ChucK: A Programming Language for On-the-fly, Real-time Audio Synthesis and Multimedia

Ge Wang
Department of Computer Science
Princeton University
Princeton, NJ. U.S.A.
gewang@cs.princeton.edu

Perry Cook
Department of Computer Science (also Music)
Princeton University
Princeton, NJ. U.S.A.
prc@cs.princeton.edu

ABSTRACT
In this paper, we describe ChucK – a programming language and programming model for writing precisely timed, concurrent audio synthesis and multimedia programs. Precise concurrent audio programming has been an unsolved (and ill-defined) problem. ChucK provides a concurrent programming model that solves this problem and significantly enhances designing, developing, and reasoning about programs with complex audio timing. ChucK employs a novel *data-driven* timing mechanism and a related *time-based synchronization* model, both implemented in a virtual machine. We show how these features enable precise, concurrent audio programming *and* provide a high degree of programmability in writing real-time audio and multimedia programs. As an extension, programmers can use this model to write code *on-the-fly* – while the program is running. These features provide a powerful programming tool for building and experimenting with complex audio synthesis and multimedia programs.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications – *specialized application languages*. D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent Programming Structures.*

General Terms
Design, Experimentation, Languages.

Keywords
Programming language, audio synthesis, multimedia, concurrency, synchronization, signal processing, real-time, compiler, virtual machine.

1. INTRODUCTION
Time and parallelism are essential notions in audio and multimedia programming. However, providing precise and yet clear programmatic control over time and timing is challenging. Existing low-level languages, such as C/C++/Java, have no inherent notion of time, whereas high-level languages hide timing from the programmer. Additionally, supporting a concurrent programming model for audio synthesis and signal processing can be very useful but has been an unsolved challenge from both the language level and the system level. Currently, no existing language supports precise, concurrent programming of audio. This fundamentally limits the way we write multimedia programs.

ChucK is a *strongly-timed*, concurrent audio programming language[9]. Its language constructs and programming model presents an elegant solution to concurrent audio programming with sample-synchronous precision. This fundamentally enhances our ability to write audio programs with complex timing. ChucK was designed to the meet the following goals.

Representation: provide a clear syntactic and semantic representation that captures important properties of audio programming paradigms.

Timing: support precise, sample-synchronous audio timing with high degree of programmability.

Concurrency: provide the ability to write modular, concurrent audio/multimedia programs with high precision and clarity.

Programmability: give precedence to a high degree of programmatic control over timing, audio synthesis, and synchronization with other media (i.e. real-time graphics).

On-the-fly Programming: use concurrency and precise timing as a framework to explore on-the-fly audio/multimedia programming (Figure 1) – to add/modify/remove parts of the program as it is running, opening new possibilities for runtime audio and media experimentation.

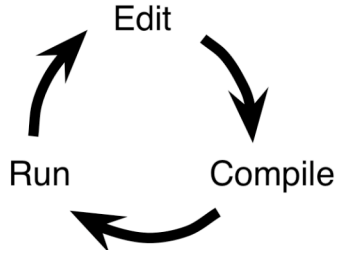


Figure 1. *On-the-fly Programming*. Editing and compiling a program during its runtime.

We will motivate these goals in *Section 2*, and discuss how ChucK addresses each in *Section 3*. We conclude and discuss future directions in *Section 4*.

MM'04, October 10-16, 2004, New York, New York, U.S.A.

2. BACKGROUND

The notion of time is inseparable from audio and multimedia programming – it is fundamentally ingrained into sound, music, and our perception of audio and visuals. For audio, timing issues exist at three fundamental levels. At the lowest level, digital audio must be computed and rendered at *sample rate* (i.e. 44100 samples/second for CD-quality audio). At a somewhat higher level, precisely timed control must be exerted (at *control-rate*) in order to synthesis audio with desired characteristics (i.e. timbres, frequencies, amplitude, filter properties, etc.). Finally, in order to organize computed sound into music, there must be a strong notion of *musical timing* (such as rhythm, sonic "shape" over time, etc.). Finally, the audio synthesis often must be precisely synchronized with input devices, real-time graphics, and other input and output media.

These factors present an interesting and unique challenge to audio synthesis programming, because it deals not only with *what* and *how*, but also always with *when*, and at vastly different time granularities. Therefore, it is essential that the programming language provide flexible and fine-grain control over timing at coarse *and* fine levels.

Existing languages such as C, C++, and Java make it possible to write precise (non-concurrent) audio code – but with a certain degree of difficulty, since there is no inherent notion of time in these languages. More specialized audio programming languages and systems such Pure Data [8], SuperCollider [6], Nyquist [4], and others [7] have facilities to reason about time, but do not embed timing directly in the program flow, nor are they sample-synchronous at the programmer level. Instead, these languages deal with timing by passing parameters to specialized modules that internally use the information to calculate audio. Also, these languages define a fixed control rate to exert control parameters. Because many synthesis elements optimally operate at different control rates, this can be both limiting and inefficient. Also, as a result, sub-control-rate manipulations must be implemented externally (in another language, such as C) in specialized "plug-ins" and imported into these languages.

Closely related to the notion of time are parallelism and concurrency. Sound, as well as music, is most often the simultaneity of many entities and events. Therefore, it would make sense to be able to program audio in a concurrent manner. However, traditional concurrent programming models (such as threads and channels) lend themselves poorly to this endeavor. One reason is that the underlying scheduling/timing mechanisms are often much too coarse (by several orders of magnitude) and imprecise for audio timing. Another important and related challenge is programmability. There is no inherent concurrent programming model that easily and precisely deals with the real-time synchronization of fine-granularity, *strongly-timed* data. Also, traditional synchronization primitives (mutexes, semaphores, condition variables, and channels) suffer from coarse granularity, high overhead, and complexity in their use. Indeed, no existing language supports precise, concurrent programming model for audio. Most languages either support audio concurrency poorly (such as C/C++/Java), or don't support it at all, and resort to dealing with concurrent events by scheduling them explicitly (and serially) from a single-process.

Additionally, there is the problem of representation. In audio synthesis, a good representation should both allow a high degree of programmability while clearly capturing useful information the domain (such as the complex flows and interactions of data and control signals). In the following sections, we show how ChucK addresses each of these issues.

3. CHUCK

ChucK is a *strongly-timed*, concurrent, and on-the-fly audio programming language[9]. It is not based on a single existing language but built from the ground up. Chuck code is type-checked, emitted over a special virtual instruction set, and run in a virtual machine with a native audio engine and a user-level scheduler. It contains the following key features:

- A straightforward way to connect audio *data-flow*.
- A *sample-synchronous* timing mechanism that provides a consistent and unified view of time – embedded directly in the program flow – making ChucK programs easy to maintain and reason about. *Data-flow* is fundamentally decoupled from *time*.
- A cooperative multi-tasking, concurrent programming model *based on time* that allows programmers to add concurrency easily and scalably. Synchronization is accurately and automatically derived from the timing information.
- Multiple, simultaneous, arbitrary, and dynamically programmable *control rates* via the timing mechanism and concurrency.
- A compiler and virtual machine that run in the same process, both accessible from within the language.

ChucK's programming model addresses several problems in audio programming: representation, level of control of data-flow and time, and concurrency. We describe the features and properties of ChucK in the context of these areas.

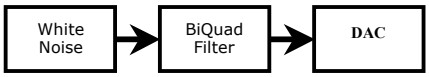
3.1 Representation

Representation deals with the expressive and elegant mapping of syntactical and semantic language constructs to audio and visual concepts. An effective representation should also be straightforward to reason about and maintain. ChucK addresses this problem in both its syntax and semantics. The syntax provides a means to specify *data-flow;* the timing semantics specify when computations occur. In this way, both high-level manipulation and low-level control is achieved. We discuss the syntactical portion here, and present the timing semantics in *Section 3.2*.

The basic sample-rate audio processing module in ChucK is a *unit generator*[5] (or a *ugen*), which generates or operates on audio samples. (Examples include noise generators, sine-wave oscillators, digital filters, and amplitude envelope generators) Unit generators have been shown to be an effective programming abstraction for representing complex audio data flow [4,5,6,8]. ChucK presents a simple syntax and semantics for connecting and manipulating unit generators.

At the heart of the syntax is the *ChucK operator*: a group of related operators (=>, −>) that denote interconnection and direction of data flow. A *unit generator graph* (or *patch*) can be quickly and clearly constructed by using => to connect *ugen*'s in a strongly ordered, left-to-right manner (Figure 2). Unit generators

in the graph implicitly compute one sample at a time. As we will see, control parameters to a unit generator can be modified using =>, in conjunction with the timing mechanism.



(a)

```
noise n => filter f => dac;
```

(b)

Figure 2. (a) A noise-filter *ugen graph* using three unit generators. (b) ChucK statement representing the patch. **dac** is the global sound output variable.

3.2 Level of Control

The level of control and abstraction provided by the language shape what can be done with the language and how it is used. In the context of audio programming, we are concerned not only with control over *data* but also over *time*. The latter deals with control rates and the manner in which time is manipulated and reasoned about in the language. Thus, the question is: *what is the appropriate level and granularity of control for data and time?*

The solution in ChucK is to provide many levels and granularity of control over data and time. The key to having a flexible level of control lies in the ChucK timing mechanism, which consists of three parts. Firstly, ChucK, time is conceptually synchronized with audio data (samples) and exposed in the language. Sample-rate computations are implicit (but also accessible from the language) for unit generators connected (directly or indirectly) to **dac**, the global audio output unit generator. Control rate and musical timing are exposed and delegated to the programmer.

Secondly, time (**time**) and duration (**dur**) are native types in the language. Time refers in a point in time whereas duration is a finite amount of time. Basic duration values are provided by default: `samp` (the duration between successive samples), `ms` (millisecond), `second`, `minute`, `hour`, `day`, and `week`. Additional durations can be inductively constructed using arithmetic operations on existing time and duration values.

```
// construct a unit generator patch
noise n => biquad f => dac;

// loop: update biquad every 100 ms
while( true )
{
    // sweep biquad center frequency
    200 + 400 * math.sin(now*FC) -> f.freq;

    // advance time by 100 ms
    100::ms +=> now;
}
```

Figure 3. *A control loop.* The **=>** ChucK operator is used to control a filter's center frequency. The last line of the loop causes time to *advance* by 100 milliseconds – this can be thought of as the control rate.

Finally, there is a special keyword **now** (of type `time`) that holds the current ChucK time, which starts from 0 (at the beginning of the program execution). **now** is the key to reasoning about and manipulating time in ChucK. Programs can read the globally consistent ChucK time by reading the value of **now**. Also, by assigning time values or adding duration values to **now** causes time to *advance*. As an important *side effect*, this operation causes the current process to *block* (allowing audio to compute) until **now** actually reaches the desired point in time (Figure 3). We call this *synchronization to time*.

This mechanism provides a consistent, sample-synchronous view of time and embeds timing control directly in the code. This formal correspondence between timing and program flow makes programs easier to write and maintain, and fulfills several properties (.e. deterministic order of computation) – therefore, ChucK is said to be *strongly-timed*. Furthermore, *data-flow* is decoupled from *time*, and control rate can be fully throttled by the programmer. Audio rates, control rates, and high-level musical timing are unified under the same timing mechanism.

3.3 Concurrent Audio Programming

Sound and music are often the simultaneity of many precisely timed entities and events. There have been many ways devised to represent simultaneity [4,6,7,8] in computer music languages. However, until ChucK, there hasn't been a truly concurrent and precisely timed programming model for audio. This aspect of ChucK is a powerful extension of the timing mechanism.

The intuitive goal of concurrent audio programming is straightforward: to write concurrent code that shares data as well as time (Figure 4).

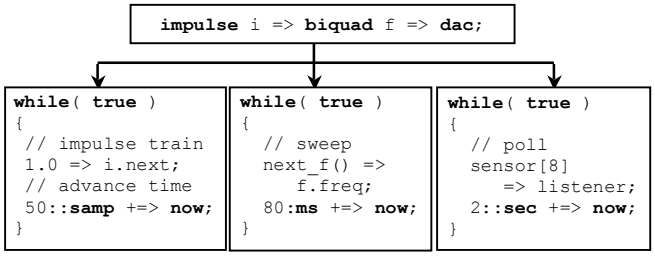


Figure 4. A unit generator patch and three concurrent paths of execution at different control rates (from left: control period = 50 samples, 80 milliseconds, and 2 seconds).

ChucK introduced the concepts of *shreds* and the *shreduler*. A shred is a concurrent entity like a thread[2]. But unlike threads, a shred is a deterministic shred of computation, synchronized by time. Each concurrent path of execution in *Figure 4* can be realized by a shred. Shreds can reside in separate source files or be dynamically spawned (*sporked*) from a single parent shred.

The key insight to understanding concurrency in ChucK is that *shreds are automatically synchronized by time*. Two independent shreds can execute with precise timing relative to each other and the virtual machine, without any knowledge of each other. This is a powerful mechanism for specifying and reasoning about time locally and globally in a synthesis program. Furthermore, it allows for any number of different control rates to execute concurrently and accurately. ChucK concurrency is orthogonal in that programmers can add concurrency without adding additional synchronization to existing code. It is scalable, because shreds are implemented as efficient user-level constructs[1] in the ChucK Virtual Machine. The timing mechanism makes it straightforward to reason about concurrent code with complex timing.

3.4 ChucK Virtual Machine

ChucK code is type-checked, compiled into virtual ChucK instructions, and executed in the ChucK Virtual Machine (Figure 5), which consists of an on-the-fly compiler, a virtual instruction interpreter, a native audio engine, the *shreduler* (which *shredules* the *shreds*), and an I/O manager. The on-the-fly compiler, the *shreduler*, and the virtual machine itself can be accessed as global objects from within the language. For example, a shred can request the compiler to parse and type-check a piece of code dynamically, and then *shredule* the code to execute as part of the same process. This mechanism, along with the timing and concurrency, forms the foundation for our on-the-fly programming model.

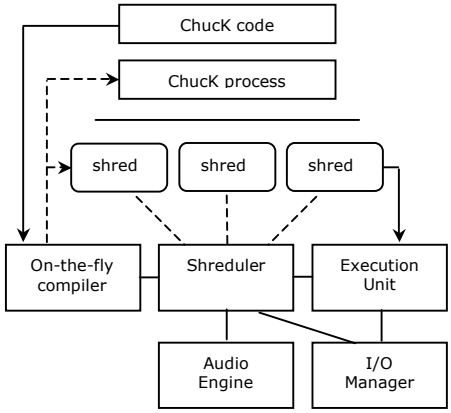


Figure 5. The ChucK Virtual Machine runtime.

3.5 On-the-fly Programming

Using a combination of concurrency, precise programmatic timing, and the architecture of the audio computation model, ChucK makes it possible to write, augment, and modify programs during runtime. In our current framework[10], new shreds can be written and *assimilated* into the virtual machine, fully capable of discovering and sharing both data and timing with the existing process. Similarly, existing shreds can be removed, suspended, or replaced. Programs constructed *on-the-fly* are no different than statically written programs in their content. Real-time audio *and* graphics software can be developed together in this manner. Indeed, audio and visuals can be seamlessly and precisely synchronized under the timing mechanism. The runtime programmability lends itself to rapid experimentation for real-time multimedia development, as well as to emerging interactive performance possibilities.

4. CONCLUSION AND FUTURE WORK

The features of ChucK provide a new, concurrent programming model for writing precise real-time audio and multimedia programs. The on-the-fly programming features of ChucK embody an immediate-run coding aesthetic and encourage runtime interaction and experimentation using the program itself.

Currently, we are continuing to add new language features and support libraries, including ones for real-time visuals (such as GlucK: OpenGL and visualization API for ChucK). We are also exploring new "context-sensitive", audio and multimedia programming environments that understand the deep structure of the program being written, and can use this to help programmers develop on-the-fly programs more efficiently.

Additionally, it would be useful to reduce the granularity of on-the-fly modules (i.e. from shreds to code segments or even instructions). Since ChucK programs are emitted into virtual instructions, we have a great degree of control over the execution and can potentially use this to our advantage.

In conclusion, ChucK is an ongoing project, which seeks to provide a new programming model and tool for researchers, composers, and developers to write and experiment with complex audio synthesis and multimedia programs.

5. ACKNOWLEDGMENTS

Our sincere thanks to the chairs of the ACM Multimedia 2004 Open Source Software Competition, Ketan Mayer-Patel and Roger Zimmerman for providing us the opportunity to present this work.

ChucK is freely available at:

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

6. REFERENCES

[1] Anderson, T. E., Bershal, B. N., Lazowska, E. D., and Levy, H. M., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." *ACM Transactions on Computer Systems*, 10(1):53-79.

[2] Birrell, A. D., "An Introduction to Programming with Threads." *Technical Report SRC-035*, Digital Equipment Corporation, January 1989.

[3] Dannenberg, R. B. and Brandt, E., "A Flexible Real-time Software Synthesis System." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 270-273, 1996.

[4] Dannenberg, R. B., "Machine Tongues XIX: Nyquist: a Language for Composition and Sound Synthesis." *Computer Music Journal*, 21(3):50-60, 1997.

[5] Mathews, M. V. *The Technology of Computer Music*. MIT Press, 1969.

[6] McCartney, J., "Rethinking the Computer Music Programming Language: SuperCollider." *Computer Music Journal*, 26(4):61-68, 2002.

[7] Pope, S. T., "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal*, 17(2):23-54, 1993.

[8] Puckette, M., "Pure Data." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 269-272, 1997.

[9] Wang, G. and Cook, P. R., "ChucK: a Concurrent and On-the-fly Audio Programming Language." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 219-226, Singapore, 2003.

[10] Wang, G. and Cook, P. R., "On-the-fly Programming: Using Code as an Expressive Musical Instrument." *In Proceedings of the Internal Conference on New Interfaces for Musical Expression*. pp. 138-143, Hamamatsu, Japan, 2004.