

Building A Sustainable Repertoire: A Performer's Approach to Realizing Interactive Electroacoustic Works

David Brooke Wetzel, DMA
Mansfield University of Pennsylvania
dwetzel@mansfield.edu
davidbrookewetzel.net

Interactive electroacoustic works often require specialized systems that may be unavailable to interested performers, especially when the original technology has become outdated or obsolete. The author describes an approach to interactive computer music interface design that enables the realization of diverse works by multiple composers within a single performance setup. This approach is based on certain common features among a number of interactive compositions discovered through analysis of the original systems and their subsequent reconstructions using newer technology. This research has been conducted from the perspective of a performer attempting to build a sustainable repertoire of interactive electroacoustic works. The primary goal is to understand the technological requirements of each piece and to create a musically faithful realization of the composer's work using equipment that is convenient to the performer. In the interests of sustainability, portability, and affordability, this approach seeks to free such works from specific electronic instruments and from the necessary involvement of the composer in each performance realization. The author presents a framework and a software prototype for incorporating a wide array of works into a single modular system, based on the author's previous work analyzing and reconstructing electroacoustic works by Musgrave, Kramer, Pennycook, and Lippe.

1. Introduction

For musicians devoted to the concert performance of interactive electroacoustic works by composers other than themselves, one of the most pressing challenges is to maintain compatibility between their own electronic performance resources and the various technical demands of composers utilizing often custom-designed or proprietary software and hardware. This challenge is made even more difficult by the fact that specialized electroacoustic systems rapidly become obsolete or unavailable.

As a clarinetist interested in developing a sustainable electroacoustic repertoire, my solution to this challenge has been to develop new realizations based on analysis of the original (now obsolete) systems required in a number of specific works. However, individual realizations of particular works still require a certain measure of maintenance in order to remain viable for periods longer than a few years. Therefore, I have begun to develop a prototype for a modular software system that incorporates multiple interactive works into

a single, standardized environment. A fairly simple core program sits at the center of a scalable interactive system, while user-maintained scripts interconnect and control simple processing modules according to the needs of each piece or performance situation. New processing modules may be created and added to the system in order to expand its functionality as needed. Existing modules can be easily modified or updated. In many cases, these modules can be adapted from, or at least modeled on, components of existing systems created by composers for specific works. The advantage, from a performer's perspective, is that only one software application requires maintenance, while numerous works and performance settings are available for use on stage without the need to switch between applications.

The following is, first, an overview of the common features among interactive electroacoustic works that make a single, adaptable, modular system possible. Secondly, I present a detailed description of the prototype software's structure. Finally, I

offer my speculation on the future of this software: its possible uses, expansions, and refinements.

2. Analysis of Existing Works

In order to maintain a viable repertoire, performers often need to successfully re-create or update existing works as the available technology changes. Such updates and changes to a composer's interactive system are most useful and trustworthy when the original system is thoroughly analyzed and understood in terms of its basic functions and controls. In other words, a performer must understand what the machine is supposed to *do*, rather than merely which buttons to push. Such an analysis can then serve as a model for re-creations of the required interactive system long after the original devices and systems used by the composer are obsolete and unavailable. Such an analysis is even more useful in cases where the composers themselves are no longer available to assist in the process of re-creation and renewal of their electronic systems.

To date, I have formally analyzed four works for clarinet and interactive electronics by Musgrave, Kramer, Pennycook, and Lippe (Wetzel 2004), and I have performed and studied numerous other interactive electroacoustic works. Throughout this research, some common features have emerged among these works that suggest the possibility of a standardized performance system for incorporating a performer's entire repertoire into a single software environment. The two most important common features are a) control of the system and its variable parameters at discrete points throughout the musical score according to an "event list," and b) the divisibility of most systems into self-contained processing modules.

2.1 The "Event List" as an Organizing Principle in Interactive Electro-acoustic Music

Numerous interactive electroacoustic works use the concept of an "event list" as a

fundamental organizing principle for interaction between performer(s) and the electronic system. For example, Bruce Pennycook developed the event-list-driven "MIDI-Live" system for his *Preascio* series (1989-93 for various instruments) in order to trigger MIDI playback files and change system settings in response to live actions by the performer (Pennycook 1991). Cort Lippe's *Music for Clarinet and ISPW* (1992, adapted for Max/MSP in 1999/2004) similarly used a software-based event list to control system variables in real time, following the performer's progress through the score (Wetzel 2004). Thea Musgrave's *Narcissus* (1987, for flute or clarinet and digital delay) gives changes to a delay system's settings at various points directly in the performer's score (Musgrave 1989). Jonathan Kramer's *Renascence* (1974, for clarinet, tape, and tape delay system) features a technician's line in the score that conveys changes to a matrix mixer controlling a long delay (Kramer 1977). The settings found within the musical scores by Musgrave and Kramer, though originally intended as directions for physical manipulation of hardware devices, are easily translated into software-based event lists similar to those used by Pennycook and Lippe. My own recent reconstructions and performances of these last two works have been based on exactly this approach (Wetzel 2006 and 2004).

The event list, in the cases mentioned above, is therefore a series of discrete points along the timeline of the musical score. At each of these points, some action is required of the performer, which results in a change to the electronic system or its sub-processing units. In all the above cases, manipulations of the interactive system required for performance can be accomplished with a combination of event list cues and a small number of external live controllers (footswitches, etc.).

2.2 The "Processing Module" in Interactive Music Systems

The works previously mentioned not only rely on an event list for execution of

real-time controls, but also utilize interactive electronic systems that can be divided into a series of discrete processing functions. In most cases, these functions are easily described using standard audio or MIDI processing terminology (e.g. oscillators, filters, delays, harmonization, MIDI/audio file playback, etc.). Occasionally, a composition features synthesis or processing algorithms that are highly unusual or wholly unique. Yet in these cases, such processing algorithms take place within a network of more conventional processing units (sound i/o, physical controller interfaces, standard audio processors, etc.). Separated from the utility functions of an interactive system, even the most exotic processing modules may often be described in abstract, device-independent terms (i.e., DSP algorithms, etc.). Therefore, most interactive systems can be seen as composed of a number of definable processing modules within a complex network.

Often, a piece makes use of processing modules similar to those employed in another composer's work. Typically each piece would feature its own implementation of such a module, with proprietary settings, connections, and methods for accessing controls. Separated from the complexities of their individual proprietary environments (i.e., a software interface dedicated to a single work), common features emerge, allowing for the description of a common implementation.

By understanding the functions of each processing module used in an interactive system, it becomes much easier to re-create its functions using newer or alternative technology. Furthermore, by understanding and possibly re-designing the various components of an interactive system, it also becomes possible to re-use those components in other situations, much as we routinely do with standard audio processing hardware or standard acoustic instruments.

3. An "Event List"-Based Approach to Modular Interface Design

Based on my previous analysis of existing works, and my own need, as a

performer, for a simplified, portable performance setup, I have designed a script-driven prototype system for performing numerous interactive works from within a single, scalable software environment. This prototype is designed in Max/MSP. However, since this system is more protocol than comprehensive application, a version of this system could be easily implemented within an alternate computer music environment.

The concept is fairly simple: an entire interactive performance system is assembled from a list of required modules. Each module is a separate processing unit designed for a specific task, such as control data processing, audio signal processing, media file playback, sampling/synthesis, audio routing, on-screen data display, or connections to external controllers or output devices. Connections between modules (audio and/or control data), and their initial settings, are defined on the fly from the event list. An interactive composition is then performed using this system as its electronic instrument, which is scripted from the event list file and/or controlled directly from other external or live input sources.

3.1 The "Interactive Event Manager" for Real-Time Linking and Event-Scripting of Processing Modules

This system is still under intensive technical and conceptual development, but for now the core program goes by the working title "Interactive Event Manager" (IEM). Using this system, an interactive work can be loaded and performed on the basis of two simple text files: 1) a "setup" file containing a list of the required processing modules and 2) an "event list" containing instructions to various modules for each event number in the score. These two text files are the only elements loaded that are specific to a particular system or composition.

The IEM core program is itself made up of three fairly simple modules. The first reads the setup file and instantiates the listed modules (Max/MSP abstractions stored

separately on disk). The second core module controls advancement through the event list, and may be controlled by other event-scriptable modules. The third core module reads the event list and sends variable parameters to the appropriate modules at each event number. Because all module variables and their interconnections are subject to script control from the event list, it is possible to create very complex networks of processing modules, and to instantly reconfigure that network at any point in the score.

3.2 The Setup Document

The setup file follows a simple format, conforming to the requirements of the Max *coll* object: each line includes an index number followed by a comma, a list of variables and their parameters, followed by a semicolon. To add a particular module to a project, the keyword “module” is followed by the module’s file name and a unique identifier name to be used in the event list as a target for sending variable parameters. A hypothetical setup file might look like this:

```
1, module IEM-soundin.abs mic1;
2, module IEM-soundin.abs mic2;
3, module IEM-pitchtrack.abs pt;
4, module IEM-SMFplayer.abs smf;
5, module IEM-footswitch.abs fs;
...
```

In this example, four modules are created based on the “.abs” files listed. Each of these modules will then be scriptable from the event list using their associated names (“mic1,” “mic2,” “pt,” “smf,” and “fs”). Note that multiple instances of the same module may be called (e.g., “IEM-soundin.abs”). They may be addressed separately from the event list as long as their unique identifiers are, indeed, unique.

3.3 The Event List

Like the setup document, the Event List follows a simple format. Each line is indexed sequentially, followed by an event number, a named module, and a series of variables and their parameters. The event list can be used to initialize module

variables before performance as well as to control system changes during performance. For example, the first few lines of an event list might look like this:

```
1, 0 mic1 ilv 80 olv 80 out bus1;
2, 0 mic2 ilv 64 olv 64 out bus2;
3, 0 pt in bus2 pt-out pt-notes;
4, 0 fs cc 64 out midi-cc1;
5, 0 reverb1 ilv 127 in1 bus1;
6, 1 smf load 1.mid play 1 tp -2;
...
```

In the preceding example, a series of modules is initialized at event 0 (lines 1-5). Variables such as input and output levels (e.g., “ilv,” and “olv”) are defined, as are audio and control data connections between modules. For instance, the module “mic2” has its variable “out” (audio output) set to “bus2” while the module “pt” has its variable “in” (audio input) set to the same value. In this scenario, the output of a microphone (mic2) is now connected to the input of a pitch-tracking module (pt). The output of the pitch-tracker, as a MIDI note number, is then sent out over a channel defined as “pt-notes,” where it may be accessed by other modules that have a control data input variable also set to the value “pt-notes.” These connections may be redefined at any later point in the event list.

The performance begins at “event 1” (line 6) in our example. At this point, the SMF player module (“smf”) loads a standard MIDI file (“load 1.mid”) and plays it at normal speed (“play 1”), transposed down two semitones (“tp -2”).

3.4 The Setup Module

The IEM, as a program, is based on three simple components designed to load and coordinate a complex network of processing modules based on the setup and event list files. The first of these “core modules,” the setup module, is shown in its Max/MSP implementation in figure 1.

Input to the event processor module, shown in figure 4, is received directly from the event manager (figure 2). In the right inlet, an event list is loaded in response to a “read” message, either from the event list or from the user interface. The current event number is received in the left inlet.

3.7 Processor Modules

Each module loaded by the Interactive Event Manager (IEM) is a Max/MSP abstraction with certain standard features allowing it to communicate with the core program and with other loaded modules. The most important feature is the unique identifier. This is set as an argument during the module’s instantiation. Therefore it is accessible in the module abstraction by the variable “#1” which is replaced within the module by the given argument. In this way, multiple instances of the same abstraction can be loaded, and each can be addressed separately using the unique identifiers listed in the setup file. Once a module has been instantiated with a unique name, it must have some method for parsing a list of variable names and their associated parameters. A simple utility for doing so, similar to the one shown in figure 3 for handling event-list cues to the event manager, is shown in figure 5.

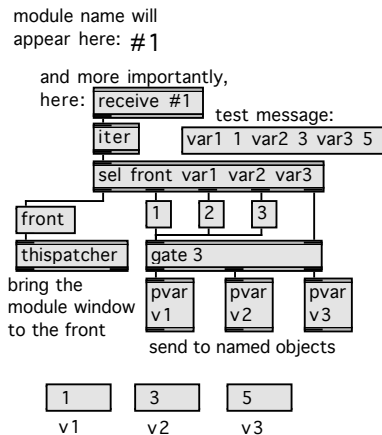


Figure 5 – “Variable manager” utility for receiving and processing variables from the event list

A utility function such as this one can be adapted for the individual needs and

variables of each module used within an IEM network.

Many modules must communicate with one another in the form of audio signals, control data, or other messages related to real-time system control. In order to accomplish this, modules should be constructed so that all data and audio inputs and outputs are sent over named send/send~ and receive/receive~ objects. Furthermore, the names of these send and receive ports should be dynamically scriptable from the event list in order to support communication and audio network reconfiguring on the fly.

An example footswitch controller module is shown in figure 6. Scriptable elements include the MIDI controller number to capture from an external source, the footswitch type (single trigger or on/off state), and output channel for the MIDI footswitch continuous controller data values.

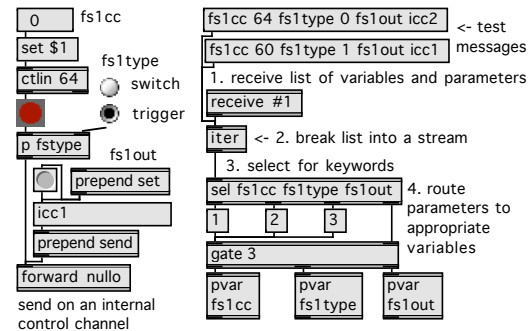


Figure 6 – Footswitch controller module

By default, the footswitch output is routed to “nullo” (null output) – effectively turned off. In the event list, the footswitch module can be routed to the event manager’s default input with the message “fs1out nextevent.” Otherwise, it may be routed elsewhere by assigning a different value for the variable “fs1out,” thereby creating a new data control channel. In order to connect this module to another, the event list might include lines similar to the following:

```
1, 1 footswitch1 fs1out ccl;
2, 1 delay holdswitch ccl;
```

Therefore, communication channels between modules may be defined, or redefined, at any point in the event list.

Processing modules that receive and send audio signals are treated similarly. Audio channels for communication between modules are defined in the event list as module variables. Therefore, each Max/MSP abstraction used as an IEM module must make the audio inputs and outputs accessible to scripting. An audio mixing module is shown in figure 7.

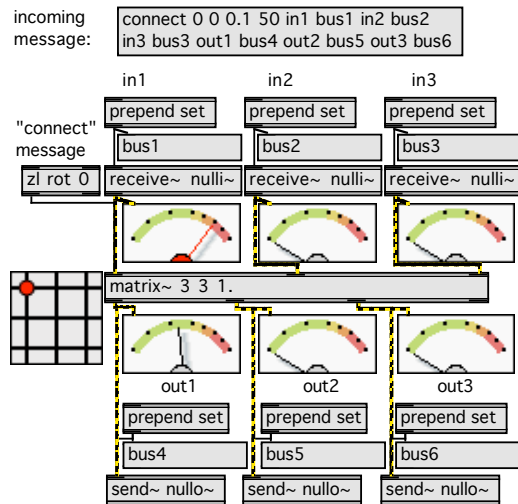


Figure 7 – Matrix mixer module

In this simplified matrix mixer module, scriptable variables include audio inputs and outputs (assignable to named audio busses) and connections between them with variable gain and ramp time

Like the internal control channels, audio connections can be made between modules on the fly via event list scripting. In the event list such a connection might be defined as:

```
1, 1 reverb1 out bus1;
2, 1 matrix8x8 in1 bus1;
```

In this example, therefore, the output (“out”) of a module named “reverb1” is routed to the first input (“in1”) of a module named “matrix8x8.” Both connections are made when the event manager executes event 1. The name given to the audio connection (“bus1” in this example) may be

anything the user likes, and may be shared by as many modules as necessary.

4 Future Plans

The system described so far is fairly simple, but I believe, extremely flexible and powerful. Because new modules only need to conform to a few basic rules, nearly any self-contained Max/MSP abstraction could be assimilated into the IEM framework. This of course opens the door to development of IEM modules by others. IEM modules could also be developed that communicate with other applications, networks, and external devices.

As a test case, I have recently converted several signal processing modules from Cort Lippe’s ISPW pieces for use with this system. Lippe’s modules can be quite complicated, but they are usually controlled from the event list using only a few variables. For example, the reverb patcher used in his *Music for Clarinet and ISPW* has three variables: input level (“Rgate”), output level (“Rout”), and feedback level (“Revfb”). To adapt Lippe’s reverb to the IEM system, the necessary modifications were minor. Inlets and outlets for audio signals were replaced with assignable *send~* and *receive~* objects, and a “variable manager” was added to parse incoming commands from the event script. Inputs for the three named variables were then connected to the variable manager, enabling event-list control.

Converting parts of a composer’s performance software for use in a generalized system has some interesting implications. In Lippe’s original version, the audio inputs and outputs are “hardwired” to other subpatchers, and variables are addressed globally from the event list. As an IEM module, Lippe’s reverb can be used in multiple simultaneous instantiations, with audio i/o connections and variable parameters accessible in real time from the event list. Therefore, it becomes possible not only to re-use, but also to reconfigure and expand a complex interactive system for further exploration in new compositions.

This, in turn, raises some rather provocative questions regarding ownership of the means to perform a given work. How synonymous, exactly, are the composition and its instrument? Should they be separated, with control over the instrument's maintenance and development passing into the hands of performers and technicians? It seems prudent at least that adaptations of existing performance software into a new system should be done in collaboration or consultation with the composer whenever possible.

Because this project began as an effort to develop a portable and maintainable repertoire, my current goal is to develop this system to the point that it can assimilate the technical requirements of five or six works for clarinet and interactive electronics in which I have invested the most time analyzing, reconstructing, and performing. Obviously, a great deal of development is needed in the areas of graphical user interface, real-time visual feedback of module activity, and general robustness for stage use. In the meantime, I am in the process of stocking the system with useful modules that do all the menial tasks such as setting up sound i/o, and defining MIDI controllers. My previous work, re-creating otherwise obsolete interactive systems, has now become a fertile source for creating numerous general-purpose processing modules. I hope collaborate with composers and other researchers as much as possible to continue adapting existing works to this system. Furthermore, I hope to inspire composers to use this system in the creation of new works. Not only does this framework free the composer from having to design an entire interactive system from scratch, but it also gives the performer a system that is adaptable, re-usable, and somewhat transparent. Ultimately, the goal of such transparency is to make interactive electroacoustic works accessible to a larger pool of interpreters. By making such works more available to performers (and therefore to audiences), over longer periods of time, the larger musical community has a better chance to hear and understand new works

that rely on complex interactive electronics. By playing such works repeatedly over time, and by hearing other musicians perform them as well, performers are in a better position to evaluate each work in terms of its musical merits and its viability within a "standard repertoire".

5 References

- Kramer, Jonathan D. 1977. *RENASCENCE, for B-Flat Clarinet, Tape Delay System, and Prerecorded Tape*. Score.
- Musgrave, Thea. 1989. *Narcissus, for Solo Clarinet in Bb with Digital Delay*. London: Novello.
- Pennycook, Bruce. 1991. Machine Songs II: The PRAESCIO Series — Composition-Driven Interactive Software. *Computer Music Journal* 15 (no. 3): 16 – 26.
- Wetzel, David B. 2006. Reconstructing Jonathan Kramer's *RENASCENCE*. Paper presented at the Spark Festival of Electronic Music and Art, University of Minnesota, Minneapolis, MN.
- . 2004. Analysis, Reconstruction, and Performance of Interactive Electroacoustic Works for Clarinet and Obsolete Technology: Selected Works by Musgrave, Pennycook, Kramer, and Lippe. DMA Dissertation, University of Arizona, Tucson.