COMBINING ANALYSIS AND SYNTHESIS IN THE CHUCK PROGRAMMING LANGUAGE

Ge Wang
Princeton University
Dept. of Computer Science
Princeton, NJ USA
gewang@cs.princeton.edu

Rebecca Fiebrink
Princeton University
Dept. of Computer Science
Princeton, NJ USA
fiebrink@cs.princeton.edu

Perry R. Cook
Princeton University
Dept. of Computer Science
(also Music)
Princeton, NJ USA
prc@cs.princeton.edu

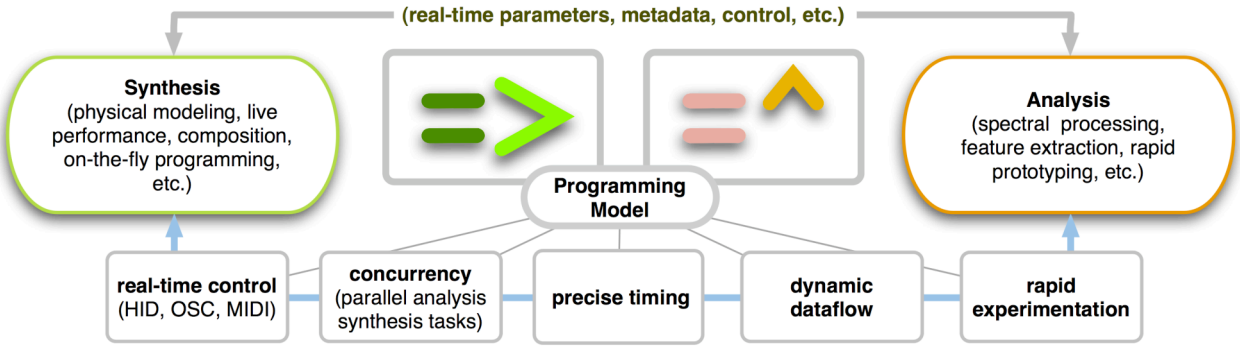

Figure 0. A ChucK-based programming model for building audio analysis and synthesis programs.

ABSTRACT

In this paper, we present a new programming model for performing audio analysis, spectral processing, and feature extraction in the ChucK programming language. The solution unifies analysis and synthesis in the same high-level, strongly-timed, and concurrent environment, extending and fully integrating with the existing language framework. In particular, we introduce the notion of a Unit Analyzer (UAna) and new constructs for dataflow, data types and semantics for operations in analysis domains, and mechanisms for seamlessly combining analysis and synthesis tasks in a precise, sample-synchronous manner. We present the motivation of our system, and describe new language-level syntaxes, semantics, and the underlying implementation. We provide code examples and discuss potential uses and benefits of the system for audio researchers, performers, and teachers.

1. MOTIVATION

Combining analysis and synthesis in the same framework can lead to interesting applications, as exemplified by the works of Roger Dannenberg and Chris Raphael [7], Nick Collins [4], and many others. Existing systems are mostly implemented in combinations of low-level C modules, open source libraries and frameworks, high-level languages, and proprietary software. We'd like to enable more people to experiment with, prototype, and create new tools and systems like these, *starting from a single unified, high-level platform*, without minimizing the need to develop plug-ins in other languages (e.g., C) or to write custom low-level modules from scratch.

Such a unified programming platform, we believe, should successfully address the following issues. First, in many musical applications, analysis and synthesis inform one another. Therefore, the environment should facilitate and encourage this symbiotic relationship, placing equal emphasis on the two, and providing flexibility and ease of programming for both. Next, the system should present a precise and flexible programming model with which programmers can rapidly prototype and implement analysis and synthesis tasks – and perhaps even do *on-the-fly*. Additionally, the high-level abstractions in the system should expose essential low-level parameters while doing away with syntactic overhead, thereby providing a highly flexible and open framework that can be easily used for a variety of tasks. Finally, it's extremely important that the written code represent the underlying algorithms and dataflow precisely and clearly. Overall, we envision a language that meets these criteria and that can be equally suitable for audio research (e.g., synthesis, spectral processing, feature extraction), pedagogy, composition, and musical performance.

There exist frameworks and languages that effectively address some components of our goals, including synthesis systems that accommodate analysis tasks and vice versa, and standalone systems that perform a specialized analysis/synthesis task. What we hope to achieve, in this work, is to produce a *single programming platform* that meets the need of a broad audience by offering solutions from programming language perspective. In doing so, we hope to encourage new and different ways to think about audio programming for audio and synthesis.

In this paper, we present our programming model for specifying precise audio analysis and synthesis tasks in the ChucK programming language – specifically designed to address the goals we outlined above. We

have designed and implemented new language constructs and integrated them into the existing ChucK framework, in a way that strives to capitalize on programmers' understanding of the existing language features, making analysis objects, dataflow, and control analogous to (but appropriately distinct from) their synthesis counterparts. The new system inherits the precise timing and concurrency model of ChucK and supports a syntax that clearly delineates dataflow and control in both analysis and synthesis processes.

The rest of this paper is organized in the following way. In Section 2, we situate our motivations and goals in the context of existing analysis and synthesis environments and tools. Section 3 presents our design of a new hybrid analysis/synthesis programming model. In Section 4, we illustrate this system using several examples of basic building blocks for analysis tasks. Section 5 addresses the implementation decisions involved in creating and integrating this model into ChucK. We conclude by discussing potential applications of a unified ChucK analysis/synthesis environment in Section 6 and future work in Section 7.

2. RELATED WORK

2.1. Analysis In Synthesis Environments

There are several high-level synthesis programming environments that provide modules for analysis-related tasks. Max/MSP [20] provides objects for forward and inverse FFT (via the `fft~`, `ifft~`, and related objects), constructs to facilitate windowing and overlap (e.g., `pfft~`), and support for frame-based operations. SuperCollider [15] provides FFT/IFFT Unit Generators as well as a variety of specialized objects for spectral processing (e.g., the `PV_*` objects as well as third-party objects for feature extraction and onset detection [5]). CSound [24] includes a variety of FFT-based spectral processing tools (e.g., `cvanal`, `hetro`, `lpanal`, `pvanal`). The analysis functionality of these systems relies on pre-made, "black-box" objects (e.g., coded and imported from C/C++) to meet the needs for common, specific analysis tasks. While these are powerful for many types of tasks, these environments, in general, are more intended for synthesis and less for designing and implementing new, low-level analysis systems directly in the language. Furthermore, many low-level analysis design tasks demand clear and precise control over time, and we believe that this may be difficult to achieve for many analysis tasks in existing systems.

Nyquist [6] allows programs to perform FFT/IFFT and directly access and manipulate frame-level data, which opens the door to explicit, low-level analysis and spectral manipulation in the program. Our system builds on this idea by providing a distinct, general class of analysis objects and by allowing a generic representation of the data passed between analysis modules (including, but not limited to, FFT frames). Furthermore, our analysis system applies the existing ChucK framework of time-based control and module connection for analysis.

2.2. Frameworks for Audio Analysis

Recent years have seen a proliferation of general tools and frameworks to perform audio analysis, particularly in the area of music information retrieval (MIR). Commonly used tools specialized for MIR include MARSYAS [23], CLAM [1], SndObj [12], MATLAB/Octave [14], M2K [8], and jAudio [16] / jMIR [17]. These tools support feature extraction from audio files, and classification and learning from these features. MARSYAS, SndObj, and CLAM also provide objects for high-level analysis/resynthesis (e.g., phasevocoder, sinusoidal resynthesis, spectral modeling synthesis). These are primarily libraries and frameworks; as such, they offer programmability at a different level than languages such as ChucK. As far as we can tell, there is no high-level language specialized for analysis tasks, much less one focused on support for real-time combined analysis and synthesis. Therefore, we feel that a ChucK-oriented programming model can be potentially interesting in its own right, and may serve as a complimentary tool to existing systems.

Rapid prototyping tools have an established role in MIR. M2K, for example, has been developed for this purpose. It provides a graphical patching environment for feature extractors, classifiers, and other modules. Dataflow and functionality in M2K itineraries are determined by connections between built-in and user-created objects, implemented in Java. MARSYAS also provides support for rapid prototyping, via Python and MARSYAS Scripting Language [3]. Our approach differs from these in that we hope to enable rapid experimentation from a single, high-level programming platform, further reducing turnaround time.

2.3. Analysis and Resynthesis Applications

There exists a set of applications such as AudioSculpt [2], SPEAR [10], and TAPESTREA [18] that integrate analysis and synthesis capabilities, in order to perform tasks such as transformation and resynthesis of existing sounds, and other processing such as cross-synthesis. These are specialized and do not intrinsically offer programmability, whereas we are interested in allowing programmers to implement similar tasks "from scratch" and to do so in real-time.

2.4. ChucK

ChucK [25] is a programming language for audio synthesis, whose programming model promotes a strong awareness of time and concurrency, and encourages rapid experimentation via on-the-fly programming. Unit generators can be dynamically connected and control can be asserted at any unit generator at any time and at any rate. ChucK also supports a precise concurrent programming model in which processes (called shreds) can be naturally synchronized via the timing mechanism and events. The property of precise low-level control over time and parallelism embedded in a high-level language is a foundation of ChucK's design, and we believe this property is highly desirable in programming analysis systems as well.

3. DESIGNING FOR ANALYSIS IN CHUCK

One of the first tasks in integrating analysis into ChucK was to design a hybrid programming model where both analysis and synthesis components fit naturally into the language and can work well together. Questions that arose include the following. How does a "strongly-timed" programming language handle operations and data/metadata in frequency (and other) domains? How might we exploit analogies to existing synthesis paradigms to allow elegant representation of analysis in the code? What is the appropriate level of detail and control, and how can we provide this while maintaining clarity and conciseness in the code? How should the similarities and differences between analysis and synthesis be reflected in the syntaxes and semantics?

One observation that shaped the answers to some of these questions was that even though we wish to carry out computation in transform domains (e.g., frequency), we still need to understand and control these operations with respect to time. For example, real-time spectral analysis is commonly performed via the Short-Time Fourier Transform (STFT), breaking up the audio stream into overlapping windows. In such analysis, parameters such as windowing, zero padding, and overlap and hop size can be crucial to the quality of the analysis. We wish to allow programmers to flexibly control these and other parameters over time, to operate on the results in a straightforward manner, and to reason about how and when analysis computations occur.

Our solution is threefold. First, we introduce the notion of a Unit Analyzer, which carries with it a set of operations and a connection model that resemble but are distinct from those of a Unit Generator. Following from this connection model, we then present an augmented dataflow model with datatypes, operators, and new objects. Third, we make use of the existing timing, concurrency, and event mechanisms in ChucK as a way to precisely control analysis processes. In the following subsections, we present each component in the context of established mechanisms for synthesis in the ChucK language. Section 4 supplies more in-depth and concrete examples to illustrate these ideas in practice.

3.1. Unit Analyzer

The Unit Generator (UGen) [13] is a building block of many synthesis systems, including ChucK. It is a modular abstraction of a single operation whose input and output are audio samples, and whose behavior is controlled through parameters that can be modified over time. In ChucK, UGens can be dynamically connected and disconnected in a global synthesis network, via the ChucK (**=>**) and unChucK (**=<**) operators. The ChucK Virtual Machine synchronizes the computation of the UGen network with that of ChucK shreds.

The principles of modularity, control via parameters, and relationship to a network of operations are well understood and might be applied to analysis "building blocks" as well, where the input and/or output may be data other than audio samples. Therefore, we introduce the notion of a Unit Analyzer (or *UAna*, pronounced "U-Wanna," plural *UAnae*). Like a UGen, a UAna defines a set of control parameters and can be dynamically patched with other UAnae and UGens. In contrast to UGens, UAnae pass generic data that may be in the form of spectral frames, feature vectors, metadata, or any other (intermediate) products of analysis. Natural candidates for UAnae include domain transformations such as FFT/DWT, feature extractors such as RMS, Flux, ZeroCrossing, and operations such as Correlation.

3.2. Dataflow

Currently in the language, the ChucK operator (**=>**) specifies how samples are passed between UGens in a synthesis network with respect to time. In a UGen-only network, UGens are connected via **=>** in a chain that terminates at a system-defined "sink" UGen (e.g., `dac`). The sink drives audio computation by "pulling" samples through the chain, starting with its "upstream" UGen neighbors. When an intermediate UGen is pulled (i.e., a downstream UGen requests the next sample), it first requests the sample from its upstream neighbor(s), then performs its own computations, and finally passes the output downstream. (This is often referred to as the "pull model.")

Figure 1. The ChucK and upChucK operators. Note how the upChucK is similar in representation but suggests orthogonal type of connection.

Because data passed between UAnae is not (necessarily) audio samples, and the relationship of UAna computation to time is fundamentally different (e.g., UAnae might compute on *blocks* of samples, or on metadata), the connections between UAnae have a different meaning. This difference is reflected in the choice of a new connection operator, the upChucK operator: `=^` (see Figure 1). `=^` has the following properties. First, a connection can be created between two UAnae using `=^` to indicate that the upstream UAna should pass its output as an input to the downstream UAna. This data is generated and passed at the pull request from the downstream UAna. Unlike the UGen pull model, however, there is no cascade of UAna pull requests that is automatically initiated by a sink. Instead, UAna chains must be *explicitly* driven by an operation in code, invoked at the UAna where analysis output is desired. This operation is performed via the `.upchuck()` member method, which initiates a cascade of pull requests upstream along all UAnae connected using `=^`, and then returns the analysis result at this point (see Figure 2 for an example). Because of the generic nature of data passed between UAnae and the need to directly access intermediate analysis data, the analysis output is represented in UAnaBlob objects,

which can contain vectors and matrices of numeric data as well as object references.

```
01# // a network of UAnae
02# UAna x =^ UAna y =^ UAna z;
03#
04# // initiate pull at 'y'; propagates to 'x';
05# // y's output in 'result'; z is unaffected
06# y.upchuck() @=> UAnaBlob result;
```

Figure 2. An example UAna network.

The combination of `=^` and the explicit `.upchuck()` operation allows the programmer to clearly construct UAna networks, and to selectively compute subgraphs at precise points in time. Each UAna caches its most recently computed UAnaBlob, associating it with a ChucK time stamp. If additional pull requests arrive at the same point in ChucK time, the cached copy is returned. Finally, it is possible to use `=>` to connect UGens and UAnae together, providing the means both to bridge synthesis and analysis subgraphs and for UAnae to generate and process audio samples (see Figure 3 for example).

```
01# // a chain of UGens and UAnae
02# adc => FFT fft =^ UAna x =^ IFFT ifft => dac;
03#
04# // compute at ifft (and upstream)
05# ifft.upchuck();
```

Figure 3. Connecting UGens to UAnae and back.

3.3. Programming Model and Time

One of the central strengths of ChucK is its precise control over time from the language. Embedded time advancement directives (e.g., `10::ms => now;`) allow programmers to specify timing at both high and low levels, and lead to more readable code in which exact timing can be readily inferred. Time is essential to synthesis in ChucK, for the timing directives serve as synchronization mechanisms between shreds in the ChucK virtual machine and UGens in the synthesis engine. In a sense, time *is* sound in ChucK.

```
01# // our patch (analysis)
02# adc => FFT fft => blackhole;
03# // set window; window size is 512
04# Windowing.hamming(512) => fft.window;
05# // set FFT size
06# 1024 => fft.size;
07#
08# // control loop
09# while( true )
10# {
11#     // take fft
12#     fft.upchuck();
13#     // HERE: do stuff with result
14#     // ...
15#     // advance time (effective hop size)
16#     256::samp => now;
17# }
```

Figure 4. FFT analysis.

We adopted this time-based programming model for the analysis system, allowing the programmer to invoke the `.upchuck()` operations at any point in time. This model extends to analysis, the precision and clarity of the synthesis programming model. For example, in Figure 4, we show an example of taking successive STFTs via the FFT UAna. Using the timing model, we are able to advance time by arbitrary "hops." Note that the key parameters of STFT are clearly represented. Furthermore, this model allows hop size and overlap to be dynamically changed in a natural and highly precise manner.

Additionally, we can create concurrent, hybrid analysis/synthesis programs, where various components can compute at potentially different rates. We can also leverage the concurrent and event programming model to provide additional clarity and control in a multi-shredded, multi-rate environment. Figure 5 shows an example involving generic UAnae.

```
01#  // a chain of UGens and UAnae
02#  adc => UAna x =^ UAna y =^ UAna z => dac;
03#  // spawn parallel processes
04#  spork ~ other();
05#  spork ~ another();
06#
07#  // control process #1
08#  while( true ) {
09#      y.upchuck();   // compute at y, x
10#      30::ms => now; // rate
11#  }
12#
13#  // control process #2
14#  fun void other() {
15#      while( true ) {
16#          z.upchuck();  // compute at z, y, x
17#          1024::samp => now; // rate
18#      }
19#  }
20#
21#  // control process #3
22#  fun void another() {
23#      while( true ) {
24#          // wait for event at 'x'
25#          // (triggered whenever y or z upchucks)
26#          x.event() => now;
27#          // use x's result on any parameter
28#          x.upchuck() => synth_param;
29#      }
30#  }
```

Figure 5. Multi-shredded, multi-rate analysis, with event notification.

The three components presented in this section form the basis of our analysis programming model.We next illustrate the flexibility and clarity of the system via more example code and discussion regarding several classes of analysis tasks.

4. EXAMPLES

4.1. Spectral processing

We first demonstrate some spectral processing building blocks by dissecting a simple FFT-based cross-synthesizer, shown in Figure 6. We first instantiate two FFT UAnae, and connect one audio source to each (lines

2–3). FFT is a special type of UAna, which "accumulates" audio samples from a UGen input. For this reason, we connect both to either `dac` or `blackhole`, which drives the UGen network (`blackhole` is much like `dac`, but produces no sound output). We then create and connect an IFFT object to the dac. IFFT is also special in that it produces audio samples. When pulled by dac, IFFT supplies the next samples in its overlap-add result buffer. Next, we set the parameters of the FFT and IFFT objects (lines 6–13). We then create two complex arrays to hold the output of the FFT's (lines 12–13). Native complex and polar datatypes and associated operations have been added to facilitate spectral processing (see Section 5).

In the control loop, the `.upchuck()` causes the FFTs to perform the transformations on the data in their respective accumulation buffers (this will compute on empty or partially empty buffers at startup). The results are acquired, point-wise multiplied, and copied back (lines 23–29). The IFFT object is then upchucked (line 31); the results of the inverse transform will be overlap-added into the IFFT's output buffer. Lastly, we advance time by one hop before repeating the loop (line 34).

```
01# // our patch
02# sourceX => FFT fftx =^ IFFT ifft => dac;
03# sourceY => FFT ffty => blackhole;
04#
05# // set FFT size
06# 1024 => fftx.size => ffty.size => int FFT_SIZE;
07# // desired window and hop size
08# 512 => int WIN_SIZE;
09# WIN_SIZE / 4 => int HOP_SIZE;
10# // set window and window size
11# // (FFT will automatically zero pad to size)
12# Windowing.hann( WIN_SIZE ) => fftx.window;
13# Windowing.hann( WIN_SIZE ) => ffty.window;
14#
15# // use this to hold contents
16# complex X[FFT_SIZE/2];
17# complex Y[FFT_SIZE/2];
18#
19# // control loop
20# while( true )
21# {
22#     // take fft's; fill result array
23#     fftx.upchuck().get( X );
24#     ffty.upchuck().get( Y );
25#     // multiply
26#     for( int i; i < X.cap(); i++ )
27#         X[i] * Y[i] => X[i];
28#     // copy data back into blob
29#     fftx.upchuck().set( X );
30#     // take ifft
31#     ifft.upchuck();
32#
33#     // advance time
34#     HOP_SIZE::samp => now;
35# }
```

Figure 6. A simple FFT-based cross synthesizer.

4.2. Feature Extraction

Our system supports straightforward extraction of arbitrary time and frequency domain features, which might be stored or used as parameters to drive real-time synthesis. For example, Figure 7 shows the UAna network for extracting several standard spectral features. In this example, we create an optional "agglomerator" UAna that does no computation but, when upchucked, drives the synchronous computation of all UAnae connected to it. The output of the FFT object is connected to each of these spectral feature extractors, so synchronous feature computation can take advantage of caching at the FFT object. Alternatively, the programmer can also choose to `.upchuck()` each feature extractor at separate rates. Note that this can be easily modified to be multi-shredded and event-driven, similar in form to the example in Figure 5.

```
01# // audio input
02# adc => FFT fft;
03# UAna agglomerator => blackhole;
04# // connecting FFT to feature extractors
05# fft =^ Centroid c =^ agglomerator;
06# fft =^ Flux f =^ agglomerator;
07# fft =^ MFCC m =^ agglomerator;
08# fft =^ RollOff r =^ agglomerator;
09#
10# // control loop
11# while( true )
12# {
13#     // upchuck!
14#     agglomerator.upchuck();
15#     // HERE: do stuff with results
16#     // ...
17#     // advance time
18#     HOP_SIZE::samp => now;
19# }
```

Figure 7. Simple spectral feature extraction. Note that parameters can be dynamically set for feature extractors at any point in the code.

The list of supported or planned feature extractors includes Centroid, RMS, Flux, RollOff, MFCC, ZeroCrossing, Correlation, and several others. Since we can directly access and manipulate the intermediate analysis results (via UAnaBlobs) at any point, it is also possible to experiment with and implement custom feature extraction algorithms directly in ChucK.

4.3. Combining Analysis and Synthesis

A primary goal of the system is easy integration of analysis and synthesis in the same program. As a simple example, consider the real-time separation of vowels and consonants in an input audio stream into distinct channels. UAnae would be responsible for identifying the vowel/consonant components while UGens would be used before and after analysis to capture and render the audio, informed by the analysis results in real-time.

Another example of a hybrid system is real-time LPC analysis, transformation, and resynthesis [19]. In this case, one might use the Correlation object to compute the linear prediction coefficients, along with pitch/non-pitch and power estimation. These coefficients can either be stored or relayed to LPC resynthesis module for transformation and rendering.

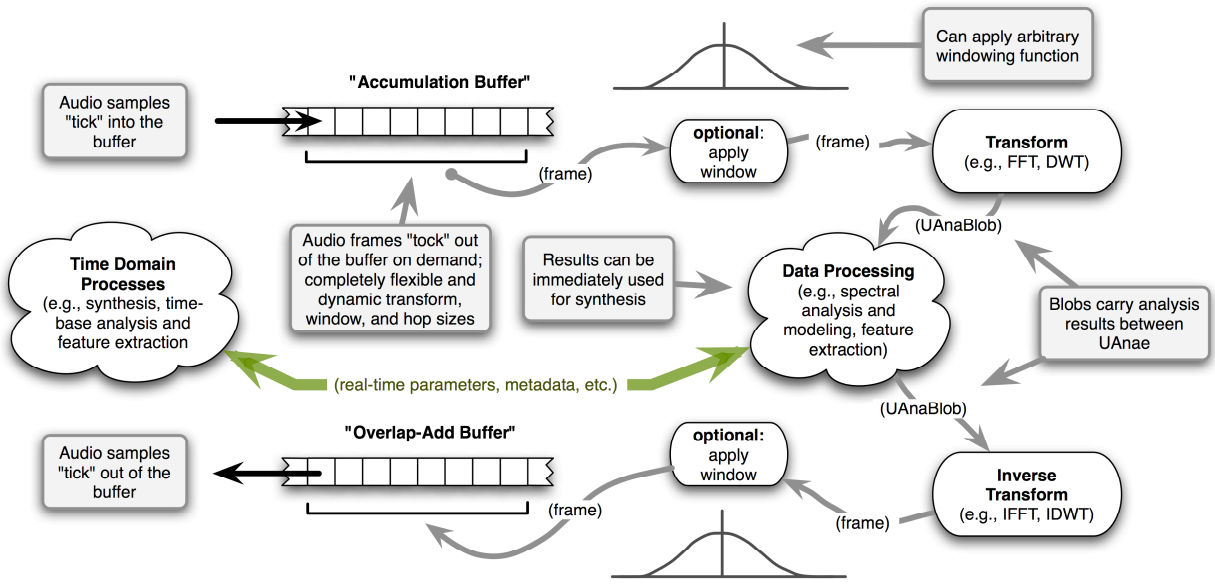

Figure 9. Underlying pipeline of a generic hybrid synthesis/analysis system.

These are a few examples of what is possible. By leveraging the generality and precision of the model, we believe it can be straightforward to specify a variety of algorithms for synthesis and analysis directly in ChucK.

5. IMPLEMENTATION

5.1. Underlying Language Support

In order to support the analysis system described above, we have made the following additions to the ChucK programming language and virtual machine. First, we added native datatypes for UAna, UAnaBlob, `complex` and `polar`, and defined their behaviors on relevant operations. We then extended the UGen framework and introduced a UAna data pipeline, as well as support for introducing new UAna objects into the type system and virtual machine. Additionally, we implemented an initial set of basic UAnae.

```
01# // the complex value (a + b*i)
02# #(a,b) => complex cmp;
03# // example polar value (rho,theta)
04# %(3,pi/2) => polar pol;
05#
06# // accessing complex members
07# cmp.re => float real;
08# cmp.im => float imag;
09# // accessing polar members
10# 2.0 => pol.mag => float magnitude;
11# pi => pol.phase => float phase;
12#
13# // converting to polar (by type cast)
14# #(3,4) $ polar => polar bear;
15# // example arithmetic expression
16# (pol * bear) $ complex + #(1,2) => complex x;
```

Figure 8. Some operations on complex and polar types.

5.2. Complex and Polar Data Types

As part of our goal of placing equal emphasis on analysis and synthesis, we introduce two additional data types: `complex` and `polar`, intended provide strongly-typed and flexible handling of data, particular for spectral processing, where the need to manipulate scalars and vectors in complex or polar form arises more frequently.

In the new analysis system, complex values can be stored in complex and polar variables and expressions and used in computation. Their components can be accessed via the dot operator. See Figure 8 for examples of use.

5.3. Dataflow Pipeline

In order to support the semantics of `.upchuck()`, we have integrated a UAna subsystem into the ChucK engine. Similar to synthesis, where UGens are "ticked" to generate and/or process audio samples, UAnae are "tocked" to generate and process UAnaBlobs. A separate "overlay" UAna network is maintained over the synthesis network. The reason is for this is that some UAnae (e.g., FFT and IFFT) deal with both audio samples and UAnaBlobs. In the case of domain transformation UAnae such as FFT and DWT, samples are ticked into an "accumulation buffer" that always holds the previous W samples, where W is equal to the window size. This allows the UAna to be upchucked at any point in time to produce UAnaBlobs that hold the results of the transformation. Because UAnaBlobs are time-aware, the system can automatically and safely reuse analysis results for `.upchuck()` requests made at the same ChucK time. An example dataflow pipeline is depicted in Figure 9.

6. POTENTIAL APPLICATIONS

6.1. Audio Analysis Research and Pedagogy

One of the primary strengths of our ChucK-based analysis system is that programmer can specify analysis task with sample-synchronous precision. This allows for the exact scheduling of operations, and also allows any parameter to be changed at any time. For example, for FFT analysis, it's possible to conduct FFT analysis with dynamically changing window, hop and FFT sizes. With concurrency, programming precise analysis and synthesis processes is straightforward. This has potential benefits in prototyping systems for audio compression, multi-rate spectral estimation and tracking, and representing complex analysis systems as smaller, simpler, concurrent components.

Due to the precise and concise syntax of the language support for analysis, and because of ChucK's support for on-the-fly programming [26], another potentially useful application is fast prototyping for new algorithms in analysis and hybrid analysis and synthesis systems. Using the tenets of ChucKian on-the-fly programming, it is possible to craft programs whose fundamental structure and logic can be altered in real-time for rapid turn-around in experimenting with new ideas.

Additionally, since the programming model is such that the code can accurately and completely specify an algorithm in terms of dataflow and timing, ChucK source code can serve as a clear and compact vehicle for communicating new ideas and algorithms within a community, or perhaps to document how part of an algorithm works. It seems potentially useful (and fun) to be able to exchange the latest ideas as code, and to then be able to immediately test it and make new modifications.

The deterministic and concise nature of the system can also facilitate education, allowing teachers to demonstrate analysis algorithms clearly, while the rapid prototyping can provide a potentially more "online" and "by-example" approach to teaching concepts in and out of the classroom. This approach also has the advantage of staying in a single environment, without having to require students (and graders!) to negotiate multiple languages and programming paradigms. Lastly, the programming model is both high-level and detailed (i.e., it exposes low-level control) at the same time. As demonstrated by earlier examples, the representation provides direct control over analysis parameters and data, allowing students to immediately and accurately experiment with these concepts.

6.2. Synthesis and Performance

Because both analysis and synthesis components work and interact in the same language, it can be straightforward to prototype and create hybrid systems for analysis-driven synthesis. For example, ChucK can serve as a useful workbench for experimenting with systems for data-driven concatenative synthesis [22], feature-based synthesis [9], and audio mosaics [11]. Also, one can imagine implementing analysis tasks such as spectral modelling synthesis [21], onset detection, instrument identification, beat-tracking, and others as building blocks for exciting new synthesis and performance systems directly in ChucK. Furthermore, the ability to integrate on-the-fly programming of analysis and learning techniques into a synthesis environment holds the potential of enabling new and interesting musical applications.

7. FUTURE WORK AND CONCLUSION

Next steps in this work include implementing additional UAnae to further enable rapid prototyping with basic analysis modules. Further development on analysis objects distributed with ChucK will continue to attend to the balance between low programming overhead and flexible control. In general, we favor simpler and more general modules over ones intended for specific tasks, as we hope to make it easy to create highly customized systems directly in ChucK. Another exciting vein of work is to investigate new language-level solutions to supporting real-time and on-the-fly learning and classification.

In conclusion, we have introduced a new programming model for specifying and controlling analysis and for working with hybrid analysis/synthesis systems. We presented our notion of a Unit Analyzer, as well as a means for understanding how general analysis tasks may be accomplished by a network of such objects. Strengths of the model include a clear representation of dynamic dataflow between and among analysis and synthesis components, and a precise mechanism for specifying and understanding how analysis is performed in time. Furthermore, the syntax exposes low-level control in a concise and straightforward manner.

Our new ChucK system offers a single platform in which programmers can craft new analysis algorithms in the language, perform analysis-informed synthesis in real-time, and rapidly prototype novel audio algorithms. The precision, clarity, and on-the-fly nature of analysis and synthesis in ChucK, we hope, can lead to exciting new applications. Finally, we hope this model can provide a different way of thinking about programming analysis (and synthesis) for research, performance, and pedagogy.

ChucK is open-source and freely available:

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

ACKNOWLEDGEMENTS

Special thanks to George Tzanetakis, Dan Trueman, and Spencer Salazar for many illuminating discussions.

REFERENCES

[1] Amatriain, X., P. Arumi, and M. Ramirez. 2002. "CLAM, yet another library for audio and music processing?" *Proc. ACM OOPSLA*.

[2] Bogaards, N., A. Röbel, and X. Rodet. 2004. "Sound analysis and processing with AudioSculpt 2." *Proc. ICMC*.

[3] Burroughs, N., A. Parkin, and G. Tzanetakis. 2006. "Flexible scheduling for dataflow audio processing." *Proc. ICMC*.

[4] Collins, N. 2007. "Towards autonomous agents for live computer music: Realtime machine listening and interactive music systems." *PhD Thesis*. Centre for Science and Music, Faculty of Music, University of Cambridge.

[5] Collins, N. 2006. "BBCut2: Incorporating beat tracking and on-the-fly event analysis." *Journal of New Music Research* 35(1): 63–70.

[6] Dannenberg, R. B. 1997. "Machine tongues XIX: Nyquist, a language for composition and sound synthesis." *Computer Music Journal* 21(3): 50–60.

[7] Dannenberg, R. B., and C. Raphael. 2006. "Music score alignment and computer accompaniment." *Communications of the ACM* 49(8): 38–43.

[8] Downie, S. 2004. "International music information retrieval evaluation laboratory (IMIRSEL): Introducing M2K and D2K." *Handout at ISMIR. http://www.music-ir.org/evaluation/m2k/ v4_ISMIR2004_Handout.pdf*

[9] Hoffman, M., and P. R. Cook. 2006. "Feature-based synthesis: A tool for evaluating, designing, and interacting with music IR systems." *Proc. ISMIR*.

[10] Klingbeil, M. 2005. "Software for spectral analysis, editing, and synthesis." *Proc. ICMC*.

[11] Lazier, A., and P. R. Cook. 2003. "MoSievius: Feature-based interactive audio mosaicing." *Proc. DAFx*.

[12] Lazzarini, V. 2000. "The sound object library." *Organised Sound* 5(1): 35–49.

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

[14] Mathworks, Inc. *MATLAB Documentation*. http://www.mathworks.com/

[15] McCartney, J. 1996. "SuperCollider: a new real-time synthesis language." *Proc. ICMC*.

[16] McEnnis, D., C. McKay, I. Fujinaga, and P. Depalle. 2005. "jAudio: A feature extraction library." *Proc. ISMIR*.

[17] McKay, C. *jMIR Sourceforge Project*. http://sourceforge.net/projects/jmir

[18] Misra, A., P. R. Cook, and G. Wang. 2006. "Musical tapestries: Re-composing natural sounds." *Proc. ICMC*.

[19] Moorer, J. 1979. "The use of linear prediction of speech in computer music applications." *Journal of the Audio Engineering Society* 27(3): 134–40.

[20] Puckette, M., 1991. "Combining event and signal processing in the MAX graphical programming environment." *Computer Music Journal* 15(3): 68–77.

[21] Serra, X., and J. O. Smith. 1989. "Spectral modeling synthesis." *Proc. ICMC*.

[22] Schwarz, D. 2004. "Data-driven concatenative sound synthesis." *PhD Thesis*. University Paris 6.

[23] Tzanetakis, G., and P. R. Cook. 2000. "MARSYAS: a framework for audio analysis." *Organised Sound* 4(3).

[24] Vercoe, B., and D. Ellis. 1990. "Real-time CSOUND: Software synthesis with sensing and control." *Proc. ICMC*.

[25] Wang, G., and P. R. Cook. 2003. "ChucK: a concurrent, on-the-fly audio programming language." *Proc. ICMC*.

[26] Wang, G., and P. R. Cook. 2004. "On-the-fly programming: Using code as an expressive musical instrument." *Proc. NIME*.