Ge Wang,* Perry R. Cook,† and Spencer Salazar*
*Center for Computer Research in Music and Acoustics (CCRMA)
Stanford University
660 Lomita Drive, Stanford, California 94306, USA
{ge, spencer}@ccrma.stanford.edu
†Department of Computer Science
Princeton University
35 Olden Street, Princeton, New Jersey 08540, USA
prc@cs.princeton.edu

ChucK: A Strongly Timed Computer Music Language

Abstract: ChucK is a programming language designed for computer music. It aims to be expressive and straightforward to read and write with respect to time and concurrency, and to provide a platform for precise audio synthesis and analysis and for rapid experimentation in computer music. In particular, ChucK defines the notion of a strongly timed audio programming language, comprising a versatile time-based programming model that allows programmers to flexibly and precisely control the flow of time in code and use the keyword `now` as a time-aware control construct, and gives programmers the ability to use the timing mechanism to realize sample-accurate concurrent programming. Several case studies are presented that illustrate the workings, properties, and personality of the language. We also discuss applications of ChucK in laptop orchestras, computer music pedagogy, and mobile music instruments. Properties and affordances of the language and its future directions are outlined.

What Is ChucK?

ChucK (Wang 2008) is a computer music programming language. First released in 2003, it is designed to support a wide array of real-time and interactive tasks such as sound synthesis, physical modeling, gesture mapping, algorithmic composition, sonification, audio analysis, and live performance. The language presents a single mechanism for precisely manipulating and reasoning about time across drastically different timescales, from individual samples to musical events, and potentially much longer durations. ChucK is the first computer music language to introduce the concept of `now` as a central timing construct, synchronously binding it to the audio sample stream (via the `now` keyword), and thereby affording a type of temporally deterministic, sample-accurate programming model. Based on this timing mechanism, ChucK also provides the ability to write concurrent audio code with the same timing assurances, internally synchronized by the timing information. Together, this precise control over time and the time-based concurrent programming model form the notion of a *strongly timed* computer music programming language.

Computer Music Journal, 39:4, pp. 10–29, Winter 2015
doi:10.1162/COMJ_a_00324
© 2015 Massachusetts Institute of Technology.

Two Observations about Audio Programming

Time is intimately connected with sound and is central to how audio and music programs are created and reasoned about. This may seem like an obvious point—as sound and music are intrinsically time-based phenomena—yet we feel that control over time in programming languages is underrepresented (or sometimes over-abstracted). Low-level languages such as C/C++ and Java have no inherent notion of time but allow custom data types to represent time, which can be cumbersome to implement and to use. High-level computer music languages tend to abstract time too much, often embodying a more declarative style and connecting elements in a way that assumes quasi-parallel modules, (e.g., similar to analog synthesizers) while hiding much of the internal scheduling. Also, timing has been traditionally broken up into two or more distinct, internally maintained rates (e.g., audio rate and control rate, the latter often arbitrarily determined for the programmer with a single rate maintained once the system begins execution). The main problem with these existing programming

Figure 1. A simple program that moves through time and randomly chooses a frequency every 100 msec. The manner and rate of moving through time is precise, dynamic, and completely up to the programmer.

```
// Synthesis patch
SinOsc foo => dac;

// Infinite time loop
while( true )
{
    // Randomly choose a frequency
    Math.random2f( 30, 1000 ) => foo.freq;
    // Advance time
    100::ms => now;
}
```

models is that the programmer knows "what" but does not always know "when," and typically cannot exert control beyond a fixed control rate, rendering low-level timing difficult or even impossible (e.g., for granular synthesis or synthesis models with precise feedback).

Second, sound and music often require the simultaneity of many parallel processes. Thus it stands to reason that a programming language for music can fundamentally benefit from a concurrent programming model that flexibly captures parallel processes and their interactions. The ability to program parallelism must be cognizant of time and yet operate independently of time. In other words, this functionality must be "orthogonal" to time to provide the maximal degree of freedom and expressiveness.

From these two observations, the ChucK insight is to expose programmability and to provide precise control over time (at all granularities), in cooperation with a time-based concurrent programming model. In particular, this entails making time itself both computable and directly controllable at any rate. Programmers specify the algorithm, logic, or pattern, according to which computation is performed in time, by embedding explicit timing information within the code. Based on this framework, the language's runtime system ensures properties of determinism and precision between the program and time. Furthermore, programmers can specify concurrent code modules, each of which can independently control its own computations with respect to time and can be synchronized to other modules via time and other synchronization mechanisms.

A Simple ChucK Program

As an example, consider the program in Figure 1, which creates a patch consisting of a sine-wave generator connected to the audio output (represented in ChucK by the `dac` object), and changes the frequency of oscillation randomly every 100 milliseconds.

In reading this (or any ChucK) code, it is often helpful to follow the code sequentially and through the various control structures (e.g., for and while loops, if–else statements), and noting the point (or points) at which "ChucK time" is advanced. For example, line 2 instantiates a `SinOsc` (sine-wave generator) called `foo`, and connects it to `dac` (an abstraction for audio output). The program flow enters the while loop on line 5, then randomly chooses a frequency between 30 and 1,000 Hz for the sine wave (line 8). The program advances time by 100 milliseconds on line 10, before returning to check the loop conditional again on line 5. It is important at this point to understand what really happens on line 10. By *chucking* (a way to send values in ChucK) the duration `100::ms` to the special ChucK time construct `now`, the program flow pauses and returns control to the ChucK virtual machine and synthesis engine, which generates audio for exactly (precisely, to the nearest sample) 100 milliseconds, before returning control back to our program. In this sense, advancing time in ChucK is similar to a `sleep()` functionality found in many languages, such as C, C++, and Java. The difference is that ChucK synchronously guarantees precision in logical time (mapped to the nearest audio sample), allowing one to specify versatile timing behaviors across the system. Furthermore, the same method of reading the code can be applied to complex ChucK programs, to reason about the timing in a straightforward way.

The Meaning of "Strongly Timed"

We define a *strongly timed* programming language as one in which there is a well-defined, precise ("sample-synchronous"), and predictable mechanism for controlling time at various timescales (temporal determinism) and across concurrent code modules (time-mediated concurrency). ChucK constructs a view of logical time (realized through the `now` keyword) that can be independent from "actual" time. Via this system, programs can be designed and specified without concern for external factors, such as machine speed, portability, and timing behavior across different systems, allowing one to more easily specify, debug, and reason about programs. Furthermore, this mechanism can be used to specify deterministic concurrent behavior. Owing to this model, ChucK appears to be unique in supporting tightly interleaved control over audio computation, allowing the programmer to move smoothly from the domain of digital signal processing and audio synthesis at the sample level to higher levels of musical and interactive control—directly in a single, high-level language.

In ChucK, the programmer manipulates time via the `now` construct (ChucK's logical present time). By setting the value of `now` to a future time, the programmer can allow ChucK time to advance accordingly. This is achieved precisely (control of time is sample synchronous and is accurate to the sample) and deterministically (the same code will behave identically across different executions and machines, free from underlying hardware timing and nondeterministic scheduling in the operating system). The programmer can make the extremely useful assumption that they are coding in a type of "suspended animation." ChucK time does not advance until explicitly instructed, and any code sequence between time-altering instructions can be deterministically mapped to a particular point on a logical ChucK timeline. This provides a framework to specify and reason about time, and naturally establishes a strong temporal ordering of when code executes.

More generally, we define *logical time* as the deterministic accounting of time internal to an audio environment (typically by counting audio samples), and which—depending on the environment—may or may not be exposed to the programmer. By contrast, we think of *actual time* more as the continuous flow of time as we experience it (system clocks and timers track this). Logical time is a useful construct because it is decoupled from actual time. In real-time systems, logical time aims to keep up with actual time while maintaining some notion of determinism. (In non-real-time situations logical time may run "as fast as possible"—which can be faster or slower than actual time—e.g., we might compute 60 logical seconds of audio in 1 second of actual time.)

The second facet of strong timing relates to concurrent programming, which is an attractive asset because it allows many complex interactions to be refactored into independent sequential modules with considerable versatility (Hoare 1978). Yet concurrency can bring difficulties for the programmer as well. For example, preemptive thread programming must manage the indeterminism of scheduling to avoid incorrect behavior (e.g., through the use of locks, semaphores, and other synchronization mechanisms); although nonpreemptive concurrency is more predictable, it requires programmers to manually yield the currently running module, which can be cumbersome to write and suboptimal in performance. Is it possible to design a solution that keeps the benefits of concurrency and naturally avoids the difficulties?

While this remains a difficult task in general, ChucK's solution directly takes advantage of its *domain-specificity*: ChucK programmers already are required to continually and explicitly deal with time in order to work with sound, and this interaction can be used to schedule concurrency in a predictable, simple manner. (Advancing time in ChucK effectively yields the current code, but we argue that it is more natural for the programmer because the focus is on time rather than context switching—a user would advance for a single code module the same as they would for concurrent modules.) Communication and context switching between concurrent blocks are efficient and logically precise, and they uphold the sample-synchronous and deterministic properties of ChucK's timing mechanism. Moreover, this can be achieved without losing generality in the language—low-level audio algorithms (e.g., granular synthesis, formant wave functions, physical models) can be directly implemented in ChucK.

Related Work

ChucK descends from the lineage of "Music N" languages (starting with Music V, cf. Mathews 1969), which also directly influenced computer music programming environments such as CSound (Vercoe 1986), Common Lisp Music (Schottstaedt 1994), Max/MSP (Puckette 1991), Pure Data (Puckette 1996), SuperCollider (McCartney 2002), and Nyquist (Dannenberg 1997)—the last of these an interactive computer music language based on Lisp (Touretzky 1984). While adopting familiar elements of audio programming found in earlier Music N languages, Nyquist (along with SuperCollider) is among the first computer music languages to remove the distinction between the "orchestra" (sound synthesis) and the "score" (musical events)—both can be implemented in the same framework. This tighter integration allows both synthesis and musical entities to be specified using a shared "mindset," favoring the high customizability of code over the ease and simplicity of note lists.

Some music programming languages do not directly synthesize audio, but operate purely at a higher symbolic level. Formula, short for Forth Music Language, is a programming language for computing control signals based on concurrent processes operating in a unified runtime environment (Anderson and Kuivila 1991). Various processes can be specified to compute pitch sequences as well as control for parameters such as volume, duration, and articulation. Haskore is a set of modules in the Haskell programming language created for expressing musical structures in a high-level declarative style of functional programming (Hudak et al. 1996). Like Formula, it is not a language for describing sound synthesis, but primarily for music information (in the case of Haskore, mostly Western music).

The inspiration from languages such as these, as well as the core departures from them, form the thesis for this article. For example, unlike any synthesis language before it, ChucK's timing mechanism inherently eliminates the traditional need to deal with system-side control rate. Furthermore, the aesthetics of the language itself favor a more immediate, deterministic mindset for specifying computer music programs.

ChucK is not the first to address this issue of enabling low-level timing in a high-level audio programming language. Music V, HMSL (Burk, Polansky, and Rosenboom 1990), and Nyquist have all embodied some form of sample-synchronous programming model. Chronic (Brandt 2002), with its temporal type constructors, was designed to make arbitrary, sub-control-rate timing programmable for synthesis. Although the mechanisms of Chronic are very different from ChucK's, one aim is the same: to free programmers from having to implement "black-box" unit generators in a lower-level language, such as C/C++, when a new lower-level feature is desired. In a sense, Chronic "zooms out" and deals with time in a global, non-real-time way. On the other hand, ChucK "zooms in" and operates at specific points in time, in a "pathologically" immediate manner. Another way to think of the difference: in Chronic the time is "all times"; in ChucK time is always instantly "now."

The practice of enabling the programmer to operate with an arbitrarily fine granularity is partially derived from the Synthesis Tool Kit (STK; cf. Cook and Scavone 1999), which exposes a manageable programming interface for efficient single-sample operations, while hiding the complexity of audio input/output management.

In a high-level sense, the idea of concurrency in ChucK is similar to the idea of mixing independent "tracks" of audio samples in CMix (Lansky 1990) and later RTCMix (http://rtcmix.org). Lansky's original idea was to provide a programming environment where the composer can deal with and perfect individual parts independently (Pope 1993). ChucK extends this idea by allowing full programmability for each concurrent code module.

On-the-Fly Programming Environments

ChucK's 2003 release coincided with the rise of live coding and influential packages such as JITLib (the Just in Time Library) for SuperCollider (Collins et al. 2003). Indeed, on-the-fly programming, although not the focus of this article, was a goal in the creation of ChucK (Wang and Cook 2004a). In the decade since, a number of new environments have been developed to support programming either as live performance practice or as a tool for rapid prototyping. They include Sonic Pi and Overtone (Aaron and Blackwell 2013), the latter using a Clojure front-end language with SuperCollider as the back-end synthesis engine; Gibber (Roberts et al. 2013), a browser-based environment for live coding in pure JavaScript; Tidal (http://yaxu.org/tidal), a mini-language for live coding embedded in Haskell; as well as Extempore, the successor to Impromptu (Sorenson 2005).

Extempore (Sorenson and Gardner 2010) takes on live coding with a notion of a "cyber-physical programming," where a human programmer interacts with a distributed real-time system procedurally by writing and editing code on the fly. The syntax of Extempore is based on Scheme, and uses a just-in-time compiled backend using LLVM (http://llvm.org) for real-time audio synthesis. Extempore makes use of temporal recursion, a design pattern in which a function can schedule itself as its final action, establishing a callback loop. A key distinction between Extempore and ChucK is that the former uses an asynchronous "schedule and forget" style of programming (i.e., events are scheduled to be executed in the future while the current code evaluation proceeds without blocking), whereas ChucK code is completely synchronous (i.e., the code waits precisely until the desired timing is fulfilled). In other words, whereas environments like Extempore schedule events, ChucK code schedules itself. The tradeoff is that while ChucK's synchronous semantics provide an elegant determinism in reasoning about time, Extempore (like SuperCollider) favors an asynchronous approach that more naturally facilitates nondeterministic distributed architectures.

Synchronous Reactive Systems

In addition to the realm of audio and music programming, it is worthwhile to provide context for this work with respect to synchronous languages for reactive systems. A reactive system maintains an ongoing interaction with its environment, rather than (or in addition to) producing a final value upon termination (Halbwachs 1993). Typical examples include air traffic control systems and control kernels for mechanical devices (such as wristwatches), trains, and even nuclear power plants. These systems must react to their environment at the environment's speed. They differ from transformational systems, which emphasize data computation instead of the ongoing interaction between systems and their environments, and from interactive systems, which influence and react to their environments at their own rate (e.g., Web browsers).

In synchronous languages for reactive systems, a *synchrony hypothesis* states that computations happen infinitely fast, allowing events to be considered atomic and truly synchronous. This affords a type of logical determinism that is an essential aspect of a reactive system (which should produce the same output at the same points in time, given the same input), and reconciles determinism with the modularity and expressiveness of concurrency. Such determinism can lead to programs that are significantly easier to specify, debug, and analyze compared with nondeterministic ones—for example, those written in more "classical" concurrent languages. Several programming languages not dealing specifically with audio have embodied this synchrony hypothesis, including: Esterel, a textual imperative language (Berry and Gonthier 1992); Lustre, a declarative dataflow language for reactive systems (Caspi et al. 1987); and SIGNAL, a dataflow language for signal processing (Le Guernic et al. 1985).

ChucK embodies some elements of synchronous languages, as there is an ongoing interaction between code and audio synthesis processes, although this interaction is not strictly reactive. In ChucK, computation is assumed to happen instantaneously with respect to logical ChucK time, which can only be advanced as explicitly requested and allowed by the system. Hence, any code between instructions to advance time can be considered atomic in time—presumed to happen instantaneously at a single point in time. This is a highly useful feature—the programmer can depend on the language to map any finite amount of computation to a specific point in time. Existing synchronous languages emphasize reaction, whereas ChucK's design goals and programming style are intended to be reactive as well as proactive and interactive—code drives and defines the environment in addition to responding to it. (This is embodied in even simple programs such as that in Figure 1.) The ChucK programming model offers events and signals as well as the ability to specify concurrent processes that move themselves through logical time, to both control and to define the system. This encourages a fundamentally different, proactive mentality when programming. Additionally, ChucK presents a visible and centralized view of logical time (via `now`) that reconciles logical time with real time. This mechanism deterministically couples and interleaves user computation with audio computation, providing a continuous notion of time mapped explicitly to the audio synthesis stream.

Core Language Features

Motivated by these observations, we present core elements that form the foundation of ChucK:

1. A unifying ChucK operator intended to represent action and directionality.
2. A precise timing model that unifies high-level and low-level timing and is straightforward to write as well as reason about from code. This is the foundation of strongly timed programming in ChucK.
3. A precise concurrent programming model that supports arbitrarily fine granularity, as well as multiple, simultaneous, and dynamic rates of control. Integrated into the timing model, this forms the concurrent aspect of strongly timed audio programming.

The ChucK Operator (=>)

At the heart of ChucK's language syntax is the ChucK operator (written as =>). Although this is not an essential element for strongly timed programming, its notation pervades the experience of working with ChucK. This left-to-right operator originates from the slang term "to chuck," meaning to throw an entity into or at another entity. The language uses this notion to help express sequential operations and dataflow. The => operator (and related operators) form the "syntactic glue" that binds ChucK code elements together. The ChucK operator conveys the ordering of synthesis elements (e.g., unit generators) and is overloaded on operand types so it makes sense in a variety of contexts (UGen connection, UGen parameter control, assignment, etc.). In contrast to graphical patching environments such as Max/MSP and Pure Data, ChucK's text-based combination of the ChucK operator, timing directives, and object-oriented programming make an efficient shorthand for precisely and clearly representing signal chains and temporal behaviors. The ChucK operator, as used to connect unit generators, visually resembles physical and graphical patching, a feature that resonates well with new ChucK programmers, especially those with previous computer music experience. Furthermore, ChucK includes a flexible set of base unit generators and library functions, making built-in synthesis and logic easy "out of the box."

Controlling Time and Temporal Determinism

A central idea in ChucK's approach is to make time itself computable, and to allow a program to be "self aware" in the sense that it always knows its position in time, and can control its own progress over time. Furthermore, many program components can share a central notion of logical ChucK time, making it possible to automatically synchronize parallel code based on time, as well as to precisely express sophisticated temporal behavior of a program. This gives rise to a programming mentality in which programs have intimate, precise, and modular control over their own timing. With respect to synthesis and analysis, an immediate ramification is that control can be asserted over any unit generator at any time and at any rate. In order to make this happen:

1. ChucK provides `time` and `dur` as native types in the language (to represent time and duration values, respectively).
2. The language allows well-defined arithmetic on time and duration (see Table 1).

Table 1. Arithmetic Operations on Operand Types

Type	*Op*	*Type*		*Result Type*	*Commute?*
`dur`	+	`dur`	→	`dur`	Yes
`dur`	−	`dur`	→	`dur`	
`dur`	*	`float`	→	`dur`	Yes
`dur`	/	`float`	→	`dur`	
`dur`	/	`dur`	→	`float`	
`time`	+	`dur`	→	`time`	Yes
`time`	−	`dur`	→	`time`	
`time`	−	`time`	→	`dur`	

The operand types, which can be `time`, `dur`, or `float`, and the types of the results. The final column indicates whether the operation is commutative.

3. The model provides a deterministic and total mapping between code, time, and audio synthesis. It is straightforward to reason about timing from anywhere in a program.
4. The language provides `now`, a special keyword (of type `time`) that holds the current ChucK time. It provides a flexible granularity that can operate at (or finer than) the sample rate, and it provides a way to work with time in a deterministic and well-defined manner.
5. ChucK offers a globally consistent means to advance time from anywhere in the program flow: by duration (`D => now;`) or by absolute time (`T => now;`).

The code example shown in Figure 1 describes one of two ways to "advance time" in ChucK. In the first method, "chuck to now," the programmer can allow time to advance by explicitly chucking a duration value or a time value to `now`, as shown previously. This embeds the timing control directly in the language, giving the programmer the ability to precisely "move forward" in ChucK time. The second method to advance time in ChucK is called "wait on event." An event could represent synchronous software triggers, as well as asynchronous messages from MIDI, Open Sound Control (OSC; Wright and Freed 1997), serial, and human interface devices (HIDs). User code execution will resume when the synchronization condition is fulfilled. While waiting for the event, the ChucK virtual machine is free to schedule audio synthesis and other synchronous computations. Wait on event is similar in spirit to chuck to `now`, except that events have no precomputed time of arrival.

It is essential to note that the logical ChucK time stands still until explicitly instructed to move forward in time (by one of these two methods to advance time). This allows an arbitrary amount of computation to be specified at any single point in time, and fulfills the synchrony assumption that computation can be viewed as happening infinitely fast. Although this seems like an absurd assumption to make, it establishes a type of temporal determinism to logically reason about when things happen. In practice, real-time audio will remain robust up to the limit imposed by computing speed, after which audio will experience glitches and interruptions—as is the case for any real-time synthesis system. The deterministic timing principle is always upheld, however, independently of robustness or performance. By combining this abstraction with the mapping of time to the audio sample stream, ChucK's timing mechanism provides absolute assurance that code is always logically mapped in time.

Another important point to note is that all synthesis systems, at some level, have to be sample synchronous (samples precisely synchronized with time)—or else DSP just does not happen. Most languages generally do not expose the ability to control all computational timing directly, however. In contrast, ChucK makes it possible to exert direct and precise control over time at all granularities. The deterministic assurance of time relationship between code and audio signal processing is essential to the design of many synthesis algorithms, especially at small timescales (e.g., on the order of tens of milliseconds and smaller, as with the single-sample example in Figure 2). As for dealing with higher-level, more "musical" timescales (e.g., fractions of seconds and above), the same timing mechanism can be used, unifying low- and high-level timing into a single construct. Alternately, the programmer can abstract the timing mechanism into high-level functions (e.g., a `playNote()` function that also takes a duration) and data structures (e.g., an object that can take and interpret an array of numbers as a temporal pattern). In these higher-level instances,

Figure 2. FM synthesis "by hand" in ChucK. Although there are more efficient means to do FM synthesis in ChucK, this extreme example program illustrates precise control over time at any rate—in this case, one sample at a time.

```
// Carrier
SinOsc c => dac;
// Modulator (driven by blackhole -- like dac but no sound)
SinOsc m => blackhole;

// Carrier frequency
220 => float cf;
// Modulator frequency
550 => float mf => m.freq;
// Index of modulation
200 => float index;

// Time loop
while( true )
{
  // Modulate around cf by polling modulator using .last()
  // (.last() returns the most recently computed sample)
  cf + (index * m.last()) => c.freq;
  // Advance time by duration of one sample
  1::samp => now;
}
```

the timing mechanism serves as a temporal building block that can be used to craft arbitrarily complex behavior in the user's programming style.

In summary, the timing mechanism moves the primary control over time from inside opaque unit generators, and a control layer of fixed time granularity, to the code directly, explicitly coupling computation to time. The programmer knows not only what is to be computed, but also precisely when, relative to ChucK time. This global control over time enables programs to specify arbitrarily complex timing, allowing a programmer or composer to "sculpt" a sound or passage into perfection by operating on it at any granularity or rate.

Time-Based Concurrency

Up to this point, we have discussed programming ChucK using one path of execution, controlling it through time. Time alone is not sufficient, however, because audio and music often involve the presence and cooperation of multiple parallel processes. We desire concurrency to expressively capture such parallelism.

If ChucK's timing serializes operations, the concurrent programming model parallelizes multiple independent code sequences precisely and deterministically. Concurrency in ChucK works because the programmer already supplies precise timing information, which can be used, it turns out, to interleave computation across all code running in parallel.

The design uses a form of nonpreemptive concurrent programming, whereby programmers have to explicitly yield the current process. ChucK programmers already do this when they explicitly advance time on a continual basis—and these operations contain all the necessary information to automatically schedule concurrency modules (e.g., when to yield, when to wake up). This concurrency adds no additional work for the programmer and requires no further synchronization. Crucially, this inherits the sample-precise determinism that the timing mechanism provides.

Such deterministic concurrency offers the versatility to specify behaviors as independent sequences of instructions, allowing complex systems to be broken down into synchronous building blocks, called *shreds* in ChucK, spawned via a special `spork ~`

operation on functions (which serves as entry points and bodies of code for the shreds). A shred, much like a thread, is an independent, lightweight process, which operates concurrently and can share data with other shreds. But unlike conventional threads, whose execution is interleaved in a nondeterministic manner by a preemptive scheduler, a shred is a deterministic piece of computation that has sample-accurate control over audio timing, and is naturally synchronized with all other shreds via the shared timing mechanism and synchronization constructs called *events*.

ChucK shreds are programmed in much the same spirit as traditional threads, with the exception of several key differences:

1. A ChucK shred cannot be preempted by another. This not only enables a single shred to be locally deterministic, but also an entire set of shreds to be globally deterministic in their timing and order of execution.
2. A ChucK shred must voluntarily relinquish the processor for other shreds to run. (In this, shreds are like nonpreemptive threads.) When a shred advances time or waits for an event, it relinquishes the processor, and gets "shreduled" by the "shreduler" to resume at a future logical time. A consequence of this approach is that shreds can be naturally synchronized to each other via time, without using any traditional synchronization primitives.
3. ChucK shreds are implemented completely as user-level primitives, and the ChucK virtual machine runs entirely in user space. User-level parallelism has significant performance benefits over kernel threads, allowing "even fine-grain processes to achieve good performance if the cost of creation and managing parallelism is low" (Anderson et al. 1992, p. 54). Indeed, ChucK shreds are lightweight—each contains only minimal state. The cost of context switching between ChucK shreds is also low, since no kernel interaction is required.

Traditionally, concurrency—especially the preemptive kind—is difficult to deal with, even for seasoned programmers. Race conditions, possibilities for deadlock, and other common pitfalls are easy to introduce and difficult to track down, stemming from the inherently nondeterministic and hence imprecise nature of preemptive scheduling. ChucK's time-based concurrency sidesteps these pitfalls by removing preemption and deriving the scheduling only by requesting that each shred mange its own timing behavior. One potential drawback of nonpreemptive concurrency is that a single shred could hang the ChucK virtual machine (along with all other active shreds) if it fails to relinquish the processor. There are ways to alleviate this drawback, however. For example, any hanging shreds can easily be identified by the ChucK virtual machine (it would be the currently running shred), and it would be straightforward to locate and remove the shred.

Multishredded programs can make the task of managing concurrency and timing much easier (and more enjoyable), just as threads make concurrent programming manageable and potentially increase overall system throughput. In this sense, shreds are powerful programming constructs. We argue that the flexibility of shreds to support deterministic, precisely timed, concurrent audio programming significantly outweighs the potential drawbacks.

Aside from asynchronous input events (e.g., incoming HID, MIDI, or OSC messages), a ChucK program is completely deterministic in nature—there is no preemptive background processing or scheduling. The order in which shreds and the rest of the ChucK virtual machine execute is completely determined by the timing and synchronization specified in the shreds. This makes it easy to reason about the global sequence of operations and timing in ChucK, and it enables multiple shreds to run at independent rates, at which they can assert control over synthesis and other parameters.

This design yields a programming model in which multiple concurrent shreds synchronously construct and control a global unit-generator network over time. The 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 is not feasible.

Figure 3. Singing synthesis program (some code omitted for brevity) (continued on next page).

```
// Synthesis patch
Impulse i => TwoZero t => TwoZero t2 => OnePole lpf;

// Formant filters
lpf => TwoPole f1 => Gain node => NRev reverb => dac;
lpf => TwoPole f2 => node;
lpf => TwoPole f3 => node;

// ... (Omitted: initialization code to set formant filter Qs,
// adjust reverb mix, etc.) ...

spork ~ generate(); // Concurrency: spawn shred #1: voice source
spork ~ interpolate(); // Spawn shred #2: interpolate pitch and formants

while( true ) // Shred #3: main shred
{
  // Set next formant targets
  Math.random2f( 230.0, 660.0 ) => target_f1freq;
  Math.random2f( 800.0, 2300.0 ) => target_f2freq;
  Math.random2f( 1700.0, 3000.0 ) => target_f3freq;
  // Random walk the scale, choose next frequency
  32 + scale[randWalk()] => Std.mtof => freq;
  // Set target period from frequency
  1.0 / freq => target_period;
  // Wait until next note
  Std.rand2f( 0.2, 0.9 )::second => now;
}

// Shred #1: generate pitched source, with vibrato
fun void generate()
{
  while( true )
  {
    // Fire impulse!
    masterGain => i.next;
    // Advance phase based on period
    modphase + period => modphase;
    // Advance time (modulated to achieve vibrato)
    (period + 0.001*Math.sin(2*pi*modphase*6.0))::second => now;
  }
}
```

Case Study: Singing Synthesis

To demonstrate both the timing mechanism and concurrency at work, we present a code example to synthesize singing (see Figure 3).

This code example demonstrates three concurrent shreds working together, each at different rates. Shred 1, `generate()`, explicitly creates an impulse train as a pitched source (time advancement changes dynamically as a function of current pitch and to create vibrato); shred 2, `interpolate()`, smoothly interpolates the fundamental period as well as three formant frequencies (rate: `10::ms`); shred 3, the main shred, randomly chooses formant frequencies

Figure 3. Singing synthesis program (some code omitted for brevity) (continued from previous page).

```
// Shred #2: to perform interpolation for various parameters
fun void interpolate()
{
  0.10 => float slew; // Slewing factor to control interpolation rate
  while( true )
  {
    (target_period - period) * slew + period => period;
    (target_f1freq - f1freq) * slew + f1freq => f1freq => f1.freq;
    (target_f2freq - f2freq) * slew + f2freq => f2freq => f2.freq;
    (target_f3freq - f3freq) * slew + f3freq => f3freq => f3.freq;
    10::ms => now;
  }
}
```

and the next frequency to be sung (rate: random intervals, between `0.2::ms` and `0.9::ms`). The shreds all operate precisely at their respective optimal rates.

Architecture Design

To support the behaviors of the ChucK language, a variety of system design decisions were required. The architecture includes a dedicated lexer, a parser, a typing checker and type system, and a virtual machine (VM) employing a user-level shreduler, which shredules the shreds. We address the core components of the system and outline the central shreduling algorithms.

Architecture Overview

ChucK programs are type-checked, emitted into ChucK shreds containing bytecode, and then interpreted in the virtual machine. The shreduler 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 synthesis network. Additionally, shreds can perform computations and change the state of any unit generators and analyzers at any point in time. These components of the ChucK runtime are depicted in Figure 4.

Audio is synthesized from the global unit-generator graph one sample at a time by "pulling" samples, starting from dedicated UGen "sinks," such as `dac` (the main audio output). Time, as specified in the shreds, is mapped by the system to the audio-synthesis stream. When a shred advances time, it can be interpreted as the shred's shreduling itself to be woken up at some future sample. The passage of time is data-driven, and this guarantees that the timing in the shreds is bound to the audio output and not to any other clocks. Furthermore, it guarantees that the final synthesis/analysis result is "correct," reproducible, and sample-faithful, regardless of whether the system is running in real time or not. Additional processes interface with I/O devices 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 (Wang and Cook 2004b).

From Code to Bytecode

Compilation of a ChucK program follows the standard phases of lexical analysis, syntax parsing, type checking, and emission into instructions (see Figure 5). Code is emitted into VM instructions (the ChucK VM instruction set contains more than a hundred different instructions, from simple arithmetic and memory operations to complex time-advance instructions), as part of a new shred, in class methods, or as globally available routines. The compiler runs as part of the virtual machine, and can compile and run new programs on demand.

Figure 4. ChucK's runtime architecture.

Figure 5. First phases in the ChucK compiler.

Figure 6. Components of a ChucK shred.

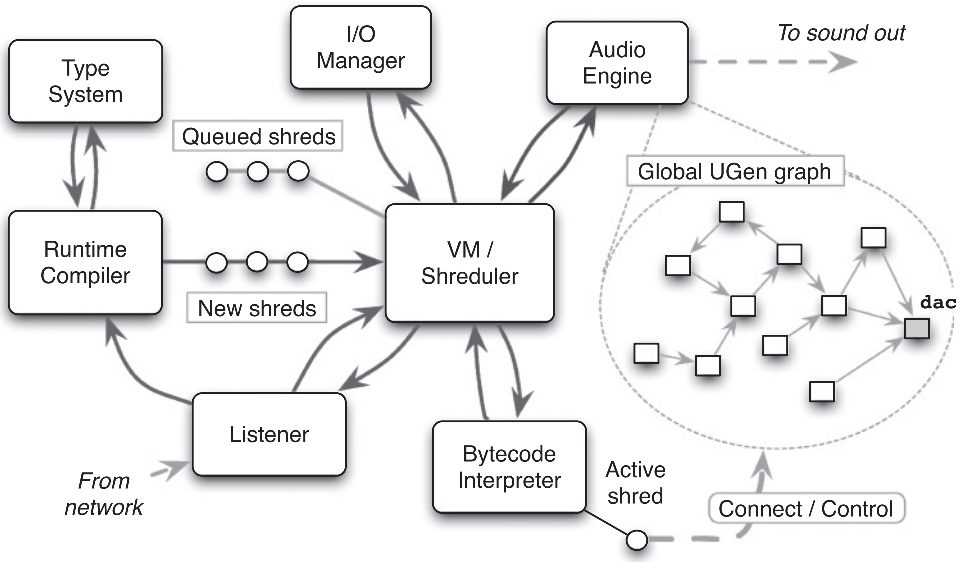


Figure 4

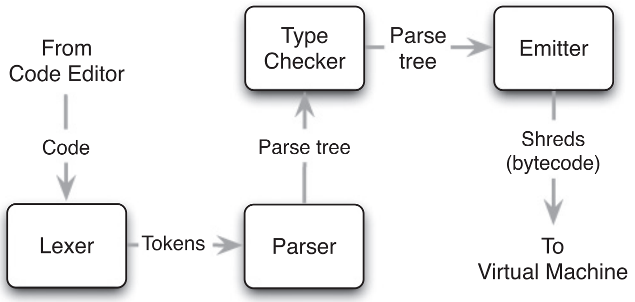


Figure 5

By default, all operations, including instruction emission, take place in main memory. This has the advantage of avoiding intermediate steps of writing instructions to disk and many costly load-time memory translations that would be necessary if the compiler and VM were to run in separate processes. The disadvantage of this in real time is that the compilation must be relatively fast, which precludes the possibility of many advanced compiler optimizations. In practice, this is manageable, and the global UGen graph computations often dominate runtime-compiled shred computations.

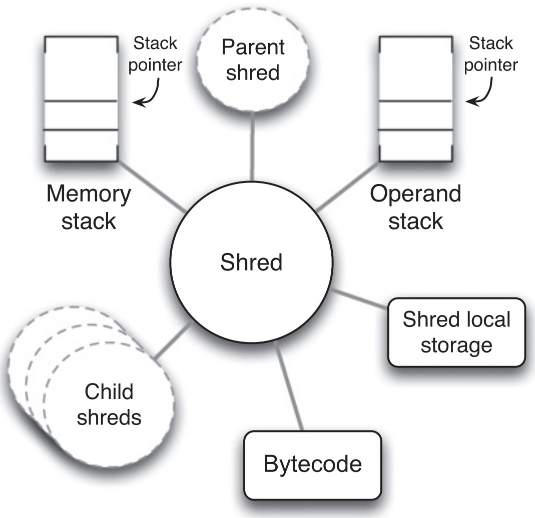
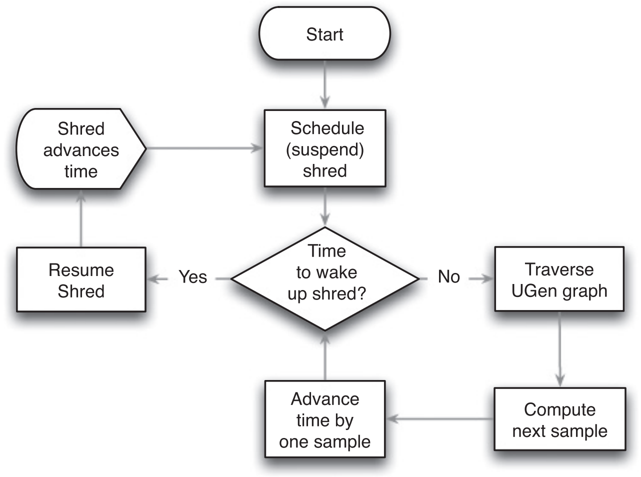


Figure 6

Virtual Machine and Shreduler

After compilation, a shred is passed directly to the ChucK virtual machine, where it is shreduled to start execution immediately. Each shred has several components, as shown in Figure 6: (1) bytecode

Figure 7. Single-shredded shreduling algorithm.



instructions emitted from the source code; (2) an operand stack for local calculations (functionally similar to hardware registers); (3) a memory stack to store local variables at various scopes, e.g., across function calls; (4) references to child shreds (shreds spawned by the current shred) and a parent shred (if any); and (5) shred-local storage, including a local view of `now`, which can be a fractional sample away from the system-wide `now`, and is used to maintain a shred's own, sub-sample timing.

The state of a shred is completely characterized by the content of its stacks and their respective pointers. It is therefore possible to suspend a shred between two instructions and resume it in full at a later ChucK time. A shred is suspended only after instructions that advance time. Shreds can spawn (or "spork," in ChucK parlance, as previously shown in Figure 3) and remove other shreds.

The shreduler serializes the synchronous execution of shreds with that of the audio engine, while maintaining the system-wide ChucK `now`. The value of `now` is mapped to the number of samples in the audio synthesis stream that have elapsed through the virtual machine since the beginning of the virtual machine.

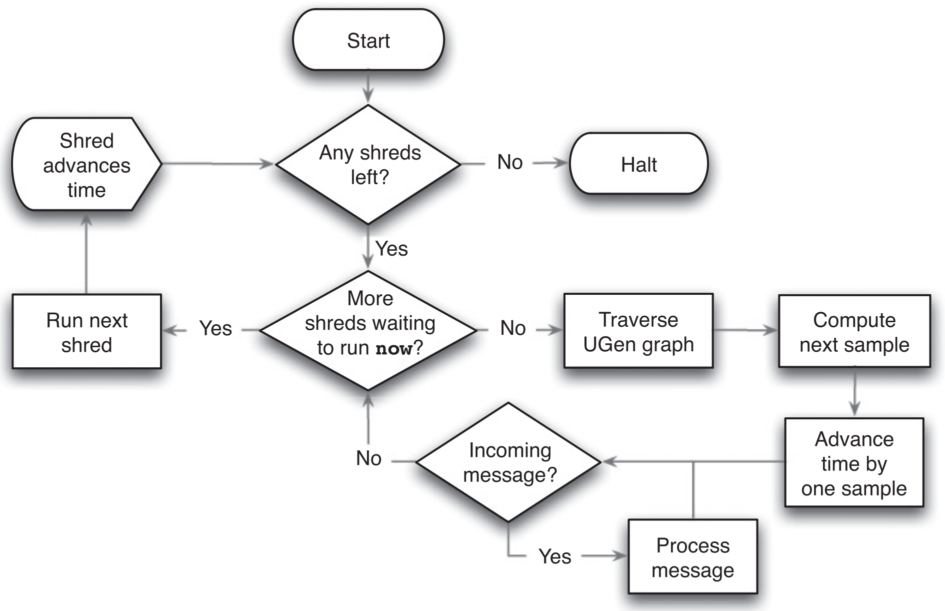
For a single shred, the shreduling algorithm is illustrated in Figure 7. A shred is initially shreduled to execute immediately—further shreduling beyond this point is left to the shred. The shreduler checks to see if the shred is shreduled to wake up at the current time (`now`). If so, the shred resumes execution in the interpreter until it schedules itself for some future time, say 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 presently wake up at `now`, 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 one sample, 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.

Two points should be noted here. First, it is possible for a shred to misbehave and never advance time or, in the real-time case, to perform enough computation to delay audio. The halting problem (Turing 1937; Sipser 2005) informs us that the VM cannot hope to detect this reliably. Nevertheless, it is possible for the user to identify this situation and manually remove a hanging shred from the interpreter. Second, the given algorithm is designed for causal operations in which time can only be advanced towards the future. (Advancing time by zero duration is identical to yielding without advancing time—relinquishing the VM to other shreds, if any, that have been shreduled to run at the current point in time.)

For multiple shreds, the mechanism behaves in a similar manner, except that the shreduler has a waiting list of shreds, sorted by requested wake-up time. A more comprehensive concurrent shreduling algorithm is shown in Figure 8. Before the system-wide `now` is advanced to the next sample, all shreds waiting to run at the current time must be allowed to execute.

In addition, it is possible for a shred to advance time by any amount, even durations less than that of a sample. To support this, each shred keeps track of a shred-local `now`, which is close to the value of the system-wide `now`, but with some allowable fractional sample difference. This enables a shred to shredule itself ahead by an arbitrarily small increment. This value is compared against the system-wide `now` when determining when to wake up a shred. It is therefore possible for a shred to run any number of times before the system-wide `now` is advanced to the next sample.

Figure 8. Multi-shredded shreduling decision flow.



Time-Driven Audio Computation

Unit generators (and more recently, unit analyzers) are created, connected, disconnected, and controlled from code. The actual computation of the audio takes place separately, however, in the audio engine via the global unit-generator graph. When the shreduler decides that it is appropriate to compute the next sample and advance time, the audio engine is invoked. The global UGen 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` is asked to compute and return the next sample (which may involve first recursively requesting the output of upstream UGens). The system marks visited nodes so that each unit generator is computed exactly once for every time step; and the audio output is cached. The output value of each UGen is stored and can be recalled, enabling arbitrary (down to single-sample delay) feedback cycles in the graph. Visiting a node that has already been marked at the current time step terminates cyclic recursion.

The architectural components make up the ChucK runtime system and hide the complexities of keeping track of logical time, shreduling, and interleaving user computation with audio synthesis, while exposing control over time and concurrency in the language.

Assessment of Design

All together, the core language features and the runtime architecture make up the ChucK programming language. Programming in ChucK is imperative and strongly typed, and may feel like a combination of Java, C, and something else: ChucK's own style. The design presents a complete and unique system to program in an "in the moment" time-centric manner. It reconciles logical time with actual time, and precise timing with concurrent programming; it does so without loss of generality in specifying fine-granularity timing algorithms.

ChucK's synchrony hypothesis is a useful property to work with, for it provides a predictable temporal mapping between code and audio synthesis stream. In practice, this is feasible for real-time operation, as long as no section of code execution is sufficiently lengthy to introduce a break in the audio output. More rigorously, as long as a ChucK program can run at least as fast as its environment (i.e., the synthesized audio stream), it runs in real time. (In this context, "fast" is taken to mean sufficiently fast to compute each audio buffer within the allotted time so as to not introduce interruptions in the resulting audio output.) The determinism such assumptions provide leads to clearer specification and debugging of temporal relationships, with no intrinsic burden to real-time operation.

Although there are unique benefits in ChucK's approach, the system also has important tradeoffs and limitations. First, as discussed earlier, there is no foolproof way to prevent misbehaving shreds from temporarily hanging the virtual machine—a user will not know to take action until after the audio has been interrupted. Second, ChucK was designed more for expressiveness than for performance throughput. The synthesis graph is traversed for each individual sample, which incurs significant overhead. Although reasonable effort was put into optimizing the existing architecture, such a sample-synchronous audio synthesis system is inherently subject to performance penalties from additional overhead dealing with each sample (e.g., extra function calls) and sacrifices the performance benefits of block processing (e.g., via certain compiler optimizations and instruction pipelining). Experiments that use ChucK's timing information to adaptively perform block-based processing have yielded promising, if preliminary, results. This is a subject of ongoing research.

Contributions

The core features and architecture of ChucK support a different way of thinking about audio programming and the design of synthesis languages. In summary, we discuss several useful properties that ChucK affords a programmer.

Temporal Determinism and Audio Programming

The ChucK programmer always codes in "suspended animation." This strongly timed property guarantees that time in ChucK does not change unless the programmer explicitly advances it. The value of `now` can remain constant for an arbitrarily long block of code, which has the programmatic benefits of (1) guaranteeing a deterministic timing structure to use and with which to reason about the system and (2) providing the programmer with a simple and natural mechanism of timing control. The deterministic nature of timing in ChucK also ensures that the program will flow identically across different executions and machines, free from the underlying hardware timing (processor, memory, bus, etc.) and from nondeterministic scheduling delays in the operating system kernel scheduler. The programmer is responsible for "keeping up with time" (i.e., specifying when to "step out" of suspended animation to advance time).

ChucK programs naturally have a strong sense of order regarding time. The timing mechanism guarantees that code appearing before time advancement operations will always evaluate beforehand, and those that appear after will evaluate only after the timing or synchronization operation is fulfilled. In other words, the blocks of code between operations that advance time are atomic; statements in each block are considered to take place at the same logical time instant. This semantic can lead to programs that are significantly easier to specify, debug, and reason about.

Furthermore, the timing mechanism allows feedback loops with single-sample delay, enabling clear representation of signal-processing networks, such as the classic Karplus-Strong plucked-string physical model (see Figure 9). It is straightforward to implement (for example) various extensions of the model (Jaffe and Smith 1983; Steiglitz 1996), as well as a number of other physical models, directly in the language—and to hear and test them on the spot, making ChucK an ideal prototyping and teaching tool.

The same time mechanism can be used both for fine-grained synthesis and for higher-level "musical" or compositional timing. ChucK makes no distinction in its intended use, and it is left up to the programmer to choose the appropriate timescales to work with: synthesis (samples to milliseconds), "note" level (hundreds of milliseconds or longer), structural (seconds or minutes), or "macro-structural" (hours, days, or even years). All timescales and control strategies are unified under the same timing mechanism.

Figure 9. Native Karplus-Strong plucked-string physical model in ChucK. The precise and flexible timing mechanism allows complex timing to be represented directly in the language, without the need to depend on opaque unit generators "outside" the language; this aspect is suitable for rapid prototyping and for clearly delineating complex synthesis algorithms.

Figure 10. Block diagram for the Karplus-Strong plucked-string model, directly implemented in ChucK, as shown in Figure 9.

```
// Feedforward elements
Noise imp => Delay delay => dac;
// Feedback (single-sample)
delay => Gain attenuate => OneZero lowpass => delay;

// Our radius (comb filter)
.99999 => float R;
// Our delay order
500 => float L;
// Set delay
L::samp => delay.delay;
// Set dissipation factor
Math.pow( R, L ) => attenuate.gain;
// Place zero in OneZero filter (for gentle lowpass)
-1 => lowpass.zero;

// Fire excitation...
1 => imp.gain;
// ... for one delay round trip
L::samp => now;
// Cease fire
0 => imp.gain;

// Advance time until desired signal level remains
(Math.log(.0001) / Math.log(R))::samp => now;
```

Figure 9

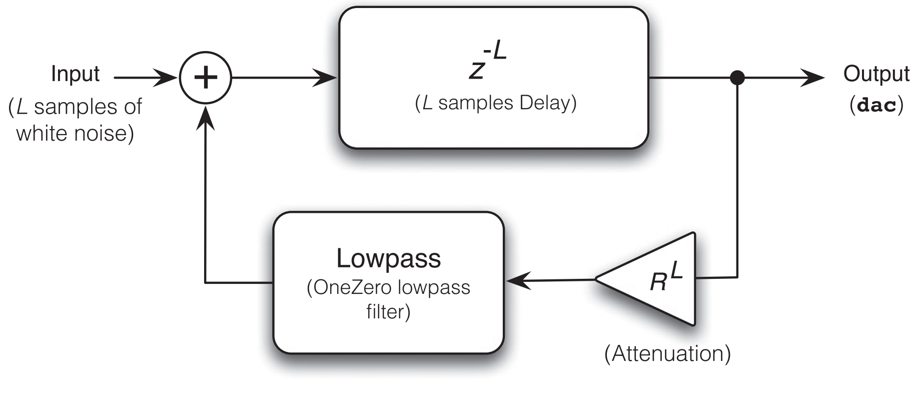


Figure 10

Case Study: Native Karplus-Strong Model

This case study shows how to natively construct a Karplus-Strong plucked-string model (Karplus and Strong 1983) in ChucK. Note that this is a straightforward translation of the signal block diagram (see Figure 10) using only elementary building blocks (Noise, OneZero filter, Delay). The key insight here is that ChucK's sample-synchronous model naturally allows single-sample feedback, which is an essential element of this physical model.

Notes on Control Rates

The manner with which a shred advances its way through time can be naturally interpreted as a kind of control rate (i.e., for asserting control over audio synthesis). Because the amount of time to advance at each point is determined by the programmer, the control rate can be as rapid (e.g., approaching or same as the sample rate) and variable (e.g., milliseconds, minutes, days, or even weeks) as desired for the task at hand. Additionally, the control rate can vary dynamically with time, because the programmer can compute or look up the value of each time increment. Finally, the possibility of a dynamic, arbitrary control rate is enhanced by ChucK's concurrency model, which allows multiple, independent control flows to compute in parallel.

Synchronous Concurrency

Shreds naturally separate each set of independent tasks into concurrent entities running at their own control rates. For example, there might be many different streams of audio samples being generated at multiple control rates; MIDI and OSC messages might arrive periodically (e.g., on the order of milliseconds) from a variety of sources, controlling parameters in the synthesis. Concurrently, packets may arrive over the network while an array of HID mice and joysticks may be generating control data. At the same time, higher-level musical processes may be computing at yet another timescale. In ChucK, it is straightforward to design and incrementally develop such a system via shreds, independently managing their local timings (refer to Figure 3 for an example of multiple concurrent shreds each operating at a different control rate).

In ChucK, timing and synchronization are the core elements in the larger machinery of strongly timed programming. On the one hand, explicit timing generates implicit synchronization (i.e., advancing time provides the information needed by the ChucK VM to precisely synchronize shreds) and, on the other hand, explicit synchronization generates implicit timing (e.g., explicitly synchronizing on events allows time to advance meanwhile).

An extremely useful property here is that whereas shreds are interleaved in time and therefore appear to be concurrent, code executes without preemption between advancements in time (e.g., via `now`). Thus these code blocks behave as critical sections or atomic transactions. This naturally precludes many common issues that arise in preemptive thread programming (e.g., race conditions) and does so without need for additional synchronization mechanisms such as critical sections or mutexes.

Concluding Remarks

ChucK's language design affords a number of useful properties, and promotes a different way of thinking about audio programming. As a software system and creative tool, it has been applied to, and has coevolved with, many artistic and design endeavors. We conclude by examining various experiments and applications of ChucK, and comment on its continued evolution.

Applications

The ChucK programming language has found a variety of applications in composition, performance, computer music research, interaction design, and sound design (see Wang 2014b)—and continues to be applied to new areas of use. Its rapid prototyping, in-the-moment mentality also helped to spur experimentation with live coding as a performance practice, leading to audiovisual development and performance systems such as the Audicle, and eventually leading to the more prominent ChucK integrated development environment, miniAudicle (Salazar, Wang, and Cook 2006). A strongly timed system combining audio synthesis and analysis was designed in 2007 (Wang, Fiebrink, and Cook 2007). ChucK has served as experimental platform for on-the-fly machine learning for real-time music information retrieval prototyping (Fiebrink, Wang, and Cook 2008; Fiebrink 2011). ChucK also served as the sound engine (and a rapid-prototyping tool during development) of the mobile app Ocarina (Wang 2014a), which transformed the first generation of app-based mobile phones (e.g., the iPhone) into an expressive musical instrument with a social dimension. Since 2008, more than 10 million users of Ocarina have blown into their mobile phones and (unwittingly) used a mobile musical instrument powered by ChucK.

Laptop Orchestras and Teaching

Many opportunities have arisen to use and evaluate ChucK as a pedagogical tool for sound synthesis, programming, computer-mediated instrument design, and live performance. One of the largest and ongoing evaluations has been taking place in the context of laptop orchestras (Trueman et al. 2006; Fiebrink, Wang, and Cook 2007; Trueman 2007; Smallwood et al. 2008; Wang et al. 2009), as ChucK mirrored the development of the first laptop orchestra at Princeton (the Princeton Laptop Orchestra, known as PLOrk, in 2005) and subsequently at Stanford (Stanford Laptop Orchestra, SLOrk, in 2008). One of the major research findings is an approach to effectively teach programming through music-making, and vice versa (Wang et al. 2008). In the decade since, laptop orchestras have proliferated. At the time of this writing, there are more than 75 laptop performance ensembles worldwide (using a variety of software environments, including ChucK). These include the Carnegie Mellon Laptop Orchestra (Dannenberg et al. 2007); L2Ork (Bukvic et al. 2010); the Laptop Orchestra of Louisiana (http://laptoporchestrala.wordpress.com), which created a performance called "In ChucK," a reimagining of Terry Riley's "In C"; and many others. ChucK has also been extensively integrated into the music technology curriculum in many programs worldwide, including art schools such as the California Institute of the Arts, through its use in the Machine Orchestra (Kapur et al. 2011). The latter spawned both a book teaching programming for musicians and digital artists (Kapur et al. 2015) as well as a massively open online course of the same name.

Ongoing and Future Work

ChucK research and development began in 2002 and, at the time of this writing, is continuing more intensively than ever. Active areas of research include various mechanisms to extend ChucK (Salazar and Wang 2012), both external to the language (e.g., ChuG-ins) and internal (e.g., ChUGens and Chubgraphs). Although ChucK has internally powered millions of instances of mobile music apps, recent research has only begun to explore the use of mobile devices (e.g., the iPad) as programming platforms for audio (Salazar and Wang 2014). Another research direction that is underway examines graphics programming integrated with audio in ChucK. To be able to specify real-time graphics, image, and video processing in the same strongly timed audio framework seems enticing and promising. Finally, as with live coding and the Audicle, ChucK is both about audio programming and the aesthetics and human mechanics of audio programming. Along this vein, it would be exciting to explore new social dimensions of collaborative audio programming, to investigate both the human–computer and human–human interaction in such settings.

Conclusion

In this article, we presented the ChucK programming language, its ideas, core language features, and the various properties associated with the language. Additionally, we examined several applications of the language, as well as its evaluation as a programming tool and pedagogical vehicle for teaching programming and music creation in tandem. Although much has been investigated, more remains to be discovered and explored in the realm of strongly timed music and audio programming—this investigation will continue.

Coding can be an expressive, creative, and ultimately satisfying process. It should aim to feel empowering (and fun) to write code—and hopefully not cumbersome to read, debug, or interpret it. A programming language cannot help but shape the way we think about solving particular problems.

It is our hope that ChucK provides a unique way of working with and thinking about time—and its variety of interactions with sound and music.

ChucK is open-source and freely available online at chuck.stanford.edu.

Acknowledgments

A great number of people supported and contributed to the development, design, and evaluation of ChucK. Aside from the authors, these include Ajay Kapur, Dan Trueman, Rebecca Fiebrink, Philip Davidson, Ari Lazier, Ananya Misra, Adam Tindale, Kassen Oud, and Chris Chafe, as well as other people from the Princeton Soundlab, CCRMA, the Stanford music department, and the ChucK community.

References

Aaron, S., and A. F. Blackwell. 2013. "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages." In *Proceedings of the ACM SIGPLAN Workshop on Functional Art, Music, Modeling, and Design*, pp. 35–46.

Anderson, D., and R. Kuivila. 1991. "Formula: A Programming Language for Expressive Computer Music." *IEEE Computer* 24(7):12–21.

Anderson, T., et al. 1992. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." *ACM Transactions on Computer Systems* 10(1):53–79.

Berry, G., and G. Gonthier. 1992. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation." *Science of Computer Programming* 19(2):87–152.

Brandt, E. 2002. "Temporal Type Constructors for Computer Music Programming." PhD dissertation, Carnegie Mellon University.

Bukvic, I. I., et al. 2010. "Introducing L2Ork: Linux Laptop Orchestra." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 170–173.

Burk, P., L. Polansky, and D. Rosenboom. 1990. "HMSL (Hierarchical Music Specification Language): A Theoretical Overview." *Perspectives of New Music* 28(2):136–178.

Caspi, P., et al. 1987. "LUSTRE: A Declarative Language for Programming Synchronous Systems." In *Annual Symposium on Principles of Programming Languages*, pp. 178–188.

Collins, N., et al. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8(3):321–330.

Cook, P. R., and G. Scavone. 1999. "The Synthesis Toolkit (STK)." In *Proceedings of the International Computer Music Conference*, pp. 164–166.

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

Dannenberg, R., et al. 2007. "The Carnegie Mellon Laptop Orchestra." In *Proceedings of the International Computer Music Conference,* vol. 2, pp. 340–343.

Fiebrink, R. 2011. "Real-Time Human Interaction with Supervised Learning Algorithms for Music Composition and Performance." PhD dissertation, Princeton University.

Fiebrink, R., G. Wang, and P. R. Cook. 2007. "Don't Forget the Laptop: Using Native Input Capabilities for Expressive Control." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 164–167.

Fiebrink, R., G. Wang, and P. R. Cook. 2008. "Support for MIR Prototyping and Real-Time Applications of the ChucK Programming Language." In *Proceedings of the International Conference on Music Information Retrieval*, pp. 153–158.

Halbwachs, N. 1993. *Synchronous Programming of Reactive Systems*. Alphen aan den Rijn, Netherlands: Kluwer.

Hoare, C. A. R. 1978. "Communicating Sequential Processes." *Communications of the ACM* 21(8):666–677.

Hudak, P., et al. 1996. "Haskore Music Notation: An Algebra of Music." *Journal of Functional Programming* 6(3):465–483.

Jaffe, D., and J. O. Smith. 1983. "Extensions of the Karplus-Strong Plucked String Algorithm." *Computer Music Journal* 7(2):56–69.

Kapur, A., et al. 2011. "The Machine Orchestra: An Ensemble of Human Laptop Performers and Robotic Musical Instruments." *Computer Music Journal* 35(4):49–63.

Kapur, A., et al. 2015. *Programming for Musicians and Digital Artists: Making Music with ChucK*. Shelter Island, New York: Manning Press.

Karplus, K., and A. Strong. 1983. "Digital Synthesis of Plucked String and Drum Timbres." *Computer Music Journal* 7(2):43–55.

Lansky, P. 1990. "CMIX." Release Notes and Manuals. Princeton, New Jersey: Department of Music, Princeton University.

Le Guernic, P., et al. 1985. “SIGNAL: A Data Flow Oriented Language for Signal Processing.” *IEEE Transactions on Acoustics, Speech and Signal Processing* 34(2):362–374.

Mathews, M. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: MIT Press.

McCartney, J. 2002. “Rethinking the Computer Music Programming Language: SuperCollider.” *Computer Music Journal* 26(4):61–68.

Pope, S. T. 1993. “Machine Tongues XV: Three Packages for Software Sound Synthesis.” *Computer Music Journal* 17(2):23–54.

Puckette, M. 1991. “Combining Event and Signal Processing in the Max Graphical Programming Environment.” *Computer Music Journal* 15(3):68–77.

Puckette, M. 1996. “Pure Data.” In *Proceedings of the International Computer Music Conference*, pp. 224–227.

Roberts, C., et al. 2013. “Gibber: Abstractions for Creative Multimedia Programming.” In *Proceedings of the ACM International Conference on Multimedia*, pp. 67–76.

Salazar, S., and G. Wang. 2012. “Chugens, Chubgraphs, and Chugins: 3 Tiers for Extending ChucK.” In *Proceedings of the International Computer Music Conference*, pp. 60–63.

Salazar, S., and G. Wang. 2014. “MiniAudicle for iPad: Touchscreen-Based Music Software Programming.” In *Proceedings of the International Computer Music Conference*, pp. 686–691.

Salazar, S., G. Wang, and P. R. Cook. 2006. “miniAudicle and ChucK Shell: New Interfaces for ChucK Development and Performance.” In *Proceedings of the International Computer Music Conference*, pp. 64–66.

Schottstaedt, B. 1994. “Machine Tongues XVII: CLM: Music V Meets Common Lisp.” *Computer Music Journal* 18(2):3–37.

Sipser, M. 2005. *Introduction to the Theory of Computation*. Boston, Massachusetts: Cengage.

Smallwood, S., et al. 2008. “Composing for Laptop Orchestra.” *Computer Music Journal* 32(1):9–25.

Sorensen, A. 2005. "Impromptu: An Interactive Programming Environment for Composition and Performance." In *Proceedings of the Australasian Computer Music Conference*, pp. 149–153.

Sorensen, A., and H. Gardner. 2010. “Programming with Time: Cyber-Physical Programming with Impromptu.” In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 832–834.

Steiglitz, K. 1996. *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music*. Upper Saddle River, New Jersey: Prentice Hall.

Touretzky, D. S. 1984. *LISP: A Gentle Introduction to Symbolic Computation*. New York: Harper and Row.

Trueman, D. 2007. “Why a Laptop Orchestra?” *Organised Sound* 12(2):171–179.

Trueman, D., et al. 2006. “PLOrk: Princeton Laptop Orchestra, Year 1.” In *Proceedings of the International Computer Music Conference*, pp. 443–450.

Turing, A. 1937. “On Computer Numbers, with an Application to the Entscheidungsproblem.” *Proceedings of the London Mathematical Society*, series 2 42(1):230–265.

Vercoe, B. 1986. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. Cambridge, Massachusetts: MIT Media Lab.

Wang, G. 2008. “The ChucK Audio Programming Language: A Strongly-Timed and On-the-Fly Environ/Mentality.” PhD dissertation, Princeton University.

Wang, G. 2014a. “Ocarina: Designing the iPhone’s Magic Flute.” *Computer Music Journal* 38(2):8–21.

Wang, G. 2014b. *The DIY Orchestra of the Future.* TED talk. Available online at www.ted.com/talks/ge_wang_the_diy_orchestra_of_the_future. Accessed September 2015.

Wang, G., and P. R. Cook. 2004a. “On-the-Fly Programming: Using Code as an Expressive Musical Instrument.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 153–160.

Wang, G., and P. R. Cook. 2004b. “Audicle: A Context-Sensitive, On-the-Fly Audio Programming Environ/Mentality.” In *Proceedings of the International Computer Music Conference*, pp. 256–263.

Wang, G., R. Fiebrink, and P. R. Cook. 2007. “Combining Analysis and Synthesis in the ChucK Programming Language.” In *Proceedings of the International Computer Music Conference*, pp. 35–42.

Wang, G., et al. 2008. “The Laptop Orchestra as Classroom.” *Computer Music Journal* 32(1):26–37.

Wang, G., et al. 2009. “Stanford Laptop Orchestra (SLOrk).” In *Proceedings of the International Computer Music Conference*, pp. 505–508.

Wright, M., and A. Freed. 1997. “Open Sound Control: A New Protocol for Communicating with Sound Synthesizers.” In *Proceedings of the International Computer Music Conference*, pp. 101–104.