Stride for Interactive Musical Instrument Design

Joseph Tilbian
Media Arts and Technology Program
University of California Santa Barbara
Santa Barbara, California
jtilbian@mat.ucsb.edu

Andrés Cabrera
Media Arts and Technology Program
University of California Santa Barbara
Santa Barbara, California
andres@mat.ucsb.edu

ABSTRACT

Stride is a language tailored for designing new digital musical instruments and interfaces. Stride enables designers to fine tune the sound and the interactivity of the instruments they wish to create. Stride code provides a high-level description of processes in a platform agnostic manner. The syntax used to define these processes can also be used to define low-level signal processing algorithms.

Unlike other domain-specific languages for sound synthesis and audio processing, Stride can generate optimized code that can run on any supported hardware platform. The generated code can be compiled to run on a full featured operating system or bare metal on embedded devices. Stride goes further and enables a designer to consolidate various supported hardware and software platforms, define the communication between them, and target them as a single heterogeneous system.

Author Keywords

Stride, Domain-Specific Language, Declarative, Reactive, Interaction Design, Code Generation, Sound Synthesis, Digital Signal Processing

ACM Classification

D.3.2 [Programming Languages] Language Classifications—Very high-level languages, D.3.3 [Programming Languages] Language Constructs and Features—Frameworks, D.3.4 [Programming Languages] Processors—Code generation

1. INTRODUCTION

The development of new electronic musical instruments often requires a mix of software and hardware. Stride[6], a domain-specific programming language for real-time sound synthesis, processing, and interaction design, abstracts hardware and software architectures, simplifying the process of software/hardware integration while giving the user control over the code generation process. These abstractions are defined in Stride systems which represent the inner workings of the target hardware and software, exposing them in a simple and consistent manner across platforms.

Stride enables its user to declare the frequency at which Stride expressions are evaluated and provide the user with the ability to control and fine tune the quality of the sounds they seek. Stride also enables its user to control where expressions get evaluated and computed. This type of control is essential to optimizing code running on a resource constrained system such as a microcontroller. A user of Stride can also design interaction using Stride *Reaction*, an abstraction to handle asynchronous events.

Because of Stride's ability to abstract hardware, heterogeneous systems can be defined and consolidated under a single Stride system. This is achieved by abstracting the communication between the hardware and software platforms encompassing the heterogeneous system. In other words, different pieces of hardware (e.g. Arduino, Raspberry Pi, Desktop, etc.) can be grouped together to appear within Stride as a single system, as the communication between the devices is handled internally by Stride according to the system definition.

The Stride language is part of the Stride environment which also encompasses the Stride integrated development environment (Stride IDE), an intermediate code generator, and a target code generator.

2. STRIDE

In the following sections we present the Stride language, the Stride environment, and Stride systems.

2.1 The Stride Language

Although Stride is a textual language inheriting concepts from Unit Generator languages like Csound[1], SuperCollider[2] and ChucK[7], its basic construct is the streaming operator ≫ which makes it conceptually similar to dataflow languages like Pure Data[5] and Max[1] (see Code 1). Stride is not a dynamic unit generator graph manager, but rather a code generator like Faust[4]. Additionally, Stride is designed to facilitate both low-level signal processing algorithms and high-level constructs, like granular synthesis and frequency domain processing, using the same syntax.

```
Oscillator(frequency: 440) >> Level(gain:
0.2) >> AudioOut;
```

Code 1: Basic Stride code showing the stream operator

Stride is designed around the declarative and dataflow paradigms. The language only has two constructs: block declarations and stream expressions. Stride allows both push (reactive) and pull programming, achieved by controlling the rate of the *signal* block, which is the fundamental building block of the language. Stride borrows some of the best features of other programming languages like multichannel expansion, single operator interfacing, multiple control rates, and per sample processing. Stride is also a self documenting language.

[1] `https://cycling74.com/products/max`

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

2.2 The Stride Environment

The Stride language has been implemented as part of a full Stride environment that includes an integrated development environment (IDE), a parser and a set of platform files that allow building and deploying on specific hardware platforms. When Stride code is compiled, an intermediary representation of code in the form of an abstract syntax tree (AST) is processed and passed to a set of code generator scripts. The resulting code is finally compiled (or cross-compiled) and deployed to hardware.

Stride is designed with embedded hardware in mind, so it can be an alternative to software tools targeting audio processing and interaction on embedded devices like Bela[3], Axoloti[2], and the OWL[8]. The main advantage of Stride is that it is platform agnostic, and can be easily ported to work on these devices. Because Stride is a programming language, it is not restricted by a fixed number of building blocks or objects and can be used to perform low-level functions with native speed.

2.3 Stride Systems

A *System* in Stride is a set of domains with their associated software and hardware platforms. For example, a simple embedded audio development board based on a microcontroller with a network peripheral can be presented to Stride as three domains: the Audio domain, the Computation domain and the Network domain. Signals and processing will be assigned to one of these domains and the code will be run appropriately. The Audio domain, a *synchronous* domain, has a fixed rate set by the digital to analog converter (DAC) clock and will constantly compute as the DAC requires samples. The Network domain is *asynchronous* and has no rate, therefore it reacts and processes data only when a message from the network is received. The Computation domain is an *immediate* domain, that is, computation will happen as quickly as possible within the domain, but depending on the complexity and the amount of computation required the duration will vary. The Computation domain can be used to compute changes in parameters resulting from messages received through the Network domain. The computed parameters can then be asynchronously passed to the Audio Domain when ready. All these domains belong to the same platform, so the code will be generated for the board's microcontroller and deployed accordingly. The system also defines the relationships between these domains and their corresponding processing priority on the hardware. In this scenario, the Audio domain will have the highest priority, followed by the Network domain and then the Computation domain.

The Stride system exposes the inner workings of a target computer and its peripherals to the user in an abstracted form. Stride does not only abstract the hardware but also the software architecture used to organize various processes. For example, the relationship between an audio callback and a network listener thread/interrupt. These abstractions grant the user full control of the underlying system without having to know the implementation details.

A noteworthy advantage of a Stride system is in its ability to expose hardware interrupts. An Interrupt domain offered by a such a system can be associated with digital input pins configured by the user to operate in interrupt mode. This allows the user to design a highly reactive system which responds to external change immediately. This is in contrast to polling a digital pin at a quasi synchronous rate.

A Stride system also enables the user to synchronize inputs and outputs from various peripherals to achieve the desired response. Values generated by an analog to digital converter (ADC) running at a predefined rate connected to a sensor can be read in the Audio domain and allow for true sample-accurate synchronization with an external event.

[2] http://www.axoloti.com/

3. MUSICAL INSTRUMENT DESIGN ON EMBEDDED HARDWARE

The design of a new musical instrument with a physical interface and whose sound is generated by an embedded computer requires multidisciplinary knowledge and deep understanding of complex hardware systems to achieve the desired means of interaction, acoustic quality, and the required response.

Stride was designed with these requirements in mind, to allow the designer to quickly model their instrument and fine tune it to achieve the artistic results they seek without having to delve deep into the complex world of programming embedded systems.

3.1 Rates and Domains

Typically, for a digital musical instrument, interactions by the player are captured through sensors. Sensors are either sampled periodically (e.g. potentiometer) or monitored for asynchronous events (e.g. button). The instrument might also be designed to receive or send messages over a physical communication layer (e.g. MIDI, OSC[9], etc.). Periodically sampled data are processed and assigned to control various parameters of the synthesized sounds by an embedded computer. Asynchronous events captured by sensors are used to start, stop, or re-trigger various elements of the sound generation process, while messages are used to control parameters, trigger events, or do both simultaneously.

Stride abstracts the hardware input and output peripherals and enables the designer to configure their properties. Stride also allows the designer to choose how often to process periodic signals by setting the rate of these signals and choose where to perform computations effected by these changes through domain assignments. Code 2 demonstrates how a sensor connected to control input 1 of a target hardware platform is used to adjust the frequency of a sine oscillator and how an event on digital input 1 is used to reset the oscillator. The output of the oscillator is routed to audio outputs 1 and 2 of the target hardware platform.

```
signal FrequencyValue {
  default: 440.0
  rate:    2048
  domain:  ControlDomain
}

signal OscillatorOutput {
  default: 0.0
  rate:    AudioRate
  domain:  AudioDomain
 }

trigger Reset {}

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

DigitalIn[1] >> Reset;

Oscillator (
  type:      'Sine'
  frequency: FrequencyValue
  reset:     Reset
)
```

```
>> OscillatorOutput;

OscillatorOutput >> AudioOut[1:2];
```

Code 2: Controlling the frequency and resetting the phase of an oscillator

In Code 2 two signal blocks and a trigger block are declared followed by four stream expressions. The two signals are called *FrequencyValue* and *OscillatorOutput*. FrequencyValue has a rate of 2048Hz and gets computed in the *ControlDomain*. OscillatorOutput[3] runs at *AudioRate*, a constant, which represents the default audio sampling rate of the *AudioDomain* domain where the signal is computed. The declared trigger is called *Reset*. In the following stream expressions, blocks are connected to each other to build a processing graph using the stream operator $\gg$. *Oscillator* is a module block. It encapsulates blocks and stream expressions which in turn define the module's processing graph. Properties of external blocks, such as rates and domains, become accessible to the module when these blocks are connected to it. In turn the module uses this information to declare the rates and domains of signals it encapsulates. Since FrequencyValue is computed in the ControlDomain, all blocks inside the Oscillator module associated with the *frequency* property of the module also get computed in the ControlDomain. The phase increment of the Oscillator is one of those values. The phase increment is assigned to the ControlDomain and is set to compute at 2048Hz. The rate at which the FrequencyValue and phase increment are updated are not tied to the rate of the ADC. When control input 1 is connected to FrequencyValue a rate change takes place. This change is defined by the system being targeted. The phase of the Oscillator gets its rate and domain assignment from the block connected to the output of the Oscillator, which is OscillatorOutput and is assigned to AudioDomain and computes at AudioRate. The domain separation promotes optimization of the audio callback function represented by the AudioDomain domain. Because of these particular domain assignments to FrequencyValue and OscillatorOutput the phase is computed per audio sample while the phase increment is only computed 2048 times per second outside the audio callback and passed to it asynchronously. The rate at which the phase increment is computed can have a significant effect on the output quality of the generated sound. If it is set too low, it will result in a 'zipper' effect and setting it high will result in excessive computations. The designer has the choice to balance these rates to achieve the desired sound quality and response they seek.

The *Control Rate* has been an important feature of MusicN languages (and successors like Pure Data) that allows running computation at a lower rate to save computation cycles. In these systems, the control rate is considered a different "type" that can not be mixed with regular audio signals, and unit generators must be programmed to handle them specifically, since control rate signals are typically scalar values while audio signals are vectors. In Stride each signal has its own rate at which it is processed, and when it is connected to a different rate, automatic up or down-sampling occurs[4].

[3]The declaration of the OscillatorOutput signal is not required. It is added to clarify the oscillator is being run at audio rate in the audio domain.

[4]The specifics of this default sample rate conversion are undefined and determined by the system. It is typically the simplest strategy, and more sophisticated forms of conversion are offered as specific functions.

3.2 Reactions

Asynchronous events are handled in Stride through an abstraction called *Reaction*. Reactions are invoked through triggers which are activated when asynchronous events are detected. Reactions can be configured to self terminate and re-arm when certain criteria are met or terminated by other asynchronous events. Code 3 demonstrates a simple reaction to start and stop a stream when changes are detected on digital inputs 1 and 2 of a target hardware platform. The stream connects the audio inputs on the target hardware platform to the audio outputs.

```
trigger Stop {}

reaction Stream {
  terminate: Stop
  streams:   AudioIn >> AudioOut;
}

DigitalIn[1] >> Stream();
DigitalIn[2] >> Stop;
```

Code 3: A reaction to start and stop a stream

In Code 3 a trigger called *Stop* is declared followed by a *reaction* block declaration called *Stream*. In the following two stream expressions digital input 1 of the target hardware platform is connected to the Stream reaction while digital input 2 is connected to the Stop trigger. A rising edge (default behavior) detected on digital pins 1 and 2 triggers the Stream reaction and activates the Stop trigger respectively.

4. HETEROGENEOUS SYSTEMS

In Stride, a heterogeneous system can be built from physically separate hardware platforms. This feature enables consolidating multiple supported hardware and software platforms under one system. The system can then be targeted as a single unit. To create a heterogeneous system in Stride, the communication between the separate hardware platforms needs to be defined and abstracted.

Imagine a scenario where a digital musical instrument requires a high number of ADC inputs to capture information from various sensors and has a computation intensive audio synthesis graph that requires a dedicated processor. If Stride supports an embedded development board called *ADC Board* with a high number of ADC inputs with an I^2C interface and another board called *DSP Board* capable of performing extensive computations with an I^2C interface and two audio output channels, they can be consolidated to form a new system. By abstracting the communication between the two boards in Stride over the I^2C interface the user only needs to make the physical connection between the two boards. Stride will handle generating the code to enable the communication between the boards and call the appropriate toolchain and programmers to deploy the resulting firmware on each board.

```
signal FrequencyValues [12] {
  default: 440.0
  rate:    ADC_Board::ControlRate
  domain:  ADC_Board::ControlDomain
}

signal OscillatorOutput {
  default: 0.0
  rate:    DSP_Board::AudioRate
  domain:  DSP_Board::AudioDomain
 }

ADC_Board::ControlIn[1:12]
>> Map (
```

```
  minimum: 20.0
  maximum: 20000.0
)
>> FrequencyValues;

Oscillator (
  type:      'Sine'
  frequency: FrequencyValues
)
>> Mix()
>> OscillatorOutput
>> DSP_Board::AudioOut[1:2];
```

Code 4: Additive synthesis on a heterogeneous system

In Code 4 a signal block bundle called *FrequencyValues* is first declared. The signal bundle has 12 channels and is assigned a rate and a domain defined in the ADC Board platform. The declaration is followed by a second signal block declaration called *OscillatorOutput* which is assigned a rate and a domain defined by the DSP Board platform. In the following stream expression the ADC inputs are read and mapped on the ADC Board and assigned to FrequencyValues. The interaction between the two platforms occurs in the following stream expression, where the output of 12 oscillators (multi-channel expansion because of the 12 channles in FrequencyValues) are mixed and streamed to OscillatorOutput. Since the phase increment of the Oscillator module is computed in the domain of the block connected to the frequency property, in this example the phase increment gets commuted on the ADC Board and is asynchronously communicated to the DSP Board where the phase of the oscillators is being computed.

If the phase increments of the oscillators need to be computed on the DSP Board rather than the ADC Board, the declaration of a second signal block bundle and the assignment of its domain to one available on the DSP Board platform achieves that. Code 5 shows the incremental changes made to Code 4 to achieve this result.

```
...

signal ReceivedValues [12] {
  default: 440.0
  rate:    DSP_Board::ControlRate
  domain:  DSP_Board::ControlDomain
}

...

FrequencyValues >> ReceivedValues;

...

Oscillator (
  type:      'Sine'
  frequency: ReceivedValues
)
>> Mix()
>> OscillatorOutput
>> DSP_Board::AudioOut[1:2];
```

Code 5: Switching domains on a heterogeneous system

In Code 5, in the declaration of the *ReceivedValues* signal block bundle, its rate and domain are assigned to *ControlRate* rate and *ControlDomain* of the DSP Board platform. In the following stream expression the information is asynchronously exchanged between the two boards over the I^2C interface from *FrequencyValues* to *ReceivedValues*. The phase increment of the oscillators is now computed in the ControlDomain of the DSP Board platform and is asynchronously passed to the AudioDomain of the DSP Board platform asychronously where the phase is being computed at AudioRate.

Stride also allows creating synchronous signal groups. If in Code 5 we wished to updated the phase increment of the 12 oscillators synchronously, the *ReceivedValues* signal block bundle can be assigned to a synchronous signal group.

5. CONCLUSIONS

In this paper we have presented Stride, a language tailored for designing new digital musical instruments and interfaces. With its capabilities to target embedded systems by generating optimized code for each target, it makes it an ideal choice for designers to design and fine tune the sound and the interactivity of the instruments they seek to create.

Stride's capability to consolidate various hardware platforms into a single platform offers a unique advantage to designers, enabling them to focus on the instrument design rather than focusing on maintain code and tools for various platforms.

6. ACKNOWLEDGMENTS

The design and development of Stride and the Stride IDE were funded in part by a graduate fellowship by the Robert W. Deutsch Foundation through the AlloSphere Research Group at UCSB and a grant by the UCSB Center for Research in Electronic Art Technology.

7. REFERENCES

[1] R. Boulanger, editor. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, 2000.

[2] J. McCartney. Supercollider: a new real time synthesis language. In *Proceedings of the 1996 International Computer Music Conference*, Hong Kong, 1996.

[3] G. Moro, A. Bin, R. H. Jack, C. Heinrichs, and A. P. McPherson. Making high-performance embedded instruments with bela and pure data. In *Proceedings of the 2016 International Conference on Live Interfaces*, Brighton, 2016.

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

[5] M. S. Puckette. Pure data. In *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, 1997.

[6] J. Tilbian and A. Cabrera. Stride: A declarative and reactive language for sound synthesis and beyond. In *Proceedings of the 2016 International Computer Music Conference*, Utrecht, 2016.

[7] 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.

[8] T. Webster, G. LeNost, and M. Klang. The owl programmable stage effects pedal: Revising the concept of the on-stage computer for live music performance. In *Proceedings of the 2014 International Conference on New Interfaces for Musical Expression*, London, 2014.

[9] 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.