Stride: A Declarative and Reactive Language for Sound Synthesis and Beyond

Joseph Tilbian
jtilbian@mat.ucsb.edu

Andrés Cabrera
andres@mat.ucsb.edu

Media Arts and Technology Program
University of California, Santa Barbara

ABSTRACT

Stride is a declarative and reactive domain specific programming language for real-time sound synthesis, processing, and interaction design. Through hardware resource abstraction and separation of semantics from implementation, a wide range of computation devices can be targeted such as microcontrollers, system-on-chips, general purpose computers, and heterogeneous systems. With a novel and unique approach at handling sampling rates as well as clocking and computation domains, Stride prompts the generation of highly optimized target code. The design of the language facilitates incremental learning of its features and is characterized by intuitiveness, usability, and self-documentation. Users of Stride can write code once and deploy on any supported hardware.

1. INTRODUCTION

In the past two decades we have witnessed the rise of multiple open-source electronic platforms based on embedded systems. One of the key factors for their success has been in the simplifications made to programming their small computers.

By the nature of their design, they have mainly targeted physical computing and graphics applications. Audio has usually been made available through extensions. Although solutions leveraging existing operating systems and languages exist, what we have not seen yet is a full featured, audio-centric, multichannel platform capable of high resolution, low latency, and high bandwidth sound synthesis and processing. We attribute this to the lack of a high level domain specific programming language (DSL) targeting such a platform. All popular DSLs in the music domain have been designed to run on computers running full featured operating systems. Stride was conceived and designed to address this problem, enabling users to run optimized code on bare metal.

2. APPROACH

The field of DSLs for sound and music composition is old and crowded. To design a modern and effective language, multiple design requirements need to be addressed.

Copyright: ©2016 Joseph Tilbian et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

For the instrument designer, sound artist, or computer musician the language must simplify or unify the interface between language entities such as variables, functions, objects, methods, etc. It must simplify interaction programming and enable parallel expansion of its entities and interfaces.

From the perspective of digital signal processing, the language must be able to perform computations on a per sample basis, on real and complex numbers, in both time and frequency domains. It must also handle synchronous and asynchronous rates.

To take advantage of the current landscape of embedded and heterogeneous systems in an efficient manner, the language must abstract hardware resources and their configuration in a general and simple way. It must abstract the static and dynamic allocation of entities as well as threading, parallelism, and thread synchronization. It must also enable seamless interfacing of its entities running at different rates.

While designing Stride, the intuitiveness of the language as well as the experience of writing programs, by beginner and advanced users alike, topped the requirements mentioned above and both had a profound impact on its syntax design.

3. LANGUAGE FEATURES

A central consideration during the design of Stride was to treat the language as an interface and try to make it as "ergonomic" as possible. Two other criteria were *readability* and *flow*. That is, users should not need to read documentation to understand code and should be able to write code with as little friction as possible as the language works in a "physically intuitive" way similar to interfacing instruments, effects processors, amplifiers, and speakers in the physical world. To achieve this, features from popular and widely used general purpose and domain specific languages were incorporated into Stride, like:

- Multichannel expansion from SuperCollider [1]
- Single operator interface and multiple control rates from Chuck [2]
- Per sample processing and discarding control flow statements from Faust [3]
- Polychronous data-flow from synchronous and reactive programming languages like SIGNAL [4]
- Declarations and properties from Qt Meta Language
- Slicing notation for indexing from Python
- Stream operator from C++

The syntax of Stride is easy to learn as there are very few syntactic constructs and rules. Entities in the language are self-documenting through their properties, which expose the function of the arguments they accept. The choice of making Stride declarative was to separate semantics from any particular implementation.

The novel and unique aspect of Stride is making *rates* and *hardware computation cores* an intrinsic part of the language by introducing *computation domains* and synchronizing rates to them. This concept enables the distribution of various synchronous and asynchronous computations, encapsulated within a single function or method, to execute in different interrupt routines or threads on the hardware. The domains can potentially be part of a heterogeneous architecture. Rather than just being a unit generator and audio graph management tool, Stride enables the user to segment computations encapsulated in a unit generator during target code generation while handling it as a single unit in their code. Stride also features reactive programming, which enables complex interaction design.

This document presents a broad introduction to Stride, leaving many details out in the interest of space.

3.1 Language Constructs

There are two main constructs to the language: *Blocks* and *Stream Expressions*. Blocks are the building entities of the language while stream expressions represent its directed graph.

3.1.1 Blocks and Stream Expressions

Blocks are declared through a block declaration statement. They are assigned a *type* and a unique *label*. Labels must start with a capital letter and can include digits and the underscore character. A block's properties, discussed in detail in § 3.1.3, are part of the declaration and define its behavior. Code 1 shows a block declaration statement of type *signal* with default property values. The signal block is labeled *FrequencyValue*.

```
signal FrequencyValue {
  default:  0.0
  rate:     AudioRate
  domain:   AudioDomain
  reset:    none
  meta:     none
}
```

Code 1: A *signal* block declaration statement with default properties

Blocks exchange *tokens* either synchronously or asynchronously through *ports*. Tokens represent a single numeric value, a Boolean value, or a character string. The number and types of ports of a block depends on its type. A block has primary and secondary ports. Primary ports are accessible through a block's label while secondary ports are accessible through its *properties*. Connections between primary ports are established in stream expressions using the *stream operator* (>>). Connections between primary and secondary ports are either established during a block's declaration or during invocation in stream expressions.

Code 2 is a stream expression where the primary ports of the *Input*, *Process*, and *Output* blocks are connected using the stream operator. A secondary port of the *Process* block, exposed through a property called *control*, is connected to a primary port of the *Value* block.

```
Input >> Process ( control: Value ) >> Output;
```

Code 2: A stream expression with four block connections

Stream expressions must end with a semicolon. They are evaluated at least once from left to right and in the top-down order in which they appear in the code.

3.1.2 Block Types

Block *types* are categorized into three groups: *Core*, *Auxiliary*, and *Modular*.

The core blocks are *signal*, *switch*, *constant*, *complex*, *trigger*, and *hybrid*. The signal block, discussed in detail in § 3.1.4, is the principal element of the language. It dictates when (the rate of token propagation) and where (the computation domain) computations occur within a stream expression. The *switch* block abstracts a toggle switch. It is asynchronous and can have one of two states: *on* or *off*, both keywords in Stride. The *trigger* block can trigger *reaction* blocks, allowing reactive programming within an otherwise declarative language. The *complex* block represents complex numbers and facilitates performing computations on them. The *hybrid* block enables the abstraction of port types allowing compile time type inference, akin to templates in object-oriented languages.

The auxiliary blocks are *dictionary* and *variable*. The dictionary block holds key and value pairs. The variable block dynamically changes the size of core block bundles, discussed in §3.1.5, enabling dynamic memory management.

The modular blocks are *module* and *reaction*. They encapsulate blocks and stream expressions to create higher level functions and reactions respectively. Unlike module blocks which operate on one token at a time, reaction blocks, when triggered, continuously execute until stopped when certain criteria are met.

3.1.3 Ports and Tokens

Ports have a direction and a type. A port's direction can either be *Input* or *Output*. Blocks receive or sample tokens through input ports and broadcast them through output ports. There are eight port types in total. A port's type is defined by two attributes. Each attribute is an element from the following two sets: {*Constant*, *Streaming*} and {*Real*, *Integer*, *Boolean*, *String*}.

The validity of connections between ports is determined by their types. Automatic type casting takes place between certain port types. Only a single connection can be established with an Input port while multiple connections can be established with an Output port. Constant Output ports can be connected to Streaming Input ports but Streaming Output ports can not be connected to Constant Input ports. Real, Integer and Boolean ports can be connected to each other but not to String ports and vice versa. Boolean Output port tokens are treated as 0 or 1 at Integer Input ports and as 0.0 or 1.0 at Real Input ports. Integer and Real Output port tokens with values 0 or 0.0 respectively are treated as false at Boolean Input ports while tokens with any other value are treated as true. Integer Output port tokens are cast to real at Real Input ports while Real Output port tokens are truncated at Integer Input ports.

3.1.4 *The Signal Block*

The signal block has five properties as shown in Code 1. The *default* property sets the block's default value as well as the primary port types to Streaming Integer, Streaming Real, or Streaming String depending on whether the value is an integer, a real or a string. The value assigned to the *rate* property sets the block to run either in synchronous or asynchronous mode. When assigned an integer or a real value it runs in synchronous mode and when assigned the keyword *none* it runs in asynchronous mode. The *domain* property sets the computation domain of the block and synchronizes its rate to the assigned domain's clock. The *reset* property resets the block to its default value when a trigger block assigned to it is triggered. All blocks in Stride have a *meta* property used for self documentation. It can be assigned any string value.

3.1.5 *Block Bundles*

Blocks can be bundled together to form *block bundles*. The primary ports of the bundled blocks are aggregated to form a single interface. Individual ports, or a set of ports, of the interface can be accessed by indexing. Indexing is not zero-based, but starts at 1. The square brackets are the bundle indexing and bundle forming operator. Core block bundles can be formed during declaration by specifying the bundle size in square brackets after the block's label. Bundles can also be formed in stream expressions by placing blocks or stream expressions in square brackets separated by commas.

3.2 Platforms and Hardware

Since Stride is a declarative language, a backend is required to translate Stride code to one that can be compiled and executed on hardware. A backend is known as a *Platform*. Stride code should start with the line of code shown in Code 3. It instructs the interpreter to load specific platform and hardware descriptor files. The platform descriptor file abstracts hardware resources and contains translation directives while the hardware descriptor file lists the available resources. When versions are not specified the latest descriptor files are loaded. A third file, the hardware configuration file, contains resource configurations. It can be specified after the hardware version in Code 3 using the keyword *with*. The default configuration file is loaded when nothing is specified. The descriptor and configuration files are written in Stride.

```
use PLATFORM version x.x on HARDWARE version x.x
```

Code 3: Loading a platform and a target hardware

The abstraction of hardware resources happens through blocks with reserved labels. These abstractions are common among all platforms. For example, *AudioIn* and *AudioOut* are signal bundles which abstract the analog and digital audio inputs and outputs of hardware. The constant block *AudioRate* abstracts the default sampling rate of these inputs and outputs, while the constant block *AudioDomain* abstracts the default audio callback function. *ControlIn*, *ControlOut*, *ControlRate*, and *ControlDomain* abstract non-audio ADCs, DACs, their default sampling rate, and related default callback function respectively. The range of *AudioIn* and *AudioOut* is [-1.0, 1.0], while that of *ControlIn* and *ControlOut* is [0.0, 1.0]. *DigitalIn* and *DigitalOut* are switch block bundles that abstract digital I/O TTL pins respectively. Communication protocols such as Serial, Open Sound Control [5], MIDI, etc. are also abstracted.

Creating aggregate systems based on multiple hardware platforms is also possible. This is achieved by abstracting the resources of aggregated hardware platforms through a single hardware descriptor file and by abstracting the communication between these platforms by the stream operator and the hardware configuration file.

3.3 Rates and Domains

A signal block is assigned a rate and a domain at declaration. Every domain has a clock with a preset rate derived from the hardware configuration file and abstracted through a constant block as discussed in § 3.2.

When a signal block is running in synchronous mode, it synchronizes itself with the clock of its assigned domain. It samples tokens at its primary input port and generates them at its assigned rate. In this mode, the signal block operates like a sample and hold circuit operating at a preset rate. When running in asynchronous mode, the signal block simply propagates tokens arriving at its primary input port. In both modes, all blocks (or stream expressions containing blocks) with connections to the signal block's primary output port recompute their state when a new token is generated. Computations happen in the domain each block is assigned to.

In Stride, rates and domains propagate through ports. The propagation is upstream. The keywords *streamRate* and *streamDomain* represent the values of the propagated rate and domain respectively where they appear. In Code 4 the values of *streamRate* and *streamDomain* in the *Map* module get their values from the *FrequencyValue* signal block.

```
ControlIn[1]
>> Map (
  minimum: 55.0
  maximum: 880.0
)
>> FrequencyValue;

Oscillator (
  type:      'Sine'
  frequency: FrequencyValue
)
>> AudioOut;
```

Code 4: A control input controlling the frequency of a sine oscillator

Since *FrequencyValue*, in Code 4, was not explicitly declared, it is treated as a signal block with default property values by the interpreter, as shown in Code 1. Therefore, in the *Map* module block the values of *streamRate* and *streamDomain* are *AudioRate* and *AudioDomain* respectively.

The *Oscillator* module in Code 4 encapsulates four signal blocks: *FreqValue*, *PhaseInc*, *Phase*, and *Output*. They represent the frequency, phase increment, phase, and output of the oscillator respectively. In the module's declaration, the rate of the first two signals is set to *none* and both are configured to receive their domain assignment from the block connected to the *frequency* property. The rate of the

Phase signal is set to *none* and is configured to receive its domain from the primary output port of the module, while the *Output* signal is configured to receive both its rate and domain from that port. This is summarized in Table 1.

Label	Rate	Domain
`FreqValue`	`none`	from 'frequency'
`PhaseInc`	`none`	from 'frequency'
`Phase`	`none`	`streamDomain`
`Output`	`streamRate`	`streamDomain`

Table 1: Labels, rates, and domains of signal blocks encapsulated in the *Oscillator* module

Unlike other DSLs, where unit generators represent a single computation unit, Stride can separate and distribute the constituent computations of its modules, such as *Oscillator*, to achieve extremely efficient and highly optimized target code.

To demonstrate the fine control Stride gives its user over generated code, consider a hypothetical platform which generates code [1] like the one shown in Code 5 based on Code 4. The hypothetical platform defines two domains: *AudioDomain* and *ControlDomain*. They are associated with the *audioTick* and *controlCallback* functions in the generated code respectively.

```
AtomicFloat ControlValue = 0.0;

void controlCallback (float *input, int size){
  ControlValue = input[0];
}

void audioTick (float &output){
  static float Phase, FreqValue, PhaseInc = 0.0;

  FreqValue = map(ControlValue, 55., 880.);
  PhaseInc = 2 * M_PI * FreqValue / AudioRate;

  output = sin(Phase);
  Phase += PhaseInc;
}
```

Code 5: Computations performed in the audio tick function on every call

The generated code is not efficient since *FreqValue* and *PhaseInc* are repeatedly computed for every audio sample. By explicitly declaring *FrequencyValue* as signal block and assigning it a slower rate, as shown in Code 6, the efficiency improves as shown in Code 7, where only changes to Code 5 are shown. This is equivalent to control rate processing in Csound and SuperCollider.

```
signal FrequencyValue { rate: 1024. }
```

Code 6: The rate of *FrequencyValue* set to 1024 Hz

```
Accumulator compute(1024. / AudioRate);

void audioTick (float &output){
  ...
  if (compute()){
    FreqValue = map(ControlValue, 55., 880.);
    PhaseInc = 2 * M_PI * FreqValue / AudioRate;
  }
  ...
}
```

Code 7: Accumulator added to reduce computation

The amount of computation can be further reduced by setting the *rate* of *FrequencyValue* to *none* and adding the *OnChange* module as shown in Code 8. Some of the computation will now happen asynchronously and in a reactive fashion. That is, only when the value of *ControlIn[1]* changes some of the computation will be performed as shown in Code 9.

```
signal FrequencyValue { rate: none }

ControlIn[1]
>> OnChange ()
>> Map ( minimum: 55. maximum: 880. )
>> FrequencyValue;
```

Code 8: Enabling asynchronous computation

```
void audioTick (float &output){
 ...
 static float PreviousValue = 0.0;
 ...
  if (ControlValue != PreviousValue){
    FreqValue = map(ControlValue, 55., 880.);
    PhaseInc = 2 * M_PI * FreqValue / AudioRate;
    PreviousValue = ControlValue;
  }
  ...
}
```

Code 9: Some computation performed only with value change

By changing the domain of *FrequencyValue*, shown in Code 10, the computations related to *FreqValue* and *PhaseInc* are performed in a reactive fashion in the control callback, as shown in Code 11. This change results in a highly efficient audio tick function.

```
signal FrequencyValue {
  rate:   none
  domain: ControlDomain
}
```

Code 10: Domain of *FrequencyValue* set to *ControlDomain*

```
AtomicFloat PhaseInc = 0.0;

void controlCallback (float *input, int size){
  static float FreqValue, PreviousValue = 0.0;

  if (input[0] != PreviousValue){
    FreqValue = map(input[0], 55., 880.);
    PhaseInc = 2 * M_PI * FreqValue/ AudioRate;
    PreviousValue = input[0];
  }
}

void audioTick (float &output){
  static float Phase = 0.0;

  output = sin(Phase);
  Phase += PhaseInc;
}
```

Code 11: Highly efficient audio tick function

Generating highly efficient subroutines is crucial to optimize performance on some embedded devices, particularly ones that support instruction caching and equipped with a tightly-coupled instruction memory.

[1] The C code shown in Code 5, 7, 9, and 11 is for demonstration purposes only. The code has not been generated by a backend implementation and is not complete.

3.4 Flow Control

Since control flow is not one of Stride's syntactical constructs, it can be realized in two ways. The first is through switching, achieved by bundling stream expressions followed by indexing the aggregate interface. The second, through triggering reaction blocks, which loop through the stream expressions they encapsulate until they are terminated.

4. CODE EXAMPLES

In the following subsections we present a few examples to demonstrate some of the features and capabilities of Stride.

4.1 Multichannel Processing

In Code 12 the levels of the first two signal blocks of the *Input* block bundle are changed by two signal blocks *A* and *B*. They are then mixed down to a single signal connected to the input ports of the first two signal blocks of the *Output* block bundle, as depicted in Figure 1. All signal blocks are declared with default values.

```
signal Input [4] {}
signal Output [4] {}
signal A {}
signal B {}

Input[1:2]
>> Level ( gain: [ A, B ] )
>> Mix ()
>> Output[1:2];
```

Code 12: Selective multichannel level adjustment and signal mixing

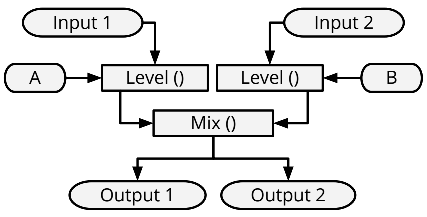

Figure 1: Selective multichannel processing

4.2 Generators, Envelopes, Controls, and a Sequencer

In Code 13 two sine oscillator module blocks, one oscillating at a perfect fifth of the other, are connected to two envelope generator module blocks. The *reset* ports of the oscillators and envelope generators are connected to the trigger block *Trigger*. This is depicted in Figure 2. When triggered, the oscillators' phase gets reset to zero (the default value) while the envelope generators restart. *Trigger* is activated on the rising edge of *DigitalIn[1]*.

```
constant Frequency { value: 440. }
trigger Trigger {}

DigitalIn[1] >> Trigger ;

Oscillator (
  type:       'Sine'
  frequency:  [ 1.0, 1.5] * Frequency
  amplitude:  [ 0.66, 0.33]
  reset:      Trigger
}
>> AD (
  attackTime: [ 0.6 , 0.8 ]
  decayTime:  [ 1.4 , 1.2 ]
  reset:      Trigger
)
>> Mix ()
>> AudioOut[1:2];
```

Code 13: Two sine oscillators connected to two attack / decay modules

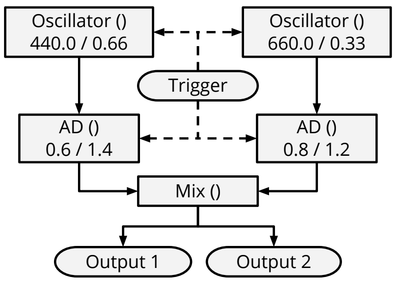

Figure 2: Sine oscillators and attack / decay modules with reset control

Code 14 extends Code 13 after modifying the block type of *Frequency*. The extension enables the control of the oscillators' frequencies through *ControlIn[1]*. When the value of *ControlIn[1]* changes, it gets mapped exponentially and smoothed at a rate 20 times less than the *AudioRate*.

```
signal Frequency { rate: AudioRate / 20. }

ControlIn[1]
>> OnChange()
>> Map ( mode: 'Exponential' minimum: 110.
maximum: 880. )
>> Smooth ( factor: 0.05 )
>> Frequency;
```

Code 14: Controlling the frequencies of the oscillators

Code 13 can also be extended by Code 15 after changing the block type of *Frequency* and reconnecting *Trigger*. The *ImpulseTrain* module block generates a trigger that triggers the *Sequencer* reaction block, whose values are imported from a Stride file called *Notes* into the *note* namespace. The file contains constant block declarations of musical notes.

```
import Notes as note

signal Frequency { default: note.C4 rate: none }

ImpulseTrain ( frequency: 0.5 )
>> ImpulseTrainValue
>> Compare ( value: 0 operator: 'Greater' )
>> Trigger
>> Sequencer (
  values: [ note.C4, note.E4, note.G4, note.C5 ]
  size:    4
  mode:    'Random'
)
>> Frequency;
```

Code 15: Control and triggering through an impulse train and a sequencer

4.3 Feedback

Code 16 is a feedback loop with 32 samples fixed delay as depicted in Figure 3. *Input* and *Feedback* signal blocks are bundled together before being connected to *Level* module blocks. The mixed output is then delayed by 32 samples and streamed into *Feedback*.

```
[ Input, Feedback ]
>> Level ( gain: [ 0.50, -0.45 ] )
>> Mix ()
>> Output
>> FixedDelay ( samples: 32 )
>> Feedback;
```

Code 16: Feedback with 32 samples delay

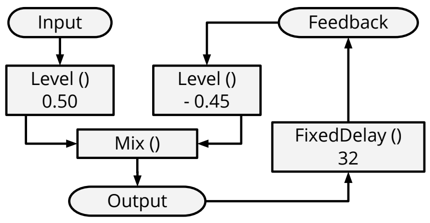

Figure 3: Feedback with 32 samples delay

4.4 Frequency Modulation Synthesis

Code 17 is a single oscillator feedback FM. The output of the oscillator controls its own frequency after being multiplied by a modulation index and offset by a base frequency. The index, base frequency, and amplitude are controlled by control inputs

```
signal Index     { rate: none }
signal Frequency { rate: none }
signal Amplitude { rate: none }

ControlIn[1:3]
>> OnChange ()
>> Map (
  mode:    [ 'Linear', 'Exponential', 'Linear' ]
  minimum: [ 0.08, 40.0 , 0.0 ]
  maximum: [ 2.00, 200.0, 1.0 ]
>> [ Index, Frequency, Amplitude ];

Oscillator (
  type:       'Sine'
  frequency:  Index * Output + Frequency
  amplitude:  Amplitude
)
>> Output;
```

Code 17: Single Oscillator Feedback Frequency Modulation

4.5 Fast Fourier Transform

Code 18 is a smoothed pitch tracker driving a sinusoidal oscillator. FFT is performed on a bundle and the magnitude of the spectrum is computed, followed by finding the index of the first maximum and converting it to a frequency value. These computations are performed at *AudioRate / Size* set by *PeakFrequency*. The computed frequency value is then smoothed at a faster rate to control the frequency of the oscillator running at *AudioRate*. *streamRate* represents the value of the rate of the output port of the *Level* module block, which is *AudioRate / Size*.

```
constant Size { value: 1024. }
signal InputBundle [Size] { rate: none }
signal PeakFrequency { rate: AudioRate / Size }
signal SmoothFrequency { rate: AudioRate / 20. }

InputBundle
>> RealFFT ()
>> ComplexMagnitude ()
>> FindPort ( at: 'Maximum' mode: 'First' )
>> Level ( gain:  streamRate / 2 )
>> PeakFrequency
>> Smooth ( factor: 0.01 )
>> SmoothFrequency;

AudioIn[1]
>> FillBundle ( size: Size )
>> InputBundle;

Oscillator ( frequency: SmoothFrequency )
>> AudioOut[1:2];
```

Code 18: FFT Peak Tracking

4.6 Multirate Signal Processing

In Code 19 a baseband signal with 8 KHz bandwidth sampled at 48 KHz is decimated by a factor of 4 before further processing is performed on it to reduce the number of computations. The signal is then interpolated back to the original sampling rate. The *DSP* module block is a placeholder for a chain of signal processing module blocks.

```
signal Input           { rate: 48000 }
signal ProcessedSignal { rate: 12000 }
signal Output          { rate: 48000 }

Input
>> Decimation (
  type:        'PolyphaseFIR'
  baseband:    8000
  attenuation: 60
  factor:      4
)
>> DSP ()
>> ProcessedSignal
>> Interpolation (
  type:        'PolyphaseFIR'
  bandwidth:   8000
  attenuation: 60
  factor:      4
)
>> Output;
```

Code 19: Multirate processing by decimation and interpolation

4.7 Granular Synthesis

In Code 20 grains are formed using sine oscillators and Gaussian envelopes. The oscillators and their corresponding envelopes are triggered by the *GrainState* switch block bundle through the *SetPort* module block, which acts as a demultiplexer. The *index* of the *SetPort* module is controlled by the *Counter* module block, which increments at the *GrainTriggerRate* value and rolls over when it reaches the *NumberOfGrains* value. The state of a grain is reset after its envelope has generated the required number of samples, computed from the *GrainDuration* value. The output of the envelopes are then mixed and sent to the audio output after adjusting the level.

```
constant NumberOfGrains   { value: 50 }
constant GrainTriggerRate { value: 15 }
constant GrainDuration    { value: 0.005 }
constant GrainFrequency   { value: 220 }
```

```

signal GrainIndex {
  default: 0
  rate:    GrainTriggerRate
}

trigger ResetGrainState [NumberOfGrains] {}

switch GrainState [NumberOfGrains] {
  default: off
  reset:   ResetGrainState
}

Counter (
  startValue: 1
  rollValue:  NumberOfGrains
  increment:  1
)
>> GrainIndex;

on
>> SetPort (
  index: GrainIndex
)
>> GrainState;

Oscillator (
  type:      'Sine'
  frequency: GrainFrequency
  reset:     GrainState
)
>> Envelope (
  type:     'Gaussian'
  size:     GrainDuration * streamRate
  start:    GrainState
  complete: ResetGrainState
)
>> Mix ()
>> Level ( gain: 1.0 / NumberOfGrains )
>> AudioOut[1:2];
```

Code 20: Synchronous triggering of statically allocated grains

Advanced granular synthesizers can be designed in Stride by allocating grains dynamically using the *variable* block to manage the size of core block bundles and triggering them using *reaction* blocks.

5. CONCLUSIONS

With its many features, Stride is an ideal language for creating and deploying new musical instruments on embedded electronic platforms. With few syntactic constructs, it is easy to learn, while its readability and intuitive coding flow make it an attractive choice for beginners and experienced users alike.

Stride documentation is available at:

`http://docs.stride.audio`

Acknowledgments

This work was funded in part by a graduate fellowship by the Robert W. Deutsch Foundation through the AlloSphere Research Group.

6. REFERENCES

[1] J. McCartney, "SuperCollider: a new real time synthesis language," in *Proceedings of the 1996 International Computer Music Conference*, Hong Kong, 1996.

[2] G. Wang and P. R. Cook, "ChucK: A Concurrent, On-the-fly, Audio Programming Language," in *Proceedings of the 2003 International Computer Music Conference*, Singapore, 2003.

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

[4] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010.

[5] M. Wright and A. Freed, "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers," in *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, 1997.