MODELS AND APIs FOR AUDIO SYNTHESIS AND PROCESSING
A PANEL DISCUSSION

Stephen Travis Pope
CREATE Lab
University of California, Santa Barbara
stp@create.ucsb.edu

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
rbd@cs.cmu.edu

ABSTRACT

The topic of this panel is object models, programming languages, and application programmer's interfaces for digital audio processing. The panel brings together developers who share the experience of designing more than one model or language for DASP software, and who are willing to compare and contrast them in terms of their underlying assumptions, the design trade-offs they made in the process, and the rules of thumb that they have learned over their years of experience.

Referring back to previous works on languages and environments for computer music, we ask the questions raised in the past: what factors contribute to the success of a computer music language or tool? How do we best apply the features of modern programming languages, development and delivery environment? What have we learned since the 1980s (or the 1960s for that matter) that influences model and language design for music?

The panel will be structured a free-flowing discussion of the decisions and trade-offs in system design, rather than a show-and-tell of each of the participants systems, or long position papers.

1. INTRODUCTION

In the words of D. Gareth Loy [1, p. 52-53], the good news is that "the ever-widening availability of computers has made their application to music one of the most exciting and challenging issues in music today. [...] They can serve the conceptualization of both musical ideas and ideas about music, fueling the experimentalism that is near the core of the musical genius of our times." The bad news is that we must ask ourselves "how indeed can we extract and realize the promise of the computer for music. How can one best represent musical imagination with a computer? The experience of translating one's musical goals into a computer's vocabulary is often anything but liberating." [1] This panel discussion will address the topic of how our models of music signals and data structures affect the design of the software libraries and programming languages we use for music. We hope to stay at a technical level, and to explore the design space of music models, contrasting system we've built or used.

There is a rich literature of reports on models, languages and APIs for audio synthesis, but too little discussion of the design issues, and the trade-offs one encounters in their implementation. Several authors have attempted surveys of music languages over the years, and a panel discussion on the topic was organized by Eric Lyon and held at Dartmouth in 2001 [2]. There are valuable but very different taxonomies of music systems and languages in [3] - [7]; discussions of music language design issues can be found in [8] - [12]. In [11], James McCartney lists and evaluates the abstractions available in modern programming languages and how they can be applied to computer music, and in [12] Carla Scaletti discusses the factors that contribute to the success of a programming language. These two article will feed into the panel discussion.

Max Mathews wrote in 1969 that,

"the two fundamental problems in sound synthesis are (1) the vast amount of data needed to specify a [sound] pressure function—hence the necessity of a very fast and effective computer program—and (2) the need for a simple, powerful language in which to describe a complex sequence of sounds. Our solution to these problems involved three principles: (1) stored functions to speed computation, (2) unit generator building blocks for sound synthesizing instruments to provide great flexibility, and (3) the note concept for describing sound sequences. [... The composer] would like to have a very powerful and flexible language in which he can specify any sequence of sounds. At the same time, he would like a very simple language in which much can be said in a few words, that is, one in which much sound can be described with little work. The most powerful and universal possibility would be to write each of the millions of samples of the pressure wave directly. This is unthinkable.

"At the other extreme, the computer could operate like a piano, producing one and only one sound each time one of 88 numbers was inserted. this would be an expensive way to build a piano. [...] In a given instrument, the composer can connect as many or as few unit generators together as he desires. Thus he can literally take any position he chooses between the impossible freedom of writing individual pressure-function samples and the straightjacket of the computer piano. [13 p. 34]

The question that arises for us is, 38 years later, are these still our goals and basic assumptions? If not, what are the fundamental design decisions that one faces in creating a new object model, language, or library API for digital audio signal processing (DASP)? What have we learned in 38 years, and how do modern processors and system architectures, real-time operating systems, and software engineering skills ranging from metamodels to object-oriented design patterns change the ground rules for modeling DASP servers today?

2. QUESTIONS

We will organize the panel discussion around a series of simple questions that the participants have had time to study beforehand. The intention is to move beyond each of the presenters simply describing their latest or favorite systems to the more interesting level of evaluating the ramifications of certain design decisions. The set of questions presented here is for example only, the panel members will select from this list according to their interests.

2.1. Core models

Given the canonical model of unit generators and buffers, what changes if we use object-oriented analysis or modeling, and object-oriented design patterns in describing DASP languages?

Do we need to talk about models and even metamodels at all in DASP system design?
If yes, are there useful precedents?
What about object-oriented design patterns?
What's a signal, a unit generator, a port?
What's the advantage of a procedural or object-instance model of unit generators, as opposed to one that concentrates on operations on signals?
Are functional models of DASP useful?
How do we handle time and functions of time?
Do we want to merge synthesis and control?
What about old-fashioned orchestra languages—are we still bound to the unit generator model?
Do we differentiate between interactive and off-line DASP systems?
How about languages geared towards "live coding?"
How (if at all) is a parameter stream (input) different from a feature vector (output)?

2.2. Scalability and Efficiency

In the early years, non-real-time performance on slow and memory-starved machines was of prime importance, and many design options were not considered because of assumptions about performance and scalability. Nowadays, implementors still express concerns related to cache performance and function-call overhead.

How have tuning and scalability considerations evolved with very-high and very-low language levels and the use of virtual machines and multi-stage compilers?
Do we do computation on blocks of data?
Is there a control rate (as different from audio rate)?
How do we integrate our systems with the call-back-oriented sound I/O APIs found on all modern operating systems?

2.3. Metadata

Score and synthesis tools need representations with abstract property models, and some DASP applications also use feature extraction and complex metadata for music information retrieval services. Is a good synthesis API automatically a good analysis API?

Are there different kinds of signals? If yes, what are their semantics?
What does it mean to have signals that are "about" other signals?
How do you handle merged data streams such as positional geometry and other annotations?

2.4. User Extension

There is always a switching point and associated learning curve when a system has more than a single language for both scoring/patching and DASP system extension (programming new unit generators or compositional algorithms).

Can a single model and language scale from composition to signal processing?
Is there a patching language or scripting format?
Is it the same a the unit-generator implementation or system extension language?
Is there a separate note-list score format or two?
Is there a procedural or stochastic composition language? How is it extended?
Are these supported by an integrated development environment (IDE) with a debugger and source code management tools?

2.5 Parallel Processing

Until recently, hardware advances came in the form of faster clocks, faster memory, and hidden parallelism within processor pipelines and multiple arithmetic units. Future hardware advances will include multiple-core processors. Dual-core CPU's are common, but we expect to see much greater parallelism in the near future.

What will we do with 32 or 64 cores?
What architectures are most suited to harnessing this kind of parallelism?
What can we learn from computer music systems of the past that used hardware parallelism in the form of multiple DSP chips?
Where do we expect to find parallelism? In basic algorithms such as the FFT? At the effect and instrument level? Spread among plug-ins, effects, mixers, and coarse-grain computation?
Do we need more computation? What will more computation do for languages, users, sounds?
How will multi-core processors affect models and APIs for DASP?

2.6 Inter-module Communication

Software systems have made great progress in supporting a modular approach based on the integration of applications and plug-ins. Not only has this approach offered a technical solution supporting more flexibility and creativity, it has also fostered a new marketplace for software DASP modules. What will we see next?

Current interfaces are oriented toward single-threaded audio processing: fill the input buffer with samples, call the plug-in to process the data, copy the samples to the next plug-in in the chain, etc. Should the interface change to communicating processes? Should the host use multiple threads?
Jack [14] supports inter-process communication, but it is also single-threaded. Is this a good idea? Will Jack soon be obsolete, or are there simple ways to adapt it for more parallelism?
Parallelism in pipelines generally requires additional buffers and more latency. Should we simply compensate by putting fewer samples in buffers? Should we look for parallelism elsewhere?
If we admit pipelining, how does the host application, operating system, or audio sub-system delegate processors to tasks? Is the assignment static, dynamic, based on what?

Given a possibly dynamic configuration of audio modules, how do we manage timing so that input controls take effect deterministically and synchronously? Or is best-effort and overall low-latency without precise timing (as in MIDI) the way to go?

REFERENCES

[1] Loy, D. G. "The CARL System: Premises, History, and Fate", Computer Music Journal 26:4, 2002.

[2] Lyon, E, et al. "The Future of Computer Music Software: A Panel Discussion", Computer Music Journal 26:4, 2002.

[3] Loy, D. G., and Abbott, C. "Programming Languages for Computer Music Synthesis, Performance, and Composition." ACM Computing Surveys 17(2): 235-266. 1985.

[4] Loy, D. G. 1989. "Composing with Computers—A Survey of Some Compositional Algorithms and Music Programming Languages." in M. V. Mathews and J. R. Pierce, eds. Current Directions in Computer Music Research. MIT Press. 1989.

[5] Wiggins, G. et al. "A Framework for the Evaluation of Music Representation Systems", Computer Music Journal, 17:3 31-42, 1993.

[6] Pope, S. T. "Music Composition and Scoring by Computer." (Invited chapter) in G. Haus, ed. Music Processing. A-R Editions, 1992.

[7] Pope, S. T. "Computer Music Workstations I have Known and Loved." in Proceedings of the International Computer Music Conference. 1995.

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

[9] Dannenberg, R., P. Desain and H. Honing. "Programming Language Design for Music," in Roads, Pope, Piccialli, and De Poli, eds., Musical Signal Processing, Swets and Zeitlinger, pp. 271-316. 1997.

[10] Puckette, M. "Max at Seventeen", Computer Music Journal 26:4, 2002.

[11] McCartney, J. "Rethinking the Computer Music Language: SuperCollider", Computer Music Journal 26:4, 2002.

[12] Scaletti, C. "Computer Music Languages, Kyma, and the Future", Computer Music Journal 26:4, 2002.

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

[14] Jack Audio Connection Kit. http://jackaudio.org