DESIGNING AND IMPLEMENTING THE CHUCK PROGRAMMING LANGUAGE

Ge Wang *Perry R. Cook*[†] *Ananya Misra*
Princeton University
Department of Computer Science ([†]also Music)

ABSTRACT

ChucK re-factors the idea of a computer music language into three orthogonal basis components: unit generator connections that are data-flow only, globally consistent "first-class" time control, and sample-synchronous concurrency. The syntax, semantic, and usage have been discussed in previous works. The focus and contributions of this paper are (1) to examine the philosophies and decisions in the language design (2) to describe ChucK's implementation and runtime model, and (3) to outline potential applications enabled by this framework. We present an experiment in designing a computer music language "from scratch" and show how things work. We hope these ideas may provides an interesting reference for future computer music systems.

1. INTRODUCTION

"The old computing is about what computers can do,
the new computing is about what people can do."
- Ben Shneiderman, HCI Researcher.

If Human Computer Interaction (HCI) research strives to give people greater and better access to using the computer, then perhaps computer music language design aims to give programmers more natural representation of audio and musical concepts. Machines have advanced to a point where system design no longer depends on blazing performance, but can focus foremost on flexibility and how users can interact with the system.

On today's machines, ChucK [10] is a real-time audio programming language that offers potentially worse performance than languages such as SuperCollider [4], Nyquist [2], Max/MSP [6], Pure Data [7], CSound [9], and other systems [5, 3]. But it still runs comfortably in real-time for many tasks and, more importantly, offers something unique: a fundamental and flexible programming model to manipulate time and parallelism.

ChucK is strongly-timed, concurrent, and embodies a *do-it-yourself* spirit. It combines the conciseness of high-level computer music languages with the transparency and programmability of low-level languages. It is readable even when programs become complex. This paper describes an experiment in designing a computer music language from scratch, and discusses its implementation, outline applications made possible by this model.

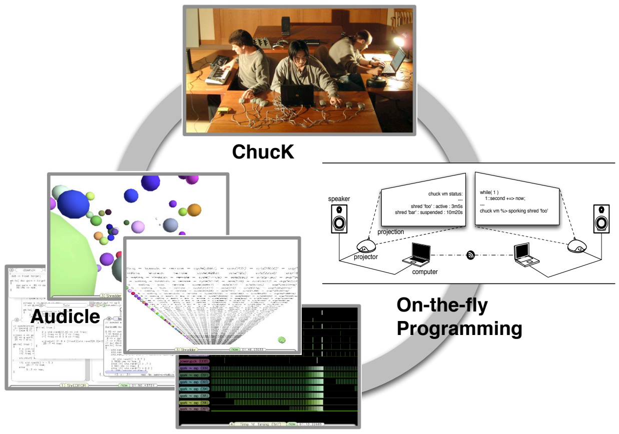


Figure 1. A short history of ChucK (2003), on-the-fly programming (2004), and the Audicle (2004).

2. LANGUAGE DESIGN GOALS

ChucK continues to be an open-source research experiment in designing a computer music language from the "ground up". While it draws ideas from many sources (C, Java, Max, SuperCollider), it isn't based on a single existing language; hence we were free (and doomed) to make language design decisions at nearly every stage. A main focus of the design was the precise programmability of time and concurrency, in a readable way. System throughput remains an important consideration, especially for real-time audio, but was not our top priority. We designed the language to provide maximal control for the programmer, and tailored the system performance around the design. Design goals are as follows.

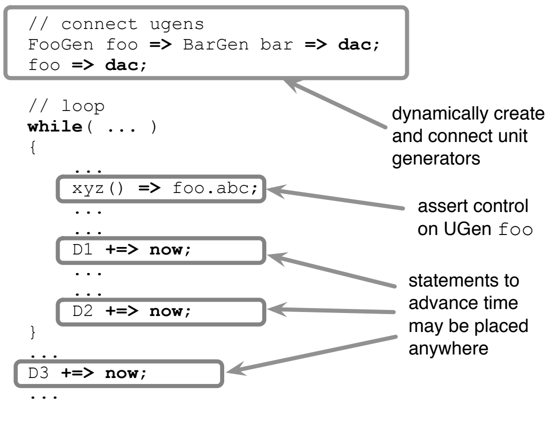
- **Flexibility**: allow the programmer to naturally specify both high and low level operations in time.
- **Concurrency**: allow the programmer to write parallel modules that share both data and time, and that can be precisely synchronized.
- **Readability**: provide/maintain a strong correspondence between code structure and timing.
- **A do-it-yourself language**: combine the expressiveness of lower-level languages and the ease of high-level computer music languages. To support high-level musical concepts, precise low-level timing, and the creation of "white-box" unit generators, all directly in the language.
- **On-the-fly**: allow programs to be edited as they run. This work is ongoing [11], and is not discusses here.

The solution in ChucK was to make time itself computable (as a first-class citizen of the language), and allowed a program to be ”self-aware” in the sense that it always knows where it is in time, and can control its own progress over time. Furthermore, if many programs can share a central notion of time, then it is possible to synchronize parallel code solely and naturally based on time. Thus arose our concept of a *strongly-timed language*, in which programs have precise control over their own timing. Control data to any unit generator can be sample-synchronously asserted at any time.

This mechanism transfered the primary control over time from inside opaque unit generators to the language, mapping program flow explicitly to time flow. This enabled the programmer / composer to specify arbitrarily complex timing and to ”sculpt” a sound or passage into perfection by operating on it at any temporal granularity.

We are not the first to address this issue of enabling low-level timing in a high-level audio programming language. Chronic [1] and its temporal type constructors was the first attempt we are aware of to make arbitrary sub-control rate timing programmable for sound synthesis. While the mechanisms of Chronic are very different from ChucK's, one aim is the same: to free programmers from having to implement ”black-box”, opaque unit generators (in a lower-level language, such as C/C++) when a new lower-level feature is desired.

However, time alone is not enough – we also need concurrency to expressively capture parallelism. Fortunately, the timing mechanism lent itself directly to a concurrent programming model. Multiple processes (called *shreds*), each advancing time in its own manner, can be synchronized and serialized directly from the timing information.


```
// connect ugens
FooGen foo => BarGen bar => dac;
foo => dac;

// loop
while( ... )
{
    ...
    xyz() => foo.abc;
    ...
    ...
    D1 +=> now;
    ...
    ...
    D2 +=> now;
}
...
D3 +=> now;
...
```
dynamically create and connect unit generators
assert control on UGen foo
statements to advance time may be placed anywhere

Figure 2. Structure for a basic ChucK shred. Any number of shreds can run in parallel.

This yields a programming model (Figure 2) in which concurrent shreds construct and control a global unit generator network over time. A scheduler (or *shreduler*) uses the timing information to serialize the shreds and the audio computation in a globally synchronous manner. It is completely deterministic (real-time input aside) and the synthesized audio is guaranteed to be correct, even when real-time isn't feasible.

3. FROM CONCEPT TO IMPLEMENTATION

ChucK programs are type-checked, emitted into ChucK shreds (processes) containing byte-code, and then interpreted in the virtual machine. A shreduler *shredules* the shreds and serializes the order of execution between various shreds and the audio engine. Under this model, shreds can dynamically connect, disconnect, and share unit generators in a global network. Additionally, shreds can perform computations and change the state of any unit generator at precisely any point in time.

Audio is synthesized from the global unit generator graph a sample at time by ”sucking” samples beginning from well-known ugen ”sinks” - such as **dac**. Time as specified in the shreds is mapped by the system to the audio synthesis stream. When a shred advances time, it is actually scheduling itself to be woken up after some future sample. In this sense, the passage of time is *data-driven*, and guarantees that the timing in the shreds is bound only to the output and not to any other clocks - and that the final synthesis is ”correct” and sample-faithful regardless of whether the system is running in real-time or not.

Additional processes interface with I/O devices (as necessary) and the runtime compiler. A server listens for incoming network messages. Various parts of the VM can optionally collect real-time statistics to be visualized externally in environments such as the Audicle [12].

3.1. Compilation + VM Instructions

Compilation of a ChucK program follows the standard phases of lexical analysis, syntax parsing, type checking, and emission into instructions. ChucK is procedural and strongly-typed. Programs are emitted into ChucK virtual machine instructions, either as part of a new shred, or as globally available objects or routines. The compiler runs in the same process as the virtual machine, and can compile new programs on-demand.

3.2. Shreds and the Shreduler

After compilation, a ChucK shred is passed directly to the virtual machine, where it is shreduled to start execution immediately. Each shred has several components (Figure 4): (1) bytecode instructions emitted from the source code, (2) an operand stack for local and temporary calculations (functionally equivalent to hardware registers), (3) a memory stack to store local variables at various scopes, i.e. across function calls, (4) references to children shreds (shreds spawned by the current shred) and a parent shred, if any, and (5) a shred-local view of **now** - which is a fractional sample away from the system-wide **now** and which enables sub-sample-rate timing.

The state of a shred is completely characterized by the content of its stacks and their respective stack pointers (sp). It is therefore possible to suspend a shred between any two instructions. Under normal circumstances, however, a shred is suspended only after instructions that advance time. Shreds can create and remove other shreds.

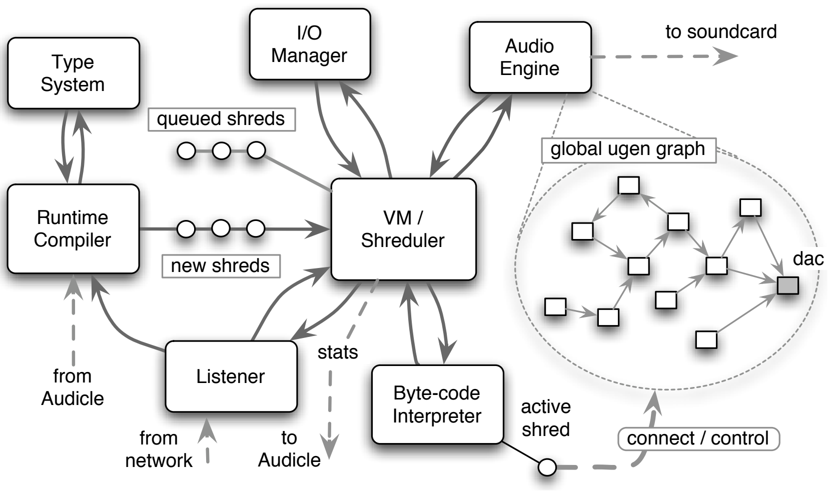


Figure 3. The ChucK Run-time.

The shreduler serializes the execution of the shred with the audio engine, and also maintains the system-wide value of the keyword **`now`**. The unit of **`now`** is mapped to the number of samples in the final synthesis that have elapsed since the beginning of the program.

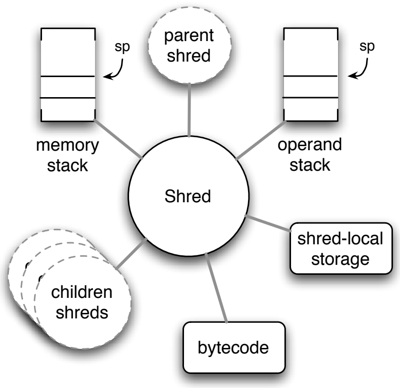


Figure 4. Components of a ChucK shred.

For a single shred, the shreduling algorithm is illustrated in Figure 5. A shred is initially shreduled to execute immediately - further shreduling beyond this point is left to the shred. The shreduler checks to see if the shred is shreduled to wake up at or before the current time (**`now`**). If so, the shred resumes execution in the interpreter until it schedules itself for some future time T. At this point, the shred is suspended and the wake-up time is set to T. Otherwise, if the shred is not scheduled to wake up at **`now`**, then the shreduler calls the audio engine, which traverses the global unit generator graph and computes the next sample. The shreduler then advances the value of **`now`** by the duration of 1 sample (called a **`samp`** in ChucK), and checks the wake-up time again. It continues to operate in this fashion, interleaving shred execution and audio computation in a completely synchronous manner.

It is possible that a shred misbehaves and never advances time or, in the real-time case, performs enough computation to delay audio. The Halting Problem tells us that the VM cannot hope to detect this reliably. However, it is possible for the user to identify this situation and manually remove a shred from the interpreter. Secondly, the above algorithm is geared towards causal, immediate mode operations in which time can only be advanced towards the future. This same model can be extended so shreds can also move backwards in time, which is not discussed here.

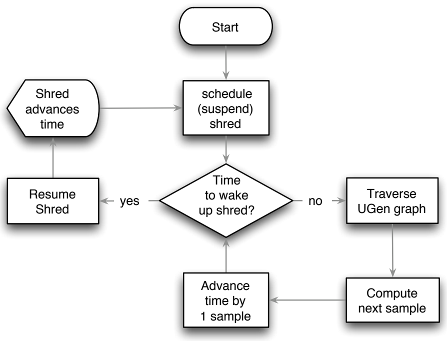


Figure 5. Shreduling a single shred with synthesis.

For multiple shreds, the mechanism behaves in a similar manner, except the shreduler has a waiting list of shreds, sorted by requested wake-up time. Before the system-wide **`now`** is advanced to the next sample, all shreds waiting to run at or before the current time are allowed to run.

3.3. Bytecode Interpreter

ChucK virtual machine bytecode instructions operate on a 4-tuple: (1) the shred operand stack, (2) memory stack, (3) reference to the shred itself, (4) the virtual machine the shred is running on. Each instruction has well-defined behavior relative to the stacks, shred, and VM. Instructions range from arithmetic operations such as ADD-FLOAT and XOR-INT, to more complex ones like ADVANCE-TIME. Because ChucK is strongly-typed, instructions can operate without any run-time type checking (after compilation), thus outperforming dynamically typed interpreters.

3.4. Audio Computation

Unit generators are dynamically created, connected, disconnected, and controlled from shreds. However, the actual audio computation takes place separately. When the shreduler decides that it's appropriate to compute the next sample and advance time, the audio engine is invoked. The global unit generator graph is traversed in depth-first order, starting from one of several well-known sinks, such as **`dac`**. Each unit generator connected to the **`dac`** either directly or indirectly is asked to compute and return the next sample. The system marks visited nodes so that each unit generator is computed exactly once for every sample. The output value of of each ugen is stored and can be recalled, enabling feedback cycles in the graph.

4. APPLICATIONS AND "COOL SIDE EFFECTS"

ChucK's strongly-timed programming model has enabled many interesting applications and side effects. Some are already implemented, others are in the works.

Visualizing the programming process. This framework lends itself to visualization. Relative timing, concurrent processes and their activities can be displayed in real-time by the "context-sensitive" Audicle programming environment (Figure 6). Combined with live coding, the Audicle provides insight into the audio programs we write.

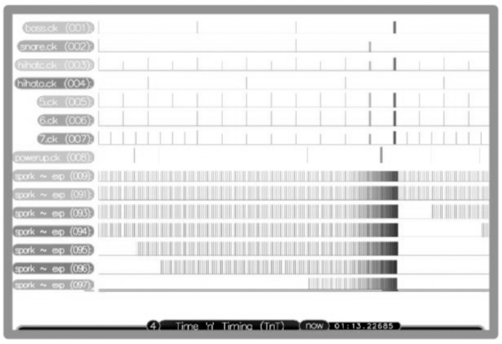


Figure 6. Real-time visualization in the Audicle.

Writing "white-box" unit generators. Since we are able to talk about time in a globally consistent and arbitrarily fine level, it is possible to construct unit generators directly in ChucK. This reduces the need to go outside the language when adding low-level functionality (granular, FOF's, etc.), encouraging new "open-source" UGen's.

Building and testing interactive systems. ChucK's concurrent model makes it easy to develop complex mappings for computer music controllers. The timing mechanism makes it easy to precisely record input sensor data for playback and storage. Further, timings of weeks and years can be programmed naturally.

Precisely specifying synthesis algorithms. Because ChucK has no dependencies on system timing, deterministic audio synthesis is guaranteed. This consistency and precision, combined with explicit readability, allows clear algorithm specification (i.e. for education and research).

Programming audio analysis. This time-based model is applicable to any system that processes audio or other stream-based data, allowing "hopping" through time in a strongly-timed manner. In addition to audio synthesis, it may be beneficial to use the same framework for feature extraction and audio analysis, connecting and controlling modules in systems like MARSYAS [8].

5. CONCLUSION

Recall the opening quote of this paper. We have entered an age in computing where it's no longer mandatory to design systems tailored around how computers can best perform, but around how humans (including programmers) can better interface and control the computer. This is not to say that good performance and optimization are not important in system design. It's just that we are no longer limited by the same "need for speed" because computers themselves have become more powerful. In this sense we are fortunate. Indeed, ChucK is an experiment afforded by these trends. Embedded in this experiment is the hope that sound becomes more *free* – both in terms of the cost in computing it, and how we program it.

`http://chuck.cs.princeton.edu/`

6. REFERENCES

[1] Brandt, E. "Temporal Type Constructors for Computer Music Programming", *Proc. ICMC*, 2000.

[2] Dannenberg, R. B. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis", *CMJ*, 21(3), 1997.

[3] Mathews, M. V. *The Technology of Computer Music*, Cambridge, MA: MIT Press. 1969.

[4] McCartney, J. "SuperCollider: A New Real-time Synthesis Language", *Proc. ICMC*, 1996.

[5] Pope, S. T. "Machine Tongues XV: Three Packages for Software Sound Synthesis", *CMJ*, 17(2), 1993.

[6] Puckette, M. "Combining Event and Signal Processing in the MAX Graphical Programming Environment", *CMJ*, 15(3), 1991.

[7] Puckett, M. Pure Data, *Proc. ICMC*. 1996.

[8] Tzanetakis, G., and P. R. Cook. "MARSYAS: A Framework for Audio Analysis", *Organised Sound 4(3)*, 2000.

[9] Vercoe, B. and D. Ellis. Real-Time CSOUND: Software Synthesis with Sensing and Control, *Proc. ICMC*, 1990.

[10] Wang, G. and P. R. Cook. "ChucK: A Concurrent, On-the-fly Audio Programming Language", *Proc. of the ICMC*, 2003.

[11] Wang, G. and P. R. Cook. "On-the-fly Programming: Using Code as an Expressive Musical Instrument", *Proc. NIME*, 2004.

[12] Wang, G. and P. R. Cook. "The Audicle: a Context-sensitive, On-the-fly Audio Programming Environ/mentality", *Proc. ICMC* 2004.