On-the-fly Programming: Using Code as an Expressive Musical Instrument

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

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

ABSTRACT

On-the-fly programming is a style of programming in which the programmer/performer/composer augments and modifies the program while it is running, without stopping or restarting, in order to assert expressive, programmable control at runtime. Because of the fundamental powers of programming languages, we believe the technical and aesthetic aspects of on-the-fly programming are worth exploring.

In this paper, we present a formalized framework for on-the-fly programming, based on the ChucK synthesis language, which supports a truly concurrent audio programming model with sample-synchronous timing, and a highly on-the-fly style of programming. We first provide a well-defined notion of on-the-fly programming. We then address four fundamental issues that confront the on-the-fly programmer: timing, modularity, conciseness, and flexibility. Using the features and properties of ChucK, we show how it solves many of these issues. In this new model, we show that (1) concurrency provides natural modularity for on-the-fly programming, (2) the timing mechanism in ChucK guarantees on-the-fly precision and consistency, (3) the Chuck syntax improves conciseness, and (4) the overall system is a useful framework for exploring on-the-fly programming. Finally, we discuss the aesthetics of on-the-fly performance.

Keywords

On-the-fly programming, code as interface, concurrency, timing, synthesis, concurrent audio programming, synchronization, real-time, compiler, virtual machine.

1. INTRODUCTION

Due to their fundamental expressive power, programming languages and systems play a pivotal role in the composition, performance, and experimentation of computer audio and electro-acoustic music. For the most part, however, the design and writing of computer music programs have been limited to off-line development and preparation, leaving only the finished program to "go live". Thus, the gamut of runtime possibility is prescribed by the functionalities that are pre-determined and programmed ahead of time.

An *on-the-fly programmable system* provides the ability to write/modify, compile, execute new/existing code, and then integrate it into a program while it is running, with precise timing and synchronization. The goal of on-the-fly programming is to enable programmers/performers/composers to actively modify their programs on-line without having to stop, code, and restart. For example, performers could add/change modules in their synthesis or composition programs, or modify mappings to their controllers during a live performance. Similarly, composers can experiment with their programs on-line, modifying synthesis components, shaping or perfecting a sound, or changing compositional elements, without having to restart.

Figure 1. On-the-fly programmers in session**.**

The features of the programming tool inevitably shape both the means by which tasks are implemented as well as the end product. By bringing the power and expressiveness of the programming language into runtime, an on-the-fly programming system has the potential to fundamentally enhance the real-time interaction between the performer/composer and the systems they create and control. *Code becomes a real-time, expressive instrument*. We believe that such a potential is worth exploring.

In this work, we define on-the-fly programming and provide a formal programming model based on ChucK [12], leveraging its properties of timing and concurrency, as well as the ChucK virtual machine. In addition, we discuss an open on-the-fly programming aesthetic. The rest of this paper is organized as follows. *Section 2* defines on-the-fly programming and identifies some of the central issues that an on-the-fly programming system should address, and also discusses on-the-fly elements found in existing programming languages. *Section 3* provides an overview of the features and properties of ChucK that are relevant to on-the-fly programming. Based on these features and properties, *Section 4* defines a formal framework and programming model for on-the-fly programming. We show how the properties of ChucK are preserved and extended in the new model. *Section 5* uses the on-the-fly model and discusses an aesthetic for live performance. Finally, we conclude and discuss future work in *Section 6*.

2. BACKGROUND

Elements of on-the-fly programming have existed in computer music systems and languages in various forms. Some performers have incorporated various runtime programmable aspects in their

systems and methodologies. However, there hasn't been a formalized framework or a genuinely on-the-fly system that defines and addresses all the issues of runtime programmability.

2.1 Definition

In order to discuss the technical and aesthetic aspects of on-the-fly programming, we first provide a well-defined notion of what it is. In this context of this study, we define on-the-fly programming to consist of all the following elements:

- First executing an existing program, or possibly an empty program – we will call this P.
- Subsequently writing new code segment Q in a programming language, and adding it (possibly with type-checking and compilation) to the existing program P, forming program P'. Specifically, by "adding Q to P", we mean (1) Q now runs in the address space of P, potentially sharing data, and (2) there is a strong notion of temporal correspondence between Q and P, such that Q can access the timing of P and also synchronized with P. We shall see an example of this in *Section 4*.
- Modifying parts of the program P while it is executing. This is general enough to include anything that modifies the program structure or logic. We intentionally leave this open, for each system may have its own way of modifying the program. We will see that in ChucK, the program can be modified via replacement of concurrent, modular code blocks using the internal timing and virtual machine interface.

2.2 Challenges

In order to bring the power and general expressiveness of programming languages into on-the-fly programming, several fundamental challenges must be addressed. We have identified the following issues:

- **Modularity** – code sections must be modular so the programmer can reason about them or modify them independently. Furthermore, the augmented code must work together in the same address space and namespace.
- **Timing** – there must be a strong consistency and notion of time and timing between the existing and new parts of the program. Sequential on-the-fly code segments need to start and stop with precision.
- **Conciseness** and **manageability** – given the substantial time constraints, we ask: how can ideas be expressed concisely in code? How do we reason about time and data flow easily?
- **Flexibility** – how flexible is the system? Does it allow programmers to take advantage of the expressive power of programming languages in a real-time setting?

Taking these challenges into account, and using the definition provided, we next evaluate some existing languages and systems. In *Sections 3* and *4*, we will show how features of ChucK provide a solution to each of these challenges.

2.3 Existing Languages and Systems

Ever since Music I [6], there has since been many computer music programming languages and systems. [5, 4, 3, 9, 6] Many of these, especially the earlier ones were designed to operate in a non-real-time manner. They are interesting and influential to more modern languages, but are not directly relevant to this study of on-the-fly programming. Additionally, performers have used runtime programmable elements during live performance and/or rehearsal. Examples go back as far as Jim Horton, Tim Perkis, and John Bischoff of The League of Automatic Composers, who tweaked live electronics with microcomputers (KIM's) during performance, and George Lewis, as well as the network group The Hub, who used languages like FORTH to modify their systems online, to more recent laptop computer musicians who construct and use various on-the-fly tools, including command-line, shell scripts, and homemade software tools [2].

Of the real-time computer music languages, on-the-fly programming elements can be found in Max [10] and Pure Data [11], which allow programmers to alter parts of their patch while it is running. However, *data-flow* (unit generators and patches) is easy to represent whereas *timing*, in general, is significantly more difficult to discern and manipulate. Also, there are no mechanisms for programming smooth transitions when connecting sub-patches. Finally, the programming semantic can prove to be rigid when trying to incrementally add new logic to existing patches and modules.

SuperCollider [8], with its client/server architecture allows for synthesis patches to be compiled/interpreted on the client and sent to the server, where they can form a network of language-neutral synthesis elements, on-the-fly. However, there lacks a formal language-level framework (in addition to parameters to unit generators) for describing timing across all parts of the program as well as for exerting low-level timing control.

3. CHUCK OVERVIEW

ChucK is designed to be a concurrent, on-the-fly audio programming language [12]. It is not based on a single existing language but built from the ground up. It is *strongly-typed* and *strongly-timed*, and runs over a virtual machine with a native audio engine and a user-level scheduler. Several features of ChucK directly support/encourage on-the-fly programming:

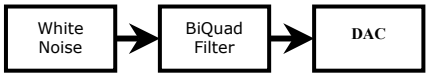
- A straightforward way to connect *data-flow* / unit generators.
- A sample-synchronous timing mechanism that provides a consistent and unified view of time. Timing is embedded directly in the program flow, making ChucK programs easy to maintain and reason about. *Data-flow* is fundamentally decoupled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

As a result, ChucK provides a programming model that solves several problems in computer music programming: representation, level of control of data-flow and time, and concurrency. We summarize the features and properties of ChucK in the context of these areas. In doing so, we lay the foundation for describing the semantics of the on-the-fly programming model in *Section 4*.

3.1 Representation

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

At the heart of the syntax is the *ChucK operator*: a group of related operators (`=>`, `->`) that denote interconnection and direction of data flow. A unit generator (*ugen*) patch can be quickly and clearly constructed by using `=>` to connect *ugen*'s in a strongly ordered, left-to-right manner (Figure 2). By default, `=>` *only* deals with *data-flow*, leaving the issues of *time* to the timing mechanism. Parameters to the unit generators can be modified using the *single* ChucK operator, `->`.



(a)

```
noise => filter => dac;
```

(b)

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

3.2 Level of Control

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

The solution in ChucK is to provide many levels and granularity of control over data and time. The key to having a flexible level of control lies in the ChucK timing mechanism, which consists of two parts. First, time (**`time`**) and duration (**`dur`**) are native types in the language. Time refers in a point in time whereas duration is a finite amount of time. Basic duration values are provided by default: `samp` (the duration between successive samples), `ms` (millisecond), `second`, `minute`, `hour`, `day`, and `week`. Additional durations can be inductively constructed using arithmetic operations on existing time and duration values.

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

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

// loop: update biquad every 100 ms
while( true )
{
    // sweep biquad center frequency
    500 + 300 * sin(now*FC) -> biquad.freq;

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

Figure 3. *A control loop.* The **`->`** ChucK operator is used to change biquad's frequency control parameter. The last line of the loop causes time to *advance* by 100 milliseconds – this can be thought of as the control rate.

This mechanism provides a consistent, sample-synchronous view of time and embeds timing control directly in the code. This strong correspondence between timing and code makes programs easier to write and maintain. *Data-flow* is decoupled from *time*. Furthermore, the timing mechanism allows for the control rate to be fully throttled by the programmer – audio rates, control rates, and high-level musical timing are unified under the same timing mechanism.

3.3 Concurrent Audio Programming

Sound and music are often the simultaneity of many precisely timed entities and events. There have been many ways devised to represent simultaneity [9, 4, 5] in computer music languages. However, until ChucK, there hasn't been a truly concurrent and precisely timed programming model for audio. This aspect of the language is a powerful extension of the timing mechanism and is essential to our model of on-the-fly programming.

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

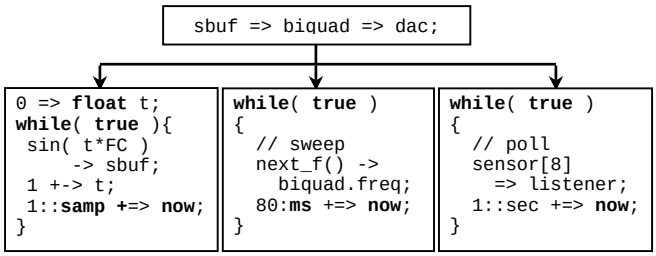


Figure 4. A unit generator patch and three concurrent paths of execution at different control rates.

ChucK introduced the concepts of *shreds* and the *shreduler*. A shred is a concurrent entity like a thread [1]. But unlike threads, a shred is a deterministic shred of computation, synchronized by time. Each of the concurrent paths of execution in Figure 4 can be realized by a shred. They can reside in separate source files or be dynamically spawned (*sporked*) from a single parent shred.

The key insight to understanding concurrency in ChucK is that *shreds are automatically synchronized by time*. Two independent shreds can execute with precise timing relative to each other and the virtual machine, without any knowledge of each other. This is a powerful mechanism for specifying and reasoning about time locally and globally in a synthesis program. Furthermore, it allows for any number of different control rates to execute concurrently and accurately. ChucK concurrency is orthogonal in

that programmers can add concurrency without modification to existing code. It is also scalable, because shreds are implemented as efficient user-level constructs in the ChucK Virtual Machine. Indeed, this mechanism is used in *Section 4* to synchronize on-the-fly program modules.

3.4 ChucK Virtual Machine

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

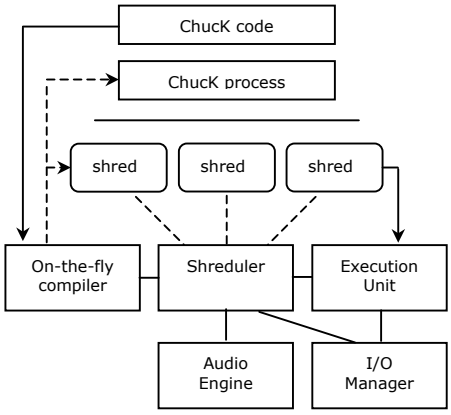


Figure 5. The ChucK Virtual Machine runtime.

4. THE ON-THE-FLY MODEL

In this section, we describe a formal on-the-fly programming model, based on the features of ChucK. We do so in two parts: *external* and *internal* semantic. We reason about key properties in the model and present an example. We show that just as concurrency in ChucK is a natural extension of the timing mechanism, we can leverage the timing mechanism and concurrency to address the challenges of on-the-fly programming.

4.1 Operational Semantics

4.1.1 External Interface

The on-the-fly programming model, at the high-level, can be described in the following way. A ChucK virtual machine begins execution, generating samples (as necessary), keeping time, and waiting for incoming shreds. ChucK shreds can be *assimilated* on-the-fly into the virtual machine, sharing the memory address space and global timing mechanism, and is said to be *active*. Similarly, an active shred can be *dissimilated*, or removed from the virtual machine, or it can be suspended or be replaced by another shred. This interface is designed to be simple, and delegates the actual timing and synchronization logic to the code within the shred (discussed in *Section 4.1.2*), leaving this flexibility to the programmer.

The high level commands to the external interface are listed below. They can be invoked on the command line, in ChucK programs (as functions calls to the `machine` and `compiler` objects), over the network, via customized graphical interfaces, or by other appropriate means.

- **Execute** – begins a new instance of the virtual machine in a new address space. Typically, this operation is used at the beginning of the session. Multiple instances of the virtual machine can coexist. The *shreduler* begins to keep track of time.

- **Add** – type-checks and compiles a new shred (from a ChucK source file, a string containing ChucK code, or a pre-compiled shred). If there are no compilation errors, the shred is allocated and *sporked* in the virtual machine with an unique ID. A new virtual stack is allocated, and the shred is *shreduled* immediately to execute from the beginning. When add fails due to compilation errors, the virtual machine continues to run as before while the programmer can attempt to debug, correct, and add the code.

- **Remove** – removes a shred by ID or name from the virtual machine. The shred's exit point function (if defined) is invoked and the shred and relevant child objects are garbage collected.

- **Suspend** – similar to remove, except the shred's suspend() function (if defined) is invoked, and the shred is removed by the *shreduler* and placed on the suspended list.

- **Resume** – resumes a suspended shred, calls its resume() function and places it in the *shreduler*'s ready-to-run list. The shred will resume execution at the suspended point in the code.

- **Replace** – invokes a remove operation followed by an add. An option exists for making the operation *atomic*.

- **Status** – queries the status of the virtual machine for the following types of information: (1) a list of active/suspended shreds ID's, source/filename, duration since assimilation (*spork* time), and (2) information on virtual machine state: currently executing shred, *shreduler* timeline, and CPU / VM usage by various parts of the system.

For example, Figure 6 shows code that adds, replaces, and removes two shreds using separate methods.

```
# start virtual machine with an "infinite time-loop"
shell%> chuck --start `while(true) 1::second +=> now;`
# add foo.ck
shell%> chuck --add foo.ck
# replace shred 0 with bar.ck
shell%> chuck --replace 0 bar.ck
# remove all shreds
shell%> chuck --remove all
```

(a)

```
// add shred from file "foo.ck"
machine.add( "foo.ck" ) => shred foo;
// advance time by 500 milliseconds
500::ms +=> now;
// replace "foo" with "bar.ck"
machine.replace( foo, "bar.ck" ) => shred bar;
// advance time by 2 seconds
2::second +=> now;
// remove "bar"
machine.remove( bar );
```

(b)

Figure 6. Two examples of using the runtime code management interface: **(a)** from a command-line shell, **(b)** from within a shred, which has timing control.

The "code-runs-code" feature is powerful because it allows a program to self-manage shreds on-the-fly with sample-synchronous precision. Users can also assimilate shreds that systematically add (potentially many) additional shreds, each with precise timing. Because the compiler and the virtual machine run in the same process, much of the intermediate processing can be eliminated. Finally, the ability to evaluate strings as code at runtime opens the possibility for self-generating on-the-fly programs with fast compilation-to-runtime response.

The status feedback is helpful for quickly surveying the state of the system and is particularly useful in an *on-the-fly* setting because it can identify hanging or non-cooperative shreds. For example, if the system runs a shred containing an infinite loop, it will fail to yield and cause the virtual machine execution unit to hang indefinitely. This type of behavior cannot be reliably detected at compile time, as the Halting Problem demonstrates. However, the *on-the-fly* programmer can identify and remove misbehaving shreds from the virtual machine *manually*, resulting in minimal interruption to the performance or session. While this recovery mechanism is far from perfect, it is far more advantageous than killing the system and restarting. Additionally, it can help the composer/performer tune the system by identifying shreds that are taking too much CPU time and optimize them individually.

This high-level semantic uses concurrent shreds as modules and provide a means of managing them. However, this interface alone is not adequate for specifying timing between incoming and existing modules. This brings us to the internal timing semantic of the on-the-fly programming model.

4.1.2 Internal Semantics

The internal semantics deal with the problem of precise timing between on-the-fly modules. The goal is to provide a consistent and accurate mechanism for shreds to synchronize with each other. In our model, the semantics are natural extensions of the ChucK timing mechanism. By querying and manipulating time using the special variable **now**, the programmer can determine the current time, and specify how the code should respond.

By the properties of ChucK timing and concurrency: (1) **now** always holds the current ChucK time, (2) changing the value of **now** *advances* time in ChucK and has the side effect of blocking the current shred (allowing audio and other shreds to compute) until **now** holds the value that was assigned to it, (3) if `t` is of type `time`, `t => now` advances time until `t` equals `now`, (4) if `d` is of type `dur` (a duration), `d +=> now` advances time by `d`. We illustrate this below with some common code segments that *synchronize to time*.

- Let time pass for some duration (in this case 10 seconds)

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

// or simply:

10::second +=> now;
```

- *Synchronize* to some absolute time t

```
t => now;
```

- *Synchronize* to absolute time t or later

```
if( t < now ) t => now;
```

- *Synchronize* to the beginning of next period of duration T

```
120::ms => dur T;      // period to synchronize to
T - (now % T) +=> now; // advance time by remainder
```

- *Synchronize* to the beginning of next period, plus offset D

```
T - (now % T) + D +=> now;
```

- Start as soon as possible

```
// no code necessary
```

The semantic allows programmers to precisely specify many more timing and synchronization behaviors. These statements can be placed to impose timing at arbitrary points in the program flow. For the purpose of initial time-based synchronizations in on-the-fly programming, they may be placed near the beginning of a shred to synchronize to time before moving on.

4.2 An On-the-fly Example

Using the operational semantics described in Section 4.1, we construct a simplified example: phasing on-the-fly. We write three shreds, each to trigger some sound at a slightly different rate, and we assimilate them one by one, with time-based synchronizations specified for the second and third shreds. The code for the three shreds and the commands to add them are shown in Figure 7. (With copy/paste, this can be realized in roughly 45 seconds.)

```
"a"=>sndbuf=>dac;

// first to run
// no need synch

while( true ){
 // trigger snd
 0 -> sndbuf.pos;
 300::ms +=> now;
}
```

```
"b"=>sndbuf=>dac;

300::ms => dur T;
T-(now%T) +=>now;

while( true ){
 // trigger snd
 0 -> sndbuf.pos;
 400::ms +=> now;
}
```

```
"c"=>sndbuf=>dac;

300::ms =>dur T;
T - (now%T) +
 150::ms +=> now;

while( true ){
 0 -> sndbuf.pos;
 500::ms +=> now;
}
```

(a)

```
shell %> chuck --start left.ck
shell %> chuck --add middle.ck
shell %> chuck --add right.ck
```

(b)

Figure 7. (a) Code for three concurrent shreds, the middle shred synchronized to the cycle of the left shred, and the right shred synchronized to cycle of the left shred offset by 150 milliseconds. **(b)** Shell command to add them on-the-fly.

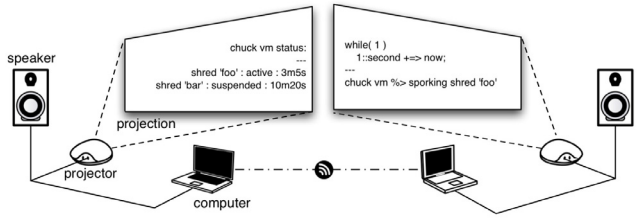
4.3 Properties

Recall the challenges we defined in *Section 2.2*: modularity, timing, conciseness, and flexibility. Using the features of ChucK and the framework we discussed, we briefly comment on the effectiveness of this model.

Concurrency provides a modular approach to breaking up the program into manageable pieces that can be added, remove, and replaced, and also synchronized to each other precisely, using the timing mechanism. This framework preserves the properties of timing in ChucK and extends them to an on-the-fly setting unifying high-level (musical), low-level (control rates), and inter-modular (shred synchronization) timing into one system. Finally, embedding the timing specifications directly in the languages and using the ChucK operator leads to cleaner and more maintainable code, which a runtime programmable system vitally demands.

5. AN OPEN ON-THE-FLY AESTHETIC

Our on-the-fly aesthetic (Figure 8) is one where the *process of on-the-fly programming* is conveyed to the audience. It addresses two important issues in computer music performance. First, it can be argued that many technical and aesthetic intentions are often difficult to discern in performance where they don't have to be or shouldn't be. The on-the-fly programming aesthetic help address this concern, for it provides a channel for the audience to see both the intention and the results. Additionally, it does this orthogonally, without necessarily depending on or interfering (usually) with the nature of the performance. Thus we call it an open aesthetic.



(2-performer schematic)

(performance)

Figure 8. An on-the-fly performance for two laptops and two laptop projectors. Note the two projections in the background. Superimposed are two projected screen shots from the performance. The schematic can be extended to any number of performers.

The second problem that the on-the-fly aesthetic addresses is the issue of virtuosity in computer music. On-the-fly programming provides a platform where the performer is able to render various types of mastery and creativity that can be immediately appreciated, or at least perceived. While typing speed may not inspire, the general expressive power of programming languages opens unlimited possibilities for clever approaches and beautiful design. The timing semantic makes ChucK code straightforward to follow, allowing the audience to more quickly and easily appreciate the design and construction of on-the-fly programs.

6. CONCLUSIONS AND FUTURE WORK

We have outlined some central challenges in on-the-fly programming, and presented a framework and an aesthetic for addressing them. The ChucK virtual machine provides a simple, yet powerful set of high-level operations to manage shreds externally, and allows the program and incoming *shreds* to manage timing and synchronization internally in the code. The concurrency model in ChucK gives a natural boundary between on-the-fly modules of the program. The timing mechanism can be use in the same manner to synchronize the incoming code to the rest of the program with sample-precision. Additionally, the syntax of the ChucK operator and the strong correspondence between timing and program flow help to design and reason about code in a time-constrained, on-the-fly setting. In its entirety, this model yields a flexible and powerful tool to create, manage, and further explore on-the-fly programs.

While this framework has many desirable properties, it still unpolished and unwieldy in many respects, because coding inherently takes time. Future work may look into programming environments that understands the deep structure of the program being written and facilitates writing and debugging on-the-fly. The performance aesthetic may explore visualizations of *program state* – in addition to code. Also, it would be interesting to investigate reducing the modular granularity, allowing finer pieces of code to be runtime modified.

`http://on-the-fly.cs.princeton.edu/`

ACKNOWLEDGMENTS

We wish to sincerely thank Andrew Appel, Brian Kernighan, Ari Lazier, Nick Collins and the authors of [2] for their support. Also thanks to the anonymous reviewers for their helpful comments.

REFERENCES

[1] Birrell, A.D. "An Introduction to Programming with Threads" Tech. Rep. SRC-035, Digital Equipment Corporation, 1989.

[2] Collins, N., A. McLean, J. Rohrhuber, A. Ward. 2004. "Live Coding in Laptop Performance." (To appear in *Organized Sound*).

[3] 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.

[4] Dannenberg, R. B., Desain, P., & Honing, H. 1997. "Programming Language Design for Music." In G. De Poli, A. Picialli, S. T. Pope, & C. Roads (eds.), *Musical Signal Processing*. Lisse: Swets & Zeitlinger.

[5] Loy, G. and C. Abbott. 1985. "Programming Languages for Computer Music Synthesis, Performance, and Composition." *Computing Surveys* 17(2):235-265.

[6] Lyon, E. 2002. "Dartmouth Symposium on the Future of Computer Music Software: A Panel Discussion." *Computer Music Journal*. 26(4):13-30.

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

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

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

[10] Puckette, M. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal*. 15(3):68-77.

[11] Puckett, M. 1996. "Pure Data." *In Proceedings of International Computer Music Conference*. International Computer Music Association, 269-272.

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