A Language for Interactive Audio Applications

Roger B. Dannenberg

School of Computer Science, Carnegie Mellon University
email: dannenberg@cs.cmu.edu

Abstract

Interactive systems are difficult to program, but high-level languages can make the task much simpler. Interactive audio and music systems are a particularly interesting case because signal processing seems to favor a functional language approach while the handling of interactive parameter updates, sound events, and other real-time computation favors a more imperative or object-oriented approach. A new language, Serpent, and a new semantics for interactive audio have been implemented and tested. The result is an elegant way to express interactive audio algorithms and an efficient implementation.

1 Introduction

Creating interactive audio programs is difficult and not very well supported by traditional programming languages such as Lisp, C++, or Java. Libraries built upon these languages can help, but even with library support, creating a dynamic, interactive, reliable system is very difficult. A particularly difficult problem is creating networks of sound generators and processors on the fly in response to input events. A simple example is a synthesizer that creates a computation in response to a key down event, updates envelope generators in response to a key up event, and frees all relevant resources when the sound decays to silence.

In the following sections, I describe some existing approaches to interactive audio programming and derive some properties that easy-to-program systems seem to share. Then, I describe some concepts that are hard to express in existing languages and systems. In particular, the problems of updating parameters and creating abstractions are described. Next, I explain some new ways of organizing programs that solve these problems. This approach has been implemented in the Aura framework using a new scripting language, Serpent, that was developed specifically to address the problem of interactive audio programming.

One of the difficulties of this type of research is evaluation. How do we know when a language is good, and what things should we avoid? These questions cannot be answered objectively, but lessons learned from the programming language community and from experience in the computer music domain can be used to derive some general guidelines. Languages should provide good abstraction mechanisms and orthogonal features. They should provide general building blocks rather than special-case solutions, and languages should minimize the semantic gap between language concepts and application concepts.

2 Existing Approaches

There are many systems available for digital audio including music synthesis and processing. A standard reference is csound (Boulanger, 2000), which has roots in the Music *N* languages (Mathews, 1969). The csound orchestra language illustrates both good and bad features for an interactive audio programming language. The good features include a somewhat functional style in which sounds are created by instances of functions (instruments) and where instances of function-like unit generators are created simply by mentioning them in an expression. It seems to be natural to express signal-processing algorithms in terms of an acyclic graph (or patch), and the csound orchestra programs tend to be semantically close to this conceptual organization.

On the negative side, csound instruments cannot be composed hierarchically. One cannot define a new unit generator in terms of primitive ones. Because all instruments output sound to global variables, flexibility is curtailed. A classic problem in csound is how to add reverberation to the mixed output of many instruments. The standard solution relies on an inelegant arrangement of global variables and programming conventions.

Arctic (Dannenberg, 1984) was intended to provide a more elegant expression of the semantics that underlie Music *N* languages. Arctic is a more pure functional language where unit generators, instruments, and even scores are all simply functions that return signals (real-valued functions of time). Arctic allows better abstraction than csound and allows a clean solution to the "reverberation problem." The functional nature of Arctic makes it somewhat awkward at handling discrete events. For example, after creating an instance of a function in response to a key-down message, the instance must inspect every key-up message for a matching key number in order to know when to stop. Furthermore, to cause an envelope to ramp to zero in response to a key-up event, the existing envelope must be stopped and replaced. In an imperative

Published as: Roger B. Dannenberg (2002). "A Language for Interactive Audio Applications." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

language, it might be a simple matter to alter the state of an envelope object.

In spite of its shortcomings, Arctic semantics are quite powerful for audio processing. A less interactive version of Arctic (called Nyquist) (Dannenberg, 1997) has been implemented and used extensively for composition tasks. The ease-of-use found in Nyquist confirms the importance of a functional notation for audio processing algorithms. This problem is how to combine a functional signal processing language with an imperative or object-oriented event-processing language.

A good example of an object-oriented event-processing language is Aura (Dannenberg & Brandt, 1996), which is an extension of C++. In Aura, an event corresponds to the arrival of a message. This causes the execution of a method (member function). In contrast to functional languages, this object-oriented approach seems very natural for receiving messages and responding to them. Aura and its predecessor, the CMU Midi Toolkit, have been used for many interactive music programs and compositions (Morales-Manzanares, Morales, Dannenberg, & Berger, 2001). However, Aura, an object-oriented language, has been much more difficult to use than Nyquist, a functional language, for audio computation.

Judging by its popularity, Max with added signal processing primitives (Dechelle et al., 1998; Lindemann, Dechelle, Sarkier, & Smith, 1991; Puckette, 2002; Zicarelli, 1998) is an interesting approach. Max uses a visual programming language to describe audio computation. The visual layout is a direct expression of the data flow graph. A drawback of this approach is that it is hard to express graphs whose structure depends upon real-time data or whose structure changes dynamically. Various extensions, such as specialized support for polyphony and the ability to activate and deactivate sections of the graph have been developed to work around the static nature of these graphs. Max was not designed to be a general programming language and it is generally considered to be problematic for many programming tasks.

SuperCollider (McCartney, 2000) is the best known system for interactive audio that provides a general-purpose programming language. Although based on an object-oriented programming language, SuperCollider adopts a highly functional programming language approach to deal with audio. SuperCollider encourages the dynamic instantiation of audio generating objects, which are described using a functional programming style. The interface between discrete event processing and signal processing is supported by a variety of classes and protocols that many users find confusing and limiting. Thus, simple tasks such as routing MIDI controller values to a particular signal processing parameter can involve rather sophisticated programming constructions.

CLM (Schottstaedt, 1994) and Jsyn (Burk, 1998) are object-oriented frameworks that support sound synthesis. The user creates unit generator objects and connects them to form patches. This approach was the first one taken in Aura (Dannenberg & Brandt, 1996), but it is not very satisfying because it is too easy to make programming mistakes. Typical errors include leaving inputs unconnected or trying to patch a control-rate output to an audio-rate input. These mistakes are not so easy to make with a more functional style of patch language. The main problem with these systems is that the user manipulates unit generators to build desired patches rather than writing expressions that express the desired sounds directly.

Open Sound Control (Matt Wright & Freed, 1997) is not a language but a protocol for communicating with a sound synthesis engine. OSC offers a model for connecting a real-time interactive program dealing with discrete events, especially parameter updates, to a synthesis program computing continuous streams of audio. One characteristic of OSC is that it uses a hierarchical naming scheme. Thus, one can set parameters of objects deeply embedded within a patch. In most implementations, this scheme works against structural abstraction because it gives full view and access to the inner workings of complex sounds. On the other hand, translations between low-level device- or algorithm-specific parameters and high-level OSC parameters provides a useful abstraction mechanism, hiding peculiarities of input devices and synthesis algorithms. (Madden, Smith, Wright, & Wessel, 2001; Matthew Wright, Freed, Lee, Madden, & Momeni, 2001) Since OSC only provides a synthesizer interface, it remains for additional structure and language design to provide a complete programmable system.

The M Orchestra Language (Puckette, 1984) is probably the closest in concept to the present work. This language allows nested expressions to describe patches in a functional style and uses an object-oriented framework to describe instrument methods that can be invoked to manipulate unit generator parameters.

As can be seen from these examples, there are many different approaches and many interesting features in existing work. However, it would be hard to argue that any approach is best or that “best” can even be defined. This is an evolving area with many possibilities remaining to explore. In this work, I introduce some new concepts for organizing interactive programs and music compositions. These are supported by a new (but mostly conventional) language and a corresponding implementation. The work is novel in its support for abstraction and the simplicity with which object-oriented control schemes can be combined with functional-programming-oriented signal processing schemes, leading to a versatile and conceptually simple programming environment.

3 Conceptual Framework

To make progress toward better languages, it helps to have a conceptual model for the organization of audio computation. There is no single “true” model, but the following model is based on an understanding of existing

systems and experience writing audio and non-audio interactive music systems.

- Audio computation is performed by a *patch*, an interconnected set of *unit generators* (UGs).
- A UG has state and can be updated, for example by setting the frequency of an oscillator or triggering an envelope to begin or end.
- A patch is described using an expression language, e.g. *multiply*(*oscil*(…), *envelope*(…)).
- A patch is an abstraction mechanism: patches can be used as if they are primitive UGs.
- Patches can be passed as parameters to other patch expressions.
- Just as UGs can be updated, patches can be updated.

The notion of "update" needs some discussion. A patch represents a synchronous computation on streams of samples. There may be streams as input and the patch may produce one or more streams of output. In addition, the patch may have parameters or state variables that are used in the stream computations. These parameters may be modified asynchronously with respect to the stream computation. Again, setting an oscillator's frequency is a good example. It is important to note that updates are not streams and they are not computed synchronously. The patch never waits or polls for an update. In our implementation, updates can only occur between the computation of a block of samples.

It is the concept of *update* that breaks the pure functional programming model. Previous work has tried to reformulate update sequences into stream-like values (Dannenberg, Desain, & Honing, 1997; Letz, Fober, & Orlarey, 2000), and recent work by Brandt (Brandt, 2000, 2001) has explored the use of type constructors to model updates within a functional framework. However, we believe that programming interactive systems can be more straightforward using updates than functional dependencies, and we want to explore how updates and functional models can coexist.

Thus, the model attempts to combine elements of functional programming with elements of object-oriented programming. From the functional programming perspective, a patch or UG is a function that returns a signal *value* that can be passed as a parameter to other functions. From the object-oriented perspective, a patch or UG is an *object* that can retain state and receive messages that alter the state. The critical language design problem is to support both of these views in a form that makes it easy to reason about program behavior.

4 The Dual Nature of a Patch

To embrace both object-oriented and functional perspectives, a patch must have a dual nature. On the one hand, it is an object. One can reference the object, examine the state of the object, and change the state. On the other hand, the patch must look like a value. The value is the audio stream computed by the patch. In more operational terms, if `a` and `b` represent patches, then as objects, we can say:

```
a.set_hz(440.0)
```

and as values, we can write an expression such as:

```
sum(a, b)
```

Described in this way, this seems almost too simple and obvious, but if so, this is a strong argument that we are on the right track. The challenge is to extend this simple beginning with the abilities to define patches, pass parameters, and support updates, all without sacrificing simplicity and ease of use.

4.1 Defining a Patch

A patch is a computation performed by a collection of UGs, which are primitives built into the system. Since a patch has the dual nature of object and value, we define it using an expression within an object-oriented class definition. (Further syntactical simplifications are possible.) The following example defines a note as the product of an oscillator and an envelope:

```
class Note(Instr):
   def patch(hz):
      mult(osc(hz), env(a,b,c))
```

Note that the last line is a functional-style description of a graph of UGs: `mult`, `osc`, and `env`. As with functional systems, evaluating this expression creates new instances of these UGs. Here is how this definition works: an instance of class `Note` is created by evaluating an expression such as `Note(440.0)`. This creates an instance of class `Note`, which inherits from class `Instr`. `Instr` defines an initialization method that, in turn, calls `patch`, passing it the `hz` parameter. The result of `patch` (a reference to the new instance of `mult`) is stored in an instance variable.

At the lowest levels, we want graphs of UGs where each UG has a direct pointer to the UGs it depends on so that samples can be read directly from one UG by the next. What happens when an `Instr` like `Note` is passed to a UG such as `mult`? In fact, UG functions like `mult` check their parameters and replace `Instr`'s with the stored value (a reference to a UG) returned from the `patch` method. Thus, at the lowest level where performance is critical, much of the abstraction falls away, and we are left with the desired computation graph of UG primitives. Retained and still visible at the higher level are objects that are used to update and manage the UGs.

4.2 Updates

Recall that updates are the point where objects and functional programming meet. How do we effect a change in a computation that is functionally specified? Somehow, after a patch has started, we need to go back to the patch, locate a specific UG, and alter some of its state. A direct

way to do this is to save references to UGs as they are instantiated. For example, we could write:

```
the_osc = osc(hz)
return mult(the_osc, env(a,b,c))
```

If `the_osc` is an instance variable of an `Instr`, then we can use it later to access the oscillator UG. This approach can work, but leads to an awkward programming style where expressions are broken up by assignments. Even if assignments are in-line within the expression, they are still bothersome, and more code must be added manually in order to perform updates.

Another possibility is to mark certain parameters as "updatable" and to automate the rest. In this approach, the patch expression might look like:

```
mult(osc(_hz:hz), env(a,b,c))
```

and the interpretation is as follows: "this instrument has a settable attribute named `_hz` whose initial value is `hz` (a parameter to `patch`). Updates will be specified in terms of attribute/value pairs. When the `_hz` attribute is updated, pass the value on to the instance of `osc`." Note that in this scheme, the instance of osc remains anonymous, and all the apparatus to manage updates can be automatically generated from this expression.

We will describe a simple implementation for clarification. Each primitive UG function such as `osc` and `mult` is implemented as a method in class `Instr`. The expression `_hz:440.0` is compiled as `update(_hz, 440.0)`, which simply creates a new object to hold the two values `_hz` and `440.0`. The `osc` method tests the parameter type: if it is float, the value is used for the initial frequency; if it is an update structure, the initial frequency is pulled from the structure, and in addition, an update map is extended to include "map updates to the `_hz` attribute to the frequency attribute of *osc*," where *osc* is a reference to the actual instance of the `osc` unit generator created by this `osc` method. Now, the program can call another method of `Instr` as follows:

```
someinstr.set(_hz, 600.0)
```

and the update will be passed to the oscillator's frequency (phase increment) parameter.

4.3 Abstraction Issues

With the techniques described above, one can create and assemble patches hierarchically. For example, the `Note` class can be combined with a filter to form a new class:

```
class Note2 (Instr):
   def patch(hz, cutoff):
      lowpass(Note(_hz:hz), _co:cutoff)
```

and instances will have two updateable parameters: `_hz`, and `_co`. There is a question here of whether *all* parameters should be updateable. Why not simply allow something like

```
a_note2.note.osc.hz = 600 ?
```

The whole reason for abstraction is to hide some details while bringing to focus other features. Unconstrained access to synthesis parameters may offer great flexibility, but it may also hinder the management of complexity and ultimately become limiting. A good analogy is programming with parameterized procedures. One always defines a procedure in terms of parameters, and it is never possible to say "call $p(5)$, but when p calls $q(x)$, pass 3 as the actual parameter value." Just as we cannot reach inside a procedure declaration to modify the code, by analogy, we should not reach inside a patch to manipulate it.

Our experience with Nyquist is that, rather than make all parameters visible, it is sufficient to modify code to expose the parameters of interest. Rather than violate the abstraction, simply redefine it to be more suitable. In our language proposal, making a parameter updateable requires only the addition of an attribute name (and a colon), e.g. `_co:` in front of the initial value expression. If the parameter of interest is several abstraction layers down, new parameters must also be added to the intermediate patches.

Figures 1 and 2 illustrate the analogy between conventional parameter passing and update parameters. Figure 1 shows how values are renamed according to formal parameter names as nested procedures are called. Figure 2 illustrates how an update message, setting an attribute to a value, propagates to nested patches, also with renaming according to a static definition.

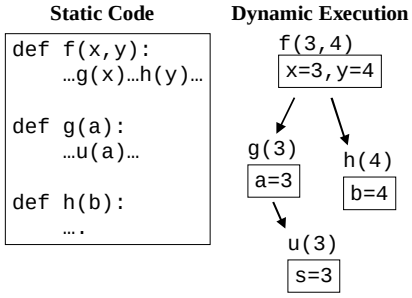

Figure 1. An illustration of conventional parameter passing. Parameters are passed from f to g and h and from g to u. Variable bindings are shown in boxes at right. u is a primitive with formal parameter s.

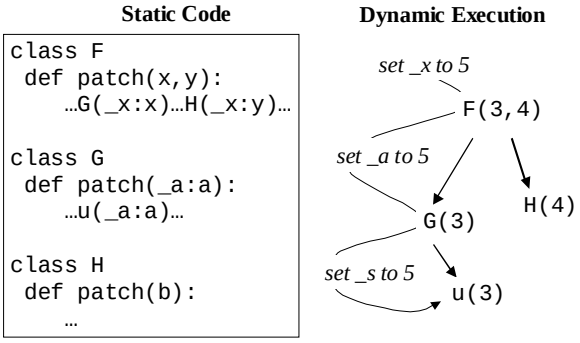

Figure 2. An analogous patch created by `F(3,4)`, which instantiates subpatches `G` and `H`. `G` includes unit generator `u`. When an update is sent to an instance of `F`, the update attribute `_x` is mapped to `_a` and passed to `G`, and then to `_s` and passed to `u`. `u` is a unit generator with attribute `_s`.

For user-defined instruments to work like unit generator primitives, it is necessary to declare *formal* attributes. Notice that the patch in `G` includes `_a:a` in its formal parameter list. This tells `F`, for example, to map a *set _x* message to a *set _a* message before sending it on to `G`, just as local variable `x` is mapped to formal parameter `a` in Figure 1 when `f` calls `g`.

4.4 Non-Identity Updates

Not all updates are simply passed through to a lower-level patch or UG. It must be possible to filter, select, and transform updates when desired. For this, the object-oriented framework is ideal for adding update "handlers" in the form of methods. A special syntax is defined as illustrated here:

```
on _hz(hz):
    set_to(id, _actual_hz, f(hz))
```

so that when an update to the `_hz` attribute arrives, the value is bound to the parameter `hz`, and the following code body is evaluated. This code applies function `f` (for example, `f` might quantize `hz` to the nearest semitone) and updates `_actual_hz`. If the patch is written to "listen" to `_actual_hz`, the patch updates will be quantized.

More elaborate schemes, including constraint systems, could easily be envisioned, but this one seems to be easy to understand and effective.

5 Implementation

To explain the implementation, we need to discuss attributes in greater detail. Our system is an extension of Aura (Dannenberg & Rubine, 1995), where objects are named by globally unique identifiers (*Aura ID*s) and where objects respond to messages of the form "set *attribute* to *value* at *time*." A new real-time object-oriented language, based on Python (Python, 2002), runs within Aura. The new language is named *Serpent*, and its objects are independent of Aura objects, mainly because of multithreading and garbage collection issues that we will ignore here. UG primitives are Aura objects, and are thus only accessed by sending updates via Aura messages. To maintain consistency and the advantages of globally unique names, `Instr` objects have "shadow" Aura objects that call Serpent methods when Aura messages are received. Thus, it is possible to send Aura messages to an `Instr`, and thus an `Instr` can masquerade as a first-class Aura object.

The examples in this article are "real," that is, executable Serpent code, and run with minor alterations under Aura to produce real-time interactive audio. A few compiler extensions have not been implemented, so the `_hz:440.0` notation must be entered as `update(_hz, 440.0)`, and the `on _hz(hz):` notation must be entered as `def set_hz(hz):` and a call to `add_handler(_hz, 'set_hz')`.

Memory and object management is an important problem. While Serpent employs per-process real-time garbage collection, Aura objects are not garbage collected (GC in Aura would require a distributed real-time garbage collector, a formidable design problem left for future work.) Thus, if an `Instr` is instantiated at run time, it needs to be freed along with all the objects it created to avoid running out of memory. Since UG functions are member functions, they can build a hidden list of patch elements, so that when the `Instr` is freed, the UGs can also be freed. Aura uses backpointers so that when a UG is freed, any references to it are automatically redirected to a source of zeros, at least limiting the damage of "dangling pointers." An `Instr` should often be freed when it stops making sound. Mechanisms exist for envelop generators and other UGs to report completion via an update, which can then be used to free the `Instr`, but at present, this is an explicit programming task.

5.1 Serpent Details

Although Aura is implemented in C++, this language is not conducive to rapid prototyping of interactive audio programs. A scripting language seemed to be necessary, but existing languages exhibit long latencies due to garbage collection and thread context switching. Serpent was designed with a real-time garbage collector that can be tuned to have less than 1ms pauses. Also, Serpent is designed to allow multiple instances in one address space. Each instance can run on a separate, preemptable thread, allowing Serpent programs to run with very low latency. Although latency is tightly controlled, Serpent is much slower than C++, so it is not suitable for DSP at the sample level. However, it is sufficiently fast to configure patches of unit generators which are written in C++.

Serpent is open source, cross-platform, and available from the author. As a real-time scripting language, it has many potential uses in computer music systems. All access to MIDI, audio, and Aura are made through external function calls and a few bits of conditional compilation, so Serpent can be configured in many ways. Serpent also has an interface to wxWindows (Smart & Roebling, 2001), bringing easy-to-program, platform-independent graphical interfaces to Aura or stand-along programs. Aura is also open source, but continues to evolve, and is not well documented. The author is happy to work with collaborators, of which there are now a few.

6 Summary

One of the interesting things about interactive audio programming systems is that they are so unlike most other programming systems. Subroutine and class libraries have not worked as well as entirely new languages and language models. Some existing languages and systems are widely used, but this seems as much a reflection of the need for support as an indication that problems are solved. Every system has well-known and widely discussed drawbacks and pitfalls. With this in mind, it is important to consider

alternatives and to explore different principles, concepts, and assumptions.

The Serpent/Aura system described here is one such new approach. It begins with the premise that functional programming is essential to express signal computation. The main advantage of functional programming is that nested expressions are so readable: it is obvious that *filter*(*sum*(*osc*(…), *osc*(…))) is the filtered sum of two oscillators. The object-oriented style of building a patch is much less readable:

```
Filter filter;
Adder sum;
Osc osc1(...), osc2(...);
filter.input = sum;
sum.input1 = osc1;
sum.input2 = osc2;
```

The Serpent/Aura organization provides "unit generator" abstraction in that user-defined patches behave just like unit generators. They return signals as values, they can be used to define new patches, and they can be updated by setting attributes to new values. It is important to note that instruments *do not* mix their outputs to some predetermined output channel, which would make it impossible to compose instruments into higher-level structures.

While the functional programming style is natural for patches, imperative programming seems more natural for building interactive, reactive systems. The design assumes that it is easier, in general, to say "make the following state changes when the key goes down" than to say (for each changeable thing) "here are my functional dependencies upon the times at which the key goes down (along with all other dependencies)."

Given these premises, we develop a model where computations exhibit a dual nature as both objects and (signal) values. The crossover between these two aspects is so transparent that code is quite readable, with intuitive semantics. References to patches, viewed as signals, can be assigned to variables, passed as parameters, and stored in data structures. Viewed as objects, patches can be updated (i.e. their internal state can be changed) in real time in response to real-time data and sensors.

One of the key innovations in this work is the concept of update: an attribute/value pair directed to a patch. This is where object-oriented or procedural reactive programs "connect" to functional patch programs. The key problem is that in functional programs, there is no state and no "object" to update. In this solution, updates become a form of parameter passing, where the parameter value is not available *after* the expression is instantiated. The update concept supports information hiding and a mapping, from "formal" or "external" attributes to which a patch responds, to "actual" or "internal" attributes of component unit generators and nested patches.

Although a particular syntax and implementation was chosen in this work, the ideas are quite general. One could imagine other ways to specify update attributes and their mapping from patches to patch components.

5 Conclusions

There is much to learn about programming interactive audio systems effectively. I have offered a novel approach and described its implementation. This approach is a direct attack on the split between functional programming, which seems perfect for most synchronous signal processing tasks, and object-oriented programming, which seems ideal for interactive, event-driven programs. The result offers great promise as a very high-level programming language system for interactive audio. An implementation is available now for sympathetic users, and a demonstration of much larger and interesting examples than could be explained within the pages of this paper will be included in the conference presentation.

6 Acknowledgements

I am very grateful to IBM Research and their Computer Music Center for financial and technical support in 2000 and 2001. Ron Kuivila offered useful insights and comments about related work and the problems addressed in this paper. This work is supported in part by NSF Award #0085945 and an IBM Faculty Partnership Award.

References

Boulanger, R. (Ed.). (2000). *The Csound Book*: MIT Press.

Brandt, E. (2000). "Temporal Type Constructors for Computer Music Programming." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 328-331.

Brandt, E. (2001). "Implementing Temporal Type Constructors for Music Programming." *Proceedings of the 2001 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 99-102.

Burk, P. (1998). "JSyn - A Real-Time Synthesis API for Java." *Proceedings of the 1998 International Computer Music Conference*. San Francisco: International Computer Music Conference, pp. 252-255.

Dannenberg, R. B. (1984). "Arctic: A Functional Language for Real-Time Control." *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. San Francisco: ACM, pp. 96-103.

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

Dannenberg, R. B., & Brandt, E. (1996). "A Flexible Real-Time Software Synthesis System." *Proceedings of the 1996 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 270-273.

Dannenberg, R. B., Desain, P., & Honing, H. (1997). Programming Language Design for Music. In C. Roads & S. T. Pope & A. Piccialli & G. d. Poli (Eds.), *Musical*

Signal Processing (pp. 271-316): Swets & Zeitlinger Publishers.

Dannenberg, R. B., & Rubine, D. (1995). "Toward Modular, Portable, Real-Time Software." *Proceedings of the 1995 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 65-72.

Dechelle, F., Borghesi, R., Ceccco, M. D., Maggi, E., Rovan, B., & Schnell, N. (1998). "jMax: a new JAVA-based editing and control system for real-time musical applications." *Computer Music Journal, 23*(3), 50-58.

Letz, S., Fober, D., & Orlarey, Y. (2000). "Real-Time Composition in Elody." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 336-339.

Lindemann, E., Dechelle, F., Sarkier, M., & Smith, B. (1991). "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal, 15*(3), 41-50.

Madden, T., Smith, R. B., Wright, M., & Wessel, D. (2001). "Preparation for Interactive Live Computer Performance in Collaboration with a Symphony Orchestra." *Proceedings of the 2001 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 310-313.

Mathews, M. (1969). *The Technology of Computer Music*: MIT Press.

McCartney, J. (2000). "A New, Flexible Framework for Audio and Image Synthesis." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Conference, pp. 258-261.

Morales-Manzanares, R., Morales, E., Dannenberg, R. B., & Berger, J. (2001). "SICIB: An Interactive Music Composition System Using Body Movements." *Computer Music Journal, 25*(2), 25-36.

Puckette, M. (1984, 1985). "The M Orchestra Language." *Proceedings of the 1984 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 17-19.

Puckette, M. (2002). Pd. http://crca.ucsd.edu/msp.

Python. (2002). Python Home Page. http://www.python.org.

Schottstaedt, B. (1994). "Machine Tongues XVII: CLM: Music V Meets Common Liso." *Computer Music Journal, 18*(2), 30-37.

Smart, J., & Roebling, R. (2001). wxWindows. http://www.wxwindows.org.

Wright, M., & Freed, A. (1997). "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers." *Proceedings of the 1997 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 101-104.

Wright, M., Freed, A., Lee, A., Madden, T., & Momeni, A. (2001). "Managing Complexity with Explicit Mapping of Gestures to Sound Control with OSC." *Proceedings of the 2001 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 314-317.

Zicarelli, D. (1998). "An Extensible Real-Time Signal Processing Environment for Max." *Proceedings of the 1998 International Computer Music Conference*. International Computer Music Association, pp. 463-466.