within which this research has grown.

11. REFERENCES

[1] V. Lazzarini, S. Yi, J. ffitch, J. Heintz, Ø. Brandtsegg, and I. McCurdy. *Csound: A Sound and Music Computing System*, Springer, 2016.

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

[3] M. Puckett, "Pure Data," *Proceedings, International Computer Music Conference*, ICMA, 1996, pp. 224-227.

[4] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4 (Winter), 2002, pp. 31-43.

[5] L. Putnam, "Gamma: A C++ Sound Synthesis Library Further Abstracting the Unit Generator," *Proceedings of the ICMC/SMC 2014*, ICMA, 2014, pp. 1382-1388.

[6] X. Amatriain, *An object-oriented metamodel for digital signal processing with a focus on audio and music*. (Ph.D. dissertation.) Barcelona, Spain: Universitat Pompeu Fabra, 2005.

[7] S. Pope, X. Amatriain, L. Putnam, J. Castellanos, and R. Avery. "Metamodels and design patterns in CSL4." *Proceedings of the 2006 International Computer Music Conference*, ICMA, 2006.

[8] S. Papetti, "The icst dsp library: A versatile and efficient toolset for audio processing and analysis applications." *Proceedings of the 9th Sound and Music Computing Conference*, 2012, pp. 535–540.

[9] P. R. Cook, G. P. Scavone. "The synthesis toolkit (STK)," *Proceedings of the 1999 International Computer Music Conference*, ICMA, 1999.

[10] K. MacMillan, M. Droettboom, and I. Fujinaga. "A system to port unit generators between audio DSP systems." *Proceedings of the International Computer Music Conference*, ICMA, 2001, pp. 103-106.

[11] R. B. Dannenberg, "The Implementation of Nyquist, a Sound Synthesis Language," *Computer Music Journal*, vol. 21, no. 3 (Fall), 1997, pp. 71-82.

[12] R. B. Dannenberg, "Combining Visual and Textual Representations for Flexible Interactive Signal Processing," *The ICMC 2004 Proceedings*, ICMA, 2004.

[13] Y. Orlarey, D. Fober, and S. Letz, "FAUST: An Efficient Functional Approach to DSP Programming," *New Computational Paradigms for Computer Music*. Editions Delatour, 2009.

[14] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and Semantical Aspects of Faust." *Soft Computing* vol. 8, no. 1, Springer-Verlag, 2004, pp. 623-632.

[15] V. Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, vol. 39, no. 4 (Winter), 2015, pp. 30-48.

[16] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4 (Winter), 2015, pp. 10-29.

[17] H. Nishino, N. Osaka, and R. Nakatsu. "The Microsound Synthesis Framework in the LC Music Programming Language." *Computer Music Journal*, vol. 39, no. 4 (Winter), 2015, pp. 49-79.

[18] G. Wang, R. Fiebrink, and P. Cook, "Combining Analysis and Synthesis in the ChucK Programming Language," *Proceedings of the 2007 International Computer Music Conference*, ICMA, 2007.

[19] Y. Orlarey, D. Fober, and S. Letz, "Adding Automatic Parallelization to Faust," *Linux Audio Conference*, 2009.