ChucK: A Concurrent, On-the-fly, Audio Programming Language

Ge Wang and †Perry R. Cook

Computer Science Department, †(also Music)
Princeton University

{gewang,prc}@cs.princeton.edu

Abstract

ChucK is a new audio programming language for real-time synthesis, composition, and performance, which runs on commodity operating systems. ChucK natively supports concurrency, multiple, simultaneous, dynamic control rates, and the ability to add, remove, and modify code, on-the-fly, while the program is running, without stopping or restarting. It offers composers and performers a powerful and flexible programming tool for building and experimenting with complex audio synthesis programs, and real-time interactive control.

1. Ideas in ChucK

In this paper we present four key ideas that form the foundation of ChucK. The goal is to design a natural audio programming language (1) to concurrently and accurately represent complex audio synthesis, (2) to enable fine-grain, flexible control over time, (3) to provide the capability to operate on multiple, dynamic and simultaneous control rates, and (4) to make possible an *on-the-fly* style of programming. ChucK runs on commodity operating systems (Linux, Windows, Solaris, MacOS).

Four major ideas form the basis of ChucK.

- A unifying, massively overloaded operator.
- A precise timing model that is capable of true concurrency and arbitrarily fine granularity. The language semantic supports multiple, simultaneous, and dynamic control rates, and naturally amortizes operations over time.
- Native language support of arbitrarily many input/output sources, MIDI, network, serial, USB, graphics, and any input/output device you can connect to the computer.
- *On-the-fly programming* enables dynamically modifiable programs for performance and experimentation.

We present ChucK with respect to these four major facets. We provide some informal 'rules' of ChucK, each of which embodies some key aspect of the programming language. In *Section 2*, we look in detail at the ChucK operator. In *Section 3*, we present the ChucK timing model, introduce the concept of *shreds* and concurrency in ChucK, and demonstrate multiple and simultaneous control paths and control rates. *Section 4* presents an overview of the integration of many types of input/output related devices and operations into the language. *Section 5* presents a special aspect of ChucK: its ability to be programmed *on-the-fly* during run-time. ChucK is implemented as a virtual machine running with a special run-time compiler with low-level audio engine. *Section 6* takes a step back and reasons about the performance benefits/drawbacks of ChucK.

2. The ChucK Operator

ChucK is a *strongly-typed*, imperative programming language. Its syntax and semantics are governed by a flexible type system. ChucK includes standard features (arithmetic, bit-wise, memory operations, etc…) and control flow mechanisms (`if`, `for`, `while`, `switch`, `goto`, `break`, `continue`, etc…) common to most modern imperative programming languages. But the heart of ChucK's syntax is based around the ChucK operator (written as **`=>`**).

1st rule of ChucK: *make use of the left-to-right ChucK operator.* **`=>`** originates from the slang term "chuck", meaning to throw an entity into or at another entity. The language uses this notion to help express sequential operations and data flow.

```
(440 => osc) => env => filt => audio[0];
```

The above code fragment constructs a simple synthesis instrument using a series of unit generators (Mathews 1969) (their declarations are omitted for the moment): an oscillator, an envelope, a filter, and, finally, audio channel 0. Notice that the single line captures the flow of the signals from left to right - the same order as we read and type.

A ChucK statement can be composed of any appropriate types of objects (including user-defined), unit generators, operations, values, and variables. The semantic of the statement depends on the types of the objects, and the overloading of the ChucK operator on those types.

The main ideas behind the ChucK operator are (1) it locally captures the left-to-right nature of program flow (with the exception of nested ChucK statements), representing the order of sequential operations more faithfully, (2) it can be chained to any arbitrary length, and (3) it reacts, or behaves, differently depending on the types of the objects being chucked. **=>**'s behavior is defined/altered through overloading using the ChucK type system.

The basic ChucK operation takes place between two entities, the ChucKer and the ChucKee:

```
x => y
```

The operation performed depends on the types of x (ChucKer) and of y (ChucKee), the combination of which maps to a particular action that is overloaded on the ChucK operator. For example, it might assign the value to a variable, or connect/disconnect one unit generator to/from another, or anything depending on the particular overloading. For example, the notion of *behavioral abstraction* in Nyquist (Dannenberg 1997) can be realized through this type of overloading **=>**.

The following are some sample ChucK statements, where the identifiers x, y etc. can be any *well-typed* entities.

Binary ChucK: the simplest ChucK statement, chucking object/value x to object/value y.

```
x => y;
```

Chain ChucK: ChucK statements can be a sequence of ChucK operations of arbitrary length. The operations are performed left to right, in exactly the same order as written.

```
w => x => y => z;
```

Nested ChucK: evaluation of ChucK expressions gives *local* precedence to parenthesis. In the statement below, the operation `v=>w` is evaluated, then `x=>y`, and the result of `v=>w` is chucked to the result of `x=>y`. Finally, that result is chucked to z.

```
v => w => ( x => y ) => z;
```

Cross ChucK: *cross-chucking* allows one or more ChucKer objects/values to be chucked to more than one ChucKee. In the statement below, the object/value w is chucked to x, then y, and then z. (This can also be achieved by a more verbose sequence of three ChucK statements.)

```
w => ( x, y, z );
```

The ChucK programming language has no assignment operator (=), but unifies this operation under the ChucK operator, as seen here:

```
5 => int x;
```

With the ChucK operator, we represent any sequence of operations in a left-to-right manner. As shown below, a piece of code is chucked over TCP to a remote host (also running ChucK), waits for a message, and prints out the result.

```
code_seg => tcp(140.180.141.103:8888)
         => local_receiver => stdout;
```

The ChucK operator can also naturally synchronize on events and objects. Synchronization primitives such as semaphores, condition variables, and monitors (Lampson 1980) can be inserted into ChucK chains as clear points of synchronization.

```
code_seg => mutex => machine;

button[0] => play_note();
```

In the next sections, we will look at the issues of timing and concurrency, input/output, and *on-the-fly programming*, each of which employs some aspect of the ChucK operator.

3. Time, Shreds, Rates

The notions of time, duration, synchronization, control rates, and simultaneity are captured automatically by ChucK's timing mechanism. ChucK allows the programmer, composer, and performer to write truly concurrent code using the framework of the timing semantic. There is no fixed notion of *control rate* – the control rate is a natural product of using the timing constructs. The timing semantic, along with concurrency leads to the ability to dynamically change control rate, as well as have many different control rates.

3.1 Time and Duration

The **`now`** keyword (of type **`time`**) is defined to always hold the exact, current ChucK time. The keyword **`dur`** refers to the duration between points in time. These allow the programmer to do precise arithmetic with time and duration as we see in the following examples.

Each duration value has a unit attached to it and is declared in the following way:

```
0.01:second => dur midi_rate;
```

In the above example, we chuck a value of `0.01:`**`second`** to a newly declared variable of type **`dur`**, called `midi_rate`. The "built-in" units are **`samp`** (duration of one sample), **`ms`**, **`second`**, **`minute`**, **`hour`**, **`day`**, **`week`**. We can also use any variable of type **`dur`** as a unit to inductively construct other durations:

```
0.7:second => dur quarter;
4:<quarter> => dur whole;
```

The above statement constructs a duration value of `4:<quarter>` (`quarter` defined in the previous statement) and chucks a *relationship* (symbolized by the angle brackets) to a new variable called `whole`. In this case, chucking a new **`dur`** value to `quarter` will automatically change `whole`. With this system, we can easily define and use any duration greater than or equal to the duration of a single sample.

Time is defined in terms of existing time values. We can perform arithmetic using time and duration to obtain new time values. Here we calculate the time value for `10:`**`second after now`**, and store this value in a new variable called `later`.

```
10:second after now => time later;
```

We can do comparisons of time values:

```
while( now < later )
{
    now => format_sec => stdout;
    1:second => now;
}
```

The above code prints out the current values of **`now`**, once every second (in real-time), for 10 seconds. It not only demonstrates the comparison of time values, but also shows several key features of the language, which are summarized by the next two "rules" of ChucK:

2nd rule of ChucK: *you always code in suspended animation*. This rule 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 reason about the system and (2) giving a simple and natural mechanism of complete timing control to the programmer. 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) and non-deterministic scheduling delays in the kernel scheduler.

In the previous code segment, the value of **`now`** is guaranteed to remain constant from the evaluation of the while condition through the first statement in the loop body. The last statement in the while loop has a special semantic, in that it updates the value of **`now`**, by one second, with the side effect of *blocking the process for that amount of time*. Once control is returned from blocking, **`now`** holds the updated value.

3rd rule of ChucK: *you are responsible for keeping up with time*. The programmer is given the responsibility for deciding when to "step out" of suspended animation and advance time. He/she can do it in one of two ways. 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 above. This allows for a natural programming approach that embeds the timing control directly in the language, giving the programmer the ability to perform computations at arbitrary points in time, and to "move forward" in ChucK time in a precise manner.

The second method to advance time in ChucK is by waiting on some event(s). These could be synchronization mechanisms (such as mutexes, semaphores, and/or condition variables), input devices, or any asynchronous event(s) such as MIDI input or packets arriving over TCP or UDP. Execution will resume when the synchronization condition is fulfilled. *Wait-on-event* is similar in spirit to *chuck-to-now*, except events have no pre-computable time of arrival.

Timing and duration are traditionally conveyed as parameters to entities (functions and objects that are internally scheduled) in existing sound synthesis languages, such as Nyquist (Dannenberg 1997) and SuperCollider (McCartney 1996). ChucK also allows the programmer to manage the timing of *computation* itself. Furthermore, the timing mechanism allows operations to be naturally amortized over time. An envelope, for example, can be generated dynamically as time moves on, in addition to being declared and calculated entirely up-front. The following sample sets up a chain of unit generators, "plays" it for 2 seconds, and then dynamically changes the envelope value:

```
osc => filt => (1.0 => env) => audio[0];
2:second => now;

until( env--(0.02) < min )
    1:ms => now;
```

In line 1, unit generators (declared elsewhere) are connected in the desired order. Note this line only *connects* the unit generators, and does not start generating samples until time is advanced, which happens in line 2. This statement advances time by two seconds, and "allows" the unit generators to compute and play samples for that duration. After exactly the number of samples in 2 seconds, control is returned and we enter the *until loop*, which decreases the value of `env` by 0.02 every millisecond, gradually silencing the unit generator chain.

There is another nice property afforded by the ChucK timing mechanism. Statements that appear in code before the time advancement are guaranteed to evaluate beforehand (desirable side-effects may remain, such as connections of unit generators), and those that appear after the time advancement will evaluate only after the timing or synchronization operation is fulfilled. This method, like the ChucK operator, encourages a strong sense of order in the program.

3.2 Dynamic Control Rate

The amount of time advancement *is* the control rate in ChucK. Since the amount of time to advance at each point is determined by the programmer, the control rate can be (1) as high (same as sample rate) or as low (any multiple of a sample duration, such as milliseconds, days, or even months) as the application desires, and (2) dynamically varying with time - since the programmer can compute or lookup the value of each time advancement. Additionally, the power of this dynamic, arbitrary control rate is greatly extended by ChucK's concurrency model, as we will see in *Sections 3.3* and *3.4*.

It is possible in ChucK to calculate each sample completely from within the language (though low-level built-in and add-in ChucK modules may be more suitable for such low-level tasks). All external events, such as MIDI, input devices, and other asynchronous events, are internally synchronized at a coarser granularity proportional to a tunable latency, determined by the underlying hardware and OS. Program logic, of course, can be placed at any granularity relative to the audio. Thus, the same ChucK timing mechanism can be used to build low-level instruments, as well as high-level compositional elements.

The practice of enabling the programmer to operate on an arbitrarily fine granularity is derived, in practice, from the Synthesis Tool Kit (STK) (Cook and Scavone 1999), which exposes a manageable programming interface for efficient single sample operations, with additional levels of internal buffering. ChucK builds on this notion to support sample-level computations as well as computations at arbitrarily large intervals.

Thus far, we have only discussed programming ChucK using one path of execution. ChucK is a concurrent language, which allows multiple independent paths of computation to be executed in parallel. The flexibility and power of the timing mechanism is greatly extended by ChucK's concurrency model, which allows multiple, precisely timed paths of computation.

3.3 Programming with Shreds

The ChucK programming language natively enables the *chuckist* to write code that operates either in series or in parallel via ChucK's concurrency model. It is also this mechanism that provides fine-grain, multiple, and simultaneous control rates. We now introduce a primitive called *shreds*. A *shred*, much like a thread, is an independent, lightweight process, which operates concurrently and can share data with other *shreds*. However, unlike traditional threads, whose execution is interleaved in a non-deterministic 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 same timing mechanism.

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

- A ChucK *shred* cannot be preempted by another *shred*. (Preemptive threads are also available in ChucK, but are not discussed here.) 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.

- A ChucK *shred* must voluntarily relinquish the processor for other shreds to run (In this they are like non-preemptive threads). But it does not do so with `yield()`. *Shreds*, by design, directly use ChucK's timing mechanism: when a *shred* advances time or waits for an event, it is, in effect, *shreduled* by the *shreduler* (which interacts with the audio engine), and relinquishes the processor. This is powerful in that it can naturally synchronize *shreds* to each other by time, without using any traditional synchronization primitives.

- ChucK *shreds* are implemented completely as user-level primitives. The entire virtual machine runs in user-space. User-level parallelism has significant performance benefits over kernel threads (Anderson et al. 1992), finding that "even fine-grain processes can achieve good performance if the cost of creation and managing parallelism is low." Indeed, ChucK *shreds* are lightweight - each only contains minimal state. The cost of context switching between ChucK *shreds* is also very low since no kernel interaction is required. Furthermore, a user-level *shreduler* is more easily modifiable.

An advantage of the *shred* approach is that the programmer has complete control over timing and the interaction of *shreds*. We gain the performance advantages from user-level parallelism. Furthermore, real-time scheduling optimizations (Dannenberg 1988) can be readily implemented by the *shreduler* without any kernel modifications. One potential drawback is that a single *shred* could hang the program if it fails to relinquish the processor. However, there are ways to alleviate this drawback:

- Any hanging *shreds* can easily by identified by the ChucK Virtual Machine, as the currently running thread, and the ChucK timing semantic makes it easy to locate and correct such issues for the programmer. *On-the-fly programming* allows for hanging *shreds* to be removed/corrected during run-time without stopping or restarting the system. We will see how this is accomplished in *Section 5*.

- It is possible to group *shreds* into separate, autonomous sets, which operate independently. This is analogous to the notion of *zones* in the Aura System (Dannenberg and Brandt 1996). Aura separates synchronous sound objects into asynchronous zones. Objects in a single zone cannot preempt each other, while one zone can preempt another zone on demand. Similarly, ChucK *shreds* behave deterministically within a set, while one set (given a higher priority) can preempt another. ChucK extends the notion of *zones* by grouping *shreds* (instead of objects), which are fully programmable and interactive.

Multi-shredded programs, in the Turing Machine sense, are not more powerful than *single-shredded* programs. But they can make the task of managing concurrency and timing much easier (and more enjoyable), just as threads make concurrent programming manageable, and potentially increase throughput. In this sense, *shreds* are more powerful *programming constructs*. We argue that the flexibility of *shreds* to empower the programmer to do deterministic, precisely timed, concurrent audio programming significantly outweighs the potential drawbacks.

The following example (Figure 1) shows the code for three *shreds*, all of which can be run singly or in parallel. The first *shred* is generating a sine tone at control rate = sample rate. The second *shred* sends a MIDI *noteon* message every 80 milliseconds. The third *shred* prints the value of a sensor every minute.

```
0 => float t;
while( true )     ¦ while( true )     ¦ while( true )
{                 ¦ {                 ¦ {
  sin( t*FC )     ¦   midimsg( c1,    ¦   sensor[9]
    => lineout;   ¦     note_on,      ¦     => stdout;
  t+1 => t;       ¦     80, 96 )      ¦   1:minute
  1:samp          ¦     => midi;      ¦     => now;
    => now;       ¦   80:ms => now;   ¦ }
}                 ¦ }
```

Figure 1. Three concurrent *shreds*.

In order to realize *shreds*, the ChucK Virtual Machine implements a *shreduler*, which is responsible for *shreduling* the *shreds*, taking into account time advancement, duration, synchronization, and audio sample generation. The *shreduler*, along with the ChucK audio engine, clocks the entire ChucK Virtual Machine.

Most existing audio programming languages handle simultaneity as parameterized objects and function invocations. Timing and duration values are often passed as arguments to synthesis entities. SuperCollider, for example, (McCartney 1996, Pope 1997) takes a highly parameterized approach to dealing with simultaneity, with the ability to schedule asynchronous, simultaneous events, but lacks the ability to write semi-autonomous code. In Nyquist (Dannenberg 1997), a degree of simultaneity is achieved by the `sim` construct, followed by a set of synthesis entities to be rendered simultaneously. The JSyn (Burk 1998) library inherently supports concurrency under Java's threading model, but because of the nature of this threading model, it has no determinism in timing, nor a high level of granularity in inter-thread communication and scheduling. ChucK adds full concurrency, with the ability to naturally and precisely synchronize and schedule all *shreds* of execution.

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 1987) and other languages. 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 *shred.*

A ChucK program is completely deterministic in nature (aside from asynchronous input events) - there is never any preemptive background processing, nor any implicit scheduling. The order that *shreds* and the virtual machine subsystem executes are determined completely 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. The concurrency model also enables multiple *shreds* to run at arbitrary control rates.

3.4 Multiple Concurrent Control Rates

4th rule of ChucK: *make use of multiple control paths and control rates using shreds*.

Computer music synthesis and performance is most often the simultaneity of many parallel sequences of operations, potentially happening at many distinct rates. *Shreds* naturally separate each set of independent tasks into concurrent entities each running on its own control rate. For example, as shown in Figure 2, many different streams of audio samples are being generated; MIDI messages arrive at a rate of 10 ms per message from a variety of sources, which control parameters in the synthesis. Concurrently, datagram packets arrive over the network every 20 ms, while an array of mice and joysticks send serial data over USB.

Figure 2. Three chuckists in session.

ChucK imposes no boundaries on the timing structure of a program - it does not make any decision about control rate or timing but instead integrates this decision into the language semantics (which the programmer can easily control). This enables the programmer to create and simultaneously execute any number of *shreds* - each potentially running at a different control rate.

4. Input/Output Devices

ChucK natively supports communicating with, and synchronizing on an arbitrary number of input and output devices. This includes physical devices and controllers such as mice, keyboards, game controllers, and MIDI devices. The language provides support for MIDI input/output, serial communication, and networking - in particular TCP byte-streams, and UDP and IP datagrams, and high-level network music protocols such as Open Sound Control (Wright and Freed 1997).

The advantage of using ChucK for I/O operations comes naturally from the availability of the *multi-shredded*, precision-timed language mechanism, and from the characteristics of the ChucK operator. For example, a *shred* may want to synchronize on events, or wait for messages to arrive over MIDI, or perhaps UDP. This is accomplished by taking advantage of the chaining nature of the ChucK operator and the timing mechanism. For example, consider the following statement:

```
msg => some_event =>
    => udp(140.180.141.103:1500);
```

Here, the `msg` is chucked to synchronize on `some_event`, which could be a signal generated from another *shred*, the availability of a mutex, an input event, or simply a statement *advancing time*. After `some_event` is fulfilled, the statement continues. The synchronization mechanism can be used as a 'gate' to allow objects to be chucked after the fulfillment of events or some amount of time.

ChucK enables the programmer to easily and quickly write code to synchronize with and communicate with input devices. For that reason, ChucK can be an ideal tool to easily use or rapidly prototype new controllers, or to write network or synchronization code.

5. On-the-fly Programming

The ChucK programming language natively supports the ability to write, compile, and execute code while the program is running. In fact, an *on-the-fly*, “real-time” compiler is part of the ChucK Virtual Machine (see Figure 3).

The goal of *on-the-fly programming* is to allow the *chuckist* to actively modify the program they are running without having to stop, re-code, and restart. For example, performers could add modules to their synthesis or composition programs, or a new controller during a live performance. Similarly, composers can experiment with their programs on-line, modifying synthesis components, adding a new instrument, or changing compositional elements, without having to restart. In fact, coding, composing, and performing are identical in ChucK.

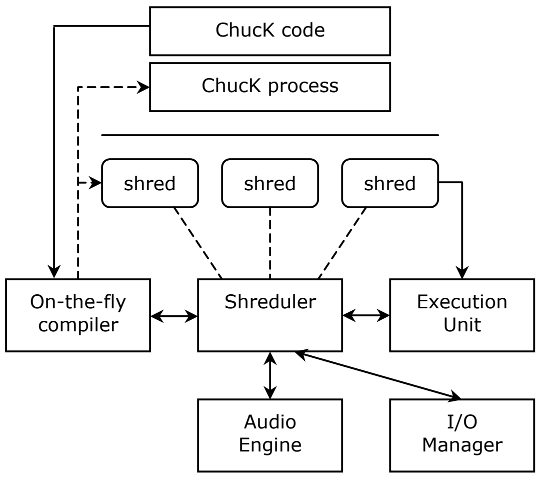

Figure 3. ChucK Virtual Machine runtime layout.

5th rule of ChucK: *use modularity of shreds to program on-the-fly*. ChucK is able to accomplish *on-the-fly programming* via the *shredding model*, **=>**, the virtual machine, and special objects in the language. The ChucK Virtual Machine has complete control over the *shreduling*, addition, and removal of *shreds*. New *shreds* can be programmed, compiled, and added into the *shreduler on-the-fly*. Existing *shreds* can be selected (via language constructs or the virtual machine), modified, or removed.

The language constructs that enable *on-the-fly programming* are based on the availability of the compiler (via the **`compiler`** object) and the virtual machine (via the **`machine`** object) in the language itself. New code can be added either externally, via interfacing the virtual machine from the OS or a shell, or from within the language. **`code`** is a native type, which allows programs to write and compile other programs *on-the-fly*.

The ChucK Virtual Machine can, at any moment, integrate a new *shred* into the overall ChucK runtime process. This practice in ChucK is called *assimilation*. The most basic way to do this is by writing new *shreds* and sending them via the shell to the currently running virtual machine process. New, augmented, or otherwise modified *shreds* or even an entire program can be seamlessly assimilated into the ChucK Virtual Machine, without stopping, restarting, or interrupting (audio) the virtual machine.

Additionally, *shreds* in the ChucK Virtual Machine can themselves read (from storage), generate, or accept *shreds* from remote hosts over the network and assimilate them. Again, the special **`machine`** object is used to interface with the virtual machine.

Shreds can also be removed (*dissimilated*) or suspended from the virtual machine - through similar facilities as those used for assimilation (from the shell via currently running code). The *dissimilated* or suspended *shreds* can then be modified and *re-assimilated* into the virtual machine.

One concern of *on-the-fly programming* is the efficiency of the programmer in constructing his/her program, from idea to algorithm to code, especially during time-critical situations such as performance or continuous composition session. Once again, the left-to-right **=>** lends itself to this task by allowing the programmer to input the sequence of operations in the general typing direction. Indeed, **=>** is a helpful entity in the language, for it conceptually and syntactically binds data-flow representation and lends itself naturally to timing and synchronization.

6. In Summary

6.1 Performance vs. Flexibility

The ChucK runtime system is implemented as a virtual machine (with audio engine and I/O manager implemented in C/C++) and a special virtual instruction set, with the ability to dynamically link-in compiled code from other languages. This virtual machine approach is in stark contrast with most real-time computer music programming languages and libraries. Given the performance impact, this is nothing to boast about, *except* that a pure virtual instruction set and VM allows for highly flexible and elegant (albeit slower) audio programming constructs at the language level. Furthermore, the language constructs provide a way to gauge and schedule resources and time in a way that the programmer can easily understand, manage, and debug.

Implementing a user-level virtual machine has enabled us ultimate flexibility in the language construct of *shreds* and *shreduling*. We are able to seamlessly, and with sample-width precision, write multi-shredded audio synthesis and composition programs that operate at multiple and simultaneous control rates.

The features of ChucK make it natural for a variety of applications. We present a few examples where various ChucK facilities can be useful.

Granular synthesis: this synthesis technique (Roads 1985) can be implemented directed in the language (without special-purpose unit generators or plug-in modules), by using one or more *shreds* and moving each along in small time intervals via the timing mechanism and performing the overlap/add operations and computations for the control data dynamically.

Collaborative composition/performance: The *on-the-fly* programming model easily allows for multiple *chuckists* to write *shreds* on different (possibly remote) hosts and to *assimilate* them via networking into a single Chuck Virtual Machine instance, *on-the-fly*. This model also facilitates the reverse process, allowing *shreds* to be distributed among hosts for real-time load balancing.

Real-time graphics: the multi-shredded model makes it straightforward to embed real-time graphics into audio programs. One or more *shreds* can be devoted to controlling 2D/3D rendering (using the same timing mechanism and at appropriate control rates) while sharing state with the rest of the program, making it easy to integrate audio and graphics with precise timing.

6.2 Conclusion

ChucK is not a loose collection of unrelated ideas, but a group of entities that work off of each other. The timing mechanism provides a precise and flexible method to work with time-based operations. *Shreds* empower programmers to write truly concurrent synthesis programs, while the timing semantic makes concurrency work correctly and efficiently without additional synchronization mechanisms. Together, *shreds* and timing allows simultaneity of multiple control rates. *Shreds* also offer a modular approach to make *on-the-fly programming* possible and manageable. The ChucK operator - a simple language construct - unifies the overall syntax and semantics of the language in a left-to-right manner, and provides natural synchronization points with events and input devices.

In conclusion, ChucK is a concurrent, on-the-fly language, built for real-time audio synthesis and performance. The ChucK operator encourages a more accurate language representation of sequential operations. *Shreds* allow for concurrency, as well as multiple and simultaneous control rates, each of which, using the timing mechanism, can be as fine as the audio sample rate, or as long as hours, days, even years. The ChucK timing mechanism provides a deterministic view and control of time, duration, and the notion of 'now'. It embeds the ability to advance time and synchronize on various events, and naturally synchronizes all *shreds*. ChucK's *on-the-fly programming* abilities enable programmers, composers, and performers to add, modify, and remove code as their programs are running, fundamentally enhancing the amount of real-time interaction in the process of audio programming.

ChucK is freely available from the authors at:

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

References

Anderson, T. E., B. N. Bershad, E. D. Lazowska, and Henry M. Levy. 1992. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." *ACM Transactions on Computer Systems*, 10(1):53-79.

Burk, P. 1998. "JSyn - A Real-time Synthesis API for Java." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 252-255.

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

Dannenberg, R B. 1988. "A Real Time Scheduler/Dispatcher." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 239-242.

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

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

Lampson, B. W. and D. D. Redell. 1980. "Experience with Process and Monitors in Mesa." *Communications of the ACM* 23(2):105-117.

Lansky, P. 1987. "CMIX" Program Documentation. Princeton, New Jersey: Princeton University. *http://silvertone.princeton.edu/winham/man/*

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

McCartney, J. 1996. "SuperCollider: A New Real-Time Synthesis Language." *In Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 257-258.

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

Pope, S. T. 1997. *Sound and Music Processing in SuperCollider*. University of California, Santa Barbara. *http://www.create.ucsb.edu/htmls/sc.book.html*

Roads, C. 1985. “Granular Synthesis of Sound.” In C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music*. Cambridge, Massachusetts: MIT Press. pp.145-159.

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.* International Computer Music Association, pp. 101-104.