Shawn L. Decker, Gary S. Kendall, Brian L. Schmidt, M. Derek Ludwig, and Daniel J. Freed
Computer Music Studio
School of Music
Northwestern University
Evanston, Illinois 60201 USA

A Modular Environment for Sound Synthesis and Composition

Introduction

As our knowledge of sound synthesis grows, it becomes increasingly apparent that no single synthesis strategy can create the wide range of musical timbres desired by composers. Similarly, as we gain experience creating compositional interfaces to synthesis programs, it becomes clear that no single musical input language or user interface can adequately accommodate a wide range of compositional styles and intentions. Thus, the attempt to satisfy these musical demands with a single general-purpose synthesis language fails not only because such programs cannot meet the increasing needs of today's composer, but because they sacrifice efficiency and power for breadth and generality. The need for new strategies becomes obvious when one considers that notions about synthesis and compositional interfaces keep changing year by year and that a great deal of software is constantly discarded.

Synthesis Programs

Traditional Synthesis Environments

Since the earliest attempts at waveform synthesis by computer, composers and researchers have devised a large family of algorithms for generating and modifying musical sound. While each specific algorithm is useful and efficient at producing a particular class of timbre, no one synthesis algorithm is capable of creating the widest range of timbres. Indeed, there is good reason to believe that a totally general approach to timbral synthesis is not feasible at all. While each synthesis algorithm presents the composer with a different set of potential attributes to be exploited compositionally, a general approach to the compositional treatment of timbre seems equally unlikely.

The traditional software environment for sound synthesis centers on a single large program to which the composer supplies both *instrument definitions* (synthesis algorithms) and *scores* (synthesis parameters through time). While these programs have been successfully used by many composers, their usefulness is quickly diminishing as new and more complex algorithms are devised for synthesis and processing. For example, monolithic synthesis programs are awkward in interactive contexts. This is especially evident in processing prerecorded sound samples and data supplied by analysis methods. In addition, they offer no potential strategy for bridging the gap between real-time and non-real-time synthesis.

Monolithic synthesis programs suffer not only in conceptual terms but in practical terms as well. Even in software synthesis, the need to execute different algorithms together often forces inappropriate organization on some algorithms. For instance, the sample-by-sample orientation of some programs is inappropriate to block-oriented processes like convolution by the fast Fourier transform (FFT). Likewise, the efficiency of an algorithm like Karplus-Strong (Karplus and Strong 1983) is virtually destroyed when the working values of pointers and past samples have to be saved and restored upon every return and exit to the algorithm. Although synthesis programs can expand to include every new synthesis algorithm, the task of maintaining a single program so large as to incorporate all of these things is extremely difficult. Prototyping new synthesis algorithms will involve recompiling large

Computer Music Journal, Vol. 10, No. 4, Winter 1986,
© 1986 Massachusetts Institute of Technology.

sections of code and necessitates familiarity with the structure of the whole program. As the program gets bigger, fixing bugs becomes more and more difficult. The author of a large program will probably be reluctant to go "back to the drawing board" when design problems do not become apparent until the program has been in actual use.

A related problem is that general-purpose score languages are at the same time too general and not general enough. The format of the score is often too general in that it forces the composer to encode one style of music (say, a traditional score in common practice notation) in a language which is intended to allow as many musical styles as possible. The problem is particularly apparent in the case of the beginner who is forced to learn an entirely general language just to do one simple musical task. This same language, however, is often not general enough to support all possible musical styles in a reasonably effective way, or, for instance, to allow the user both sonic and structural representations within the given score. Because these score languages provide the only standard for musical representation and yet are very inadequate to the task, synthesis environments generally contain very little software that is related to composition and the manipulation of musical materials. What little software there is tends to get thrown away after being used for one composition.

A Modular Synthesis Environment

Our approach to solving these problems has been to develop a highly modular software environment for sound synthesis within the constraints of the UNIX[1] operating system (Decker and Kendall 1984). Instead of providing composers with a single synthesis program, we have provided an environment that supports a collection of single-purpose synthesis programs, each designed to execute a specific synthesis or signal-processing task. This concept was influenced by the CARL software package (Moore 1982), especially the modular signal-processing tools which, in our experience, are the most successful part of the package. Working within this environment is conceptually different from working with traditional synthesis programs in that the composer is not restricted to synthesizing all the component sounds in a composition at the same time. A composer can interactively create short sections, experiment with post-processing and rely on mixing and editing to assemble the final version of the composition.

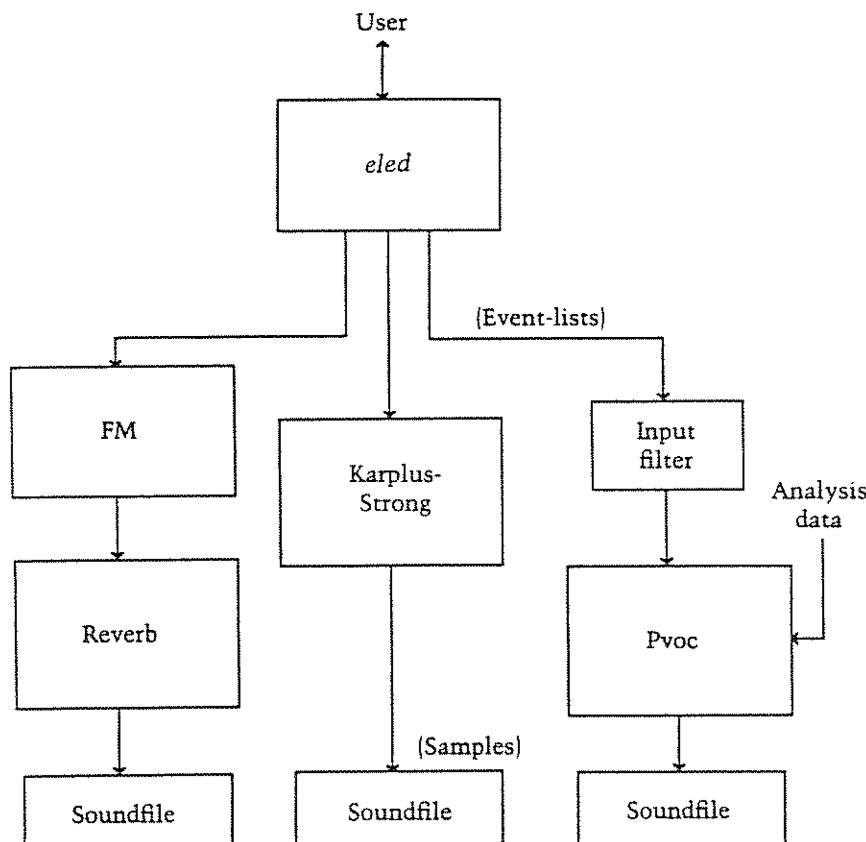
There are a number of practical advantages to providing such an environment:

1. Synthesis routines and composition tools can exist as independent programs implemented in the manner that is most efficient and appropriate to the task.
2. Because synthesis and composition programs are not bundled together in large packages, the repertoire of programs can grow gracefully through the addition of new synthesis algorithms or new composition tools.
3. Composition programs are independent of the particular synthesis algorithms and also independent of whether the synthesis is performed by hardware or software. Thus, it is possible to provide a unified approach to real-time and non-real-time activities.
4. Individual composers with unique needs can customize the environment by adding their own programs, but they need not create their composition software entirely from scratch.
5. Special categories of users (for instance, beginners in sound synthesis) can be given software environments with user interfaces and synthesis programs that are tailored to their needs and capabilities.
6. It is possible to substitute new hardware for synthesis programs or for compositional input routines. This allows new hardware devices to be integrated easily into the software environment.

Within our environment, each synthesis algorithm and signal-processing technique is captured in a standalone program. Such programs exist for additive synthesis, simple frequency modulation (FM),

1. UNIX is a trademark of Bell Laboratories.

Fig. 1. Example of synthesis configuration with a user controlling FM, Karplus-Strong, and phase-vocoder synthesis through the eled editor.



Karplus-Strong, digital filtering, reverberation, and so forth. These programs can easily be connected in series by a UNIX facility called a *pipe*. When UNIX programs are "piped" together, they are executed concurrently with data output from the first being sent to the input of the second, the second to the third, and so on. Thus, the output of a frequency modulation program can be sent to a filter and the output of that sent to a reverberator simply by piping these programs together. UNIX also provides a facility for branching a single stream of data into multiple streams, and conversely, for combining multiple streams into one stream. Although pipes are slower than traditional synthesis programs at passing data from one processing unit to another, we believe that the flexibility of pipes more than compensates for any loss of speed (see Moore 1981). Pipes allow composers to experiment quickly with different combinations of synthesis modules and enable them to achieve compositional goals more quickly. The most frequently used signal-processing and synthesis combinations are eventually implemented as single programs.

Providing a Data Format for Modular Synthesis

A central feature of our modular software environment is the setting of standards to ensure that all compositional programs can communicate among themselves and with synthesis programs. The principal standard within our environment is the data structure for the representation of scores or other time-oriented information called *event lists* (Decker and Kendall 1985). Event-lists generalize and broaden the concept of a score to include any temporally or-

ganized *events*, which are characterized by a list of attributes, including such diverse items as MIDI notes for a synthesizer, mixing commands, or directives to compositional programs. The data structure is adaptable to such a wide range of time-oriented tasks because it imposes no assumptions on the format of the data and because it is "self-descriptive." Each type of event referenced within an event-list is described by a *field-descriptor* that is included as part of the event-list. For example, when a composer requests a MIDI-controlled synthesizer to perform an event-list score that contains only events for software synthesis, the MIDI software is able to use the information contained in the field-descriptor to decode the events and produce the best performance it can manage. Field-descriptors help insure that mature software can recognize and manage new types of event-list data as they develop.

Another important characteristic of the event-list format is that it permits many different kinds of events to be intermixed. A composer can maintain a single unified score that combines compositional procedures, synthesis parameters, and mixing information. Each field-descriptor contains the name of the program for which its particular type of event is intended; a given synthesis program will most likely ignore events that are not intended for it. In general, when the synthesis programs are executed, the events that are intended for each specific program are directed to it alone. The samples generated by each program can either be summed together into one soundfile, or may be stored as individual soundfiles to be edited and mixed at a later time. The original event-list may itself contain the mixing commands.

A block diagram of typical interconnections between programs is shown in Fig. 1. An event-list editor is being used to create a score that references a number of synthesis modules. To realize this score, the editor separates the event-list into three streams: one going to an FM synthesis program whose output is then "piped" through a reverberator program, one going to a Karplus-Strong algorithm, and the third going to a phase-vocoder resynthesis program. The stream of samples from each synthesis program is stored in a separate soundfile and will be combined later using an event-list driven soundfile mixing program. The phase vocoder program is a good example of how easy it is to integrate programs written at other facilities into the event-list environment. The pvoc program (Dolson 1983) written at the Computer Audio Research Laboratory (CARL) at the University of California, San Diego has been integrated into this environment by the addition of a program which translates event-list data into ASCII commands to the phase vocoder.

Programming for Software Synthesis

The programming of standalone synthesis modules is supported by *inslib* (Freed 1986). *inslib* provides a traditional framework for building synthesis instruments from a collection of *unit generators.* An *inslib* "instrument" will read an event-list from its input and will generate digital samples of sound. The *inslib* package consists of a library of routines written in the C programming language that can be used in whole or in part to program synthesis algorithms. *inslib* provides a main routine that interprets the event-list input, starts and stops notes (allowing for arbitrary numbers of simultaneous notes), maintains space for variables, and performs all other generic aspects of synthesis. The user provides a set of C routines that call upon unit generators to synthesize a single note. The events are generated in time-sequential order, so that the output can be piped into a soundfile or a signal-processing program.

inslib instruments have several advantages over interpreted instrument definitions such as those found in general-purpose synthesis programs like *Cmusic* (Moore 1982). Because instruments are written in a full-fledged programming language, the user can easily mix code written in C with calls to *inslib*'s library of unit generators. Compiled instruments are more efficient, and since instruments are written to perform only one specific algorithm, it is possible to optimize both execution time and memory space. Once the instrument is programmed and debugged it is extremely efficient, but creating the instrument will require more time and knowledge from the user than writing an interpreted instrument definition in Cmusic.

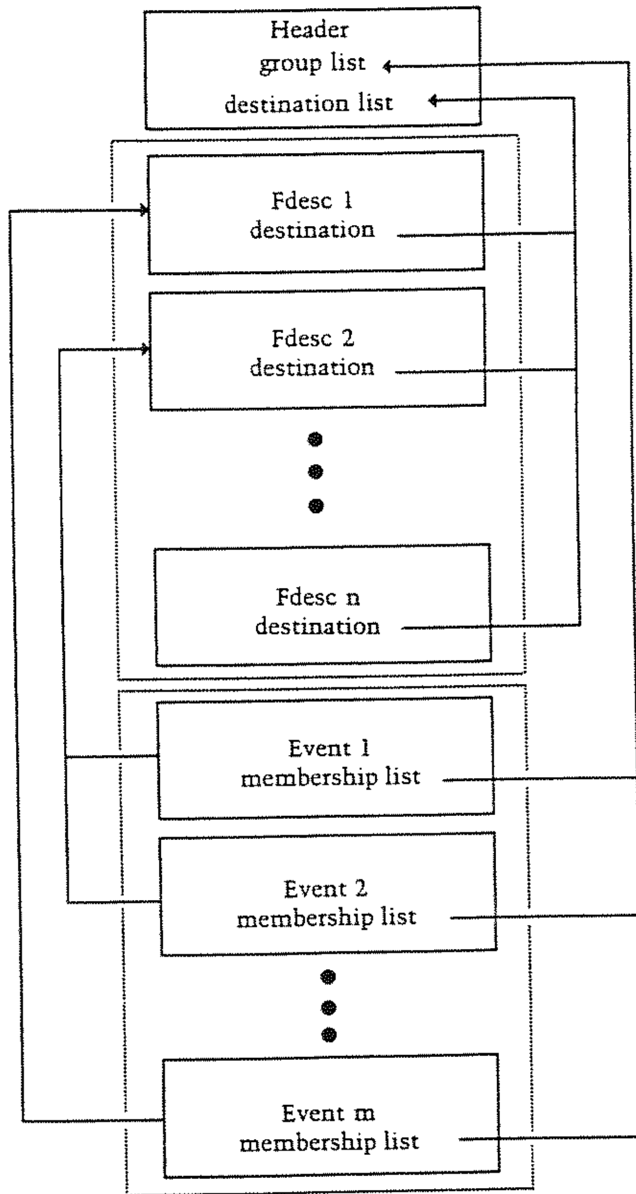


Fig. 2. Event-list *organization.*

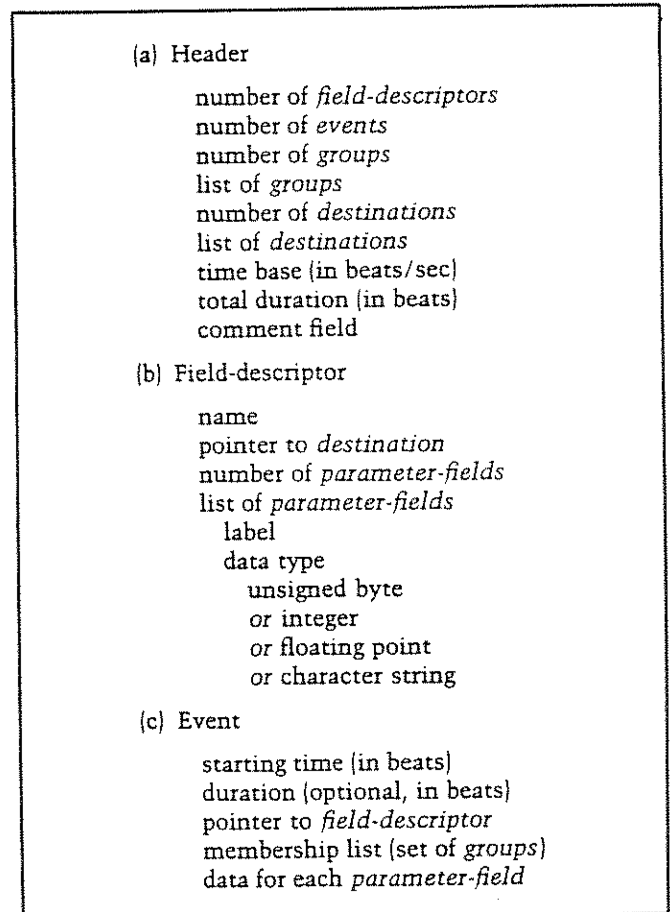


Fig. 3. Event-list *structures for headers, field-descriptors, and events.*

The Event-List Data Format

Each event-list contains three types of structures, which are organized as shown in Fig. 2. The *header* structure (Fig. 3a) contains the general attributes of the list, including the number of field-descriptors and events that are in the list, a list of the programs needed to execute this event-list, and a list of user-defined group names to which events in the list can belong. When loaded into memory, the header also contains pointers to a linked-list of field-descriptor structures and to a linked-list of event structures.

The field-descriptor structure (Fig. 3b) is a template that defines a specific type of event and the format of the data for that event. Data for each event is contained in one or more *fields*, each field being one of four data types: a byte, an integer, a floating point number, or a character string. For each specific type of event, the field-descriptor contains the name of the type of event, the number of fields, the data type for each field, and a label that describes the function or purpose of each field. The field-descriptor also contains the possible "destinations"

Fig. 4. Yamaha DX7 field-descriptors *(a) and* event-list *(b).*

(a) Field-descriptors

Starting-time	Duration	Name	Parameter-fields
stime	dur	**mn**	m_chan$_{UNSIGNED}$ pitch$_{STRING}$ velocity$_{UNSIGNED}$
			mn (MIDI note) will play the *pitch* with the specified key *velocity* on MIDI channel *m_chan* at *stime* for *dur* beats.
stime	dur	**mv**	m_chan$_{UNSIGNED}$ voice_name$_{STRING}$
			mv (MIDI voice) will load the specified *voice* to all the instruments on MIDI channel *m_chan* at *stime.*
stime	dur	**mt**	st_val$_{UNSIGNED}$ end_val$_{UNSIGNED}$
			mt (MIDI tempo) will set the tempo to *st_val* at *stime.* If *dur* is non-zero, then an accelerando (decelerando), from *st_val* to *end_va,* is performed.

(b) Events-list

0.00	3.00	mt	65	75	
0.00	0.00	mv	1	tub_bells	
0.00	0.00	mv	2	voices	
0.50	0.50	mn	1	D#(−1)	60
1.00	4.00	mn	2	A(1)	80
1.00	4.00	mn	2	Cs(1)	80
1.00	0.66	mn	1	E(0)	68
1.67	0.66	mn	1	C(0)	65
2.33	0.66	mn	1	B(0)	63
3.00	7.00	mn	2	Bb(0)	95
3.00	7.00	mn	2	D(0)	95
3.00	0.50	mn	1	Df(−1)	60
3.00	0.50	mn	1	F(0)	67

of this particular type of event, that is, the names of synthesis modules to which this type of event can be routed.

The principal part of the event-list is the list of events (Fig. 3c). Each event has a starting time, an optional duration, and a pointer to the field-descriptor that describes the format of the event. The data for the event is a list of fields, which each contain the type of data specified by the field-descriptor. Nothing is presupposed about the scope or contents of a type of event, other than that it has a discrete starting point in time, and that it follows the form prescribed by the field-descriptor. An event also contains a membership list for the groups listed in the header. Each event may belong to an arbitrary number of groups. The user usually assigns events to a group when those events are created or when those events are selected for manipulation.

We have chosen this very simple grouping mechanism over other methods of organization because it makes the minimum number of assumptions about the information stored within the event-list. In particular, we have avoided explicitly hierarchical groupings within event-lists. Hierarchical operations can still be performed on the event-list by intermediate programs that interpret the groups in a hierarchical fashion.

A field-descriptor is typically developed for a specific piece of synthesis software or hardware. Figure 4 shows a representation of the field-descriptors for

the Yamaha DX7 followed by a list of events. Once the field-descriptor has been created, it can be added to a library of field-descriptors. Individual or multiple libraries can be loaded by any program that manipulates events. For example, we have established field-descriptor libraries for our most commonly used forms of synthesis including the Yamaha DX7 and TX816 under MIDI control and the various software synthesis and soundfile mixing modules written with the *inslib* package.

For anyone wishing to manipulate event-lists within their own programs, a library of routines supports both sequential read/write and memory-buffered linked-list manipulations. The sequential access routines are used for programs that process one event at a time (for example, most synthesis modules), and the linked-list routines are used for programs manipulating an entire event-list (for example, an editing program). It is important to be able to convert data in event-list format to and from a simple character representation both for debugging purposes and for users who wish to write simple character-oriented C programs. These conversions are supported by both routines written in C and by standalone programs.

The *eled* Editor

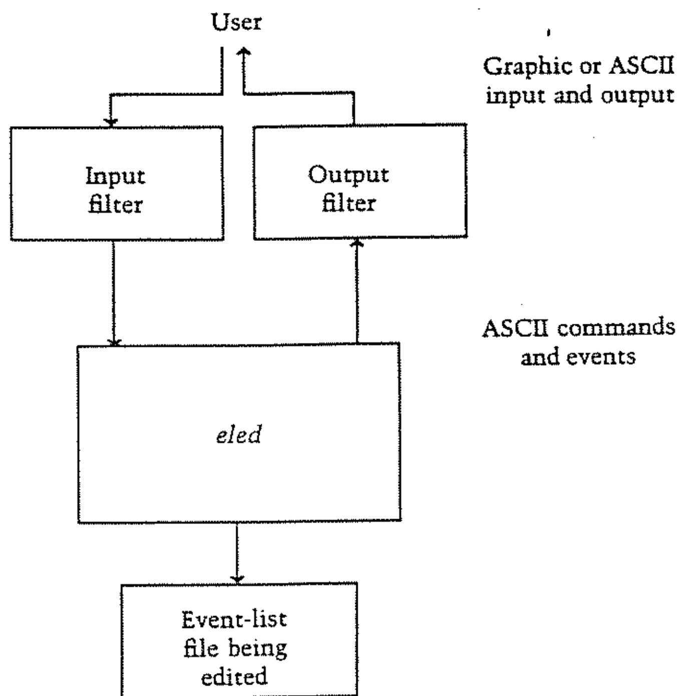
The manipulation of event-lists is performed with an event-list editor called *eled* (Decker and Kendall 1985). We decided to design a single editor that could be tailored to the requirements of many specific applications because of the numerous advantages provided by a unified editing environment. First, an editor that adapts to different contexts reduces the amount of software effort required to support a wide range of applications. Second, a single editor provides the user with a unified approach to a variety of common tasks that might otherwise each require a unique orientation. Finally, because the event-list format is applicable to both real-time and non-real-time operations, one common editor gives the user a single entry-point into both hardware and software synthesis, allowing for the graceful movement from non-real-time to real-time activities.

Considering the wide range of applications to which event-lists are applied, there are numerous issues that a multipurpose event-list editor must address that are not common to single-purpose editors. First, the choice of how to represent events to the user should be dependent on the nature of the application and the user's particular approach to the application. For example, the graphic representation of mixing scripts must be entirely different from that of musical scores. Second, the set of editing commands itself should be also dependent on the nature of the events and the intentions of the user. Even in the case of musical scores, editing commands must be tailored to different compositional approaches and to the composer's own notion of the structure of the music. Third, the need for interactive auditioning of the material being edited is important to many applications and crucial to some (soundfile mixing, for example). The editor must be able to invoke any of the potentially numerous real-time programs which might be able to "perform" the event-list. If the material is unrealizable in real-time, the editor should be able to start software synthesis programs to generate a soundfile that the user can listen to later. Last, the editor must provide the user with a variety of means for selecting and grouping events based on the structural features of the material being edited. The means of selecting and grouping events must be useful in a broad range of contexts and not impose a restrictive strategy on the user.

Design Considerations

Given our experience with a modular synthesis environment that makes extensive use of interprocess communication, we designed *eled* to rely on a variety of external programs. By itself, *eled* supports only a basic facility for ASCII editing. Events and field-descriptors are represented to the user as simple lines of text. The editor implements only a handful of terse commands, including commands for adding, deleting, modifying, playing, and recording events. Generally, the user invokes *eled* with one of a variety of external programs that provide alternative representations of event-lists. This external program becomes a filter that can map events

Fig. 5. Using eled *with input and output filters to modify the user's editing environment.*



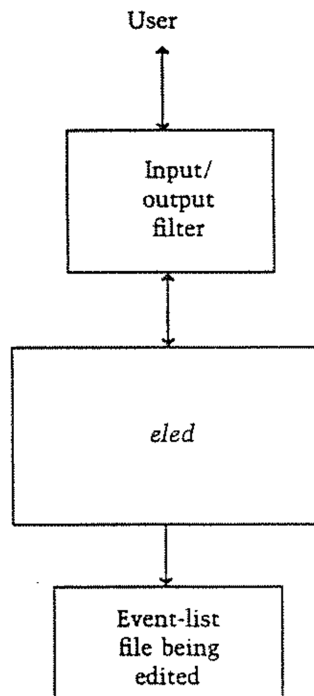
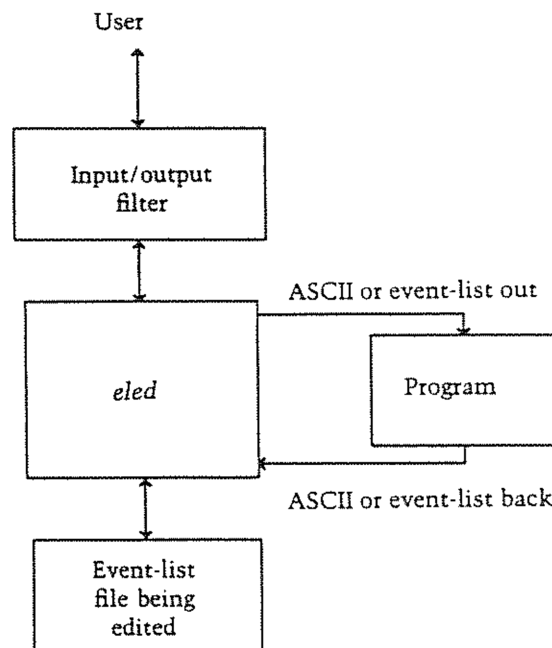


Fig. 6. Using an external command within eled *to modify an* event-list.

Fig. 6



supplied by *eled* into the chosen representation and return changes made within this representation back to *eled.* Figure 5 shows two ways in which input and output filters can be coupled with the *eled* editor. Within the framework of a multiple-window graphic workstation, it is possible to provide multiple representations of an event-list simultaneously. This is particularly useful in score editing, for example, when a graphic representation displays pitch and duration and an ASCII representation displays other synthesis parameters.

The *eled* editor uses a similarly modular approach in providing extended editing commands. The user can invoke *eled* with commands that it transparently filters through external programs. Figure 6 shows a block diagram of *eled* passing events to and from an external program in response to a user command. This capability enables the user to employ various external programs to implement commands exactly as if the commands were a natural part of the editor. Users can easily create new

Fig. 7. *Examples of* elex *syntax.*

Example operators and functions

!*x*	Logical negation of *x*.
num *x*	TRUE if *x* is a number.
null *a*	TRUE if *a* is a zero-length string.
frac *x*	The fractional part of *x*.
sin *x*	Sine of *x* radians.
cls *a*	The pitch class of *a*.
x or *y*	Logical or.
x == *y*	Logical equality.
a + *b*	The concatenation of *b* onto *a*.
m ? *n*	A random integer from *m* to *n*.
x deg	Converts *x* from degrees to radians.
x db	Value of *x* in decibels.

Event values

st	Starting time (in beats).
dur	Duration (in beats).
fdn	*Field-descriptor* name.
p*n*	Value of the *n*th *parameter-field*.
p'*a*'	Value of the *parameter-field* labeled *a*.

Event constraint functions

f'*a*'	TRUE if the *field-descriptor* name is *a*.
m'*a*'	TRUE if the event is a member of group *a*.
E p'*a*'	TRUE if the event has a *parameter-field* labeled *a*.

Constraint expression examples

(dur >= 5.0 and p1 == 3)
Only events with a duration of 5 beats or more and a value of 3 in the first *parameter-field*.

(frac(st)! = 0 and p'velocity' > 42)
Only events that are not on a beat and which have a key velocity greater than 42.

(f'mn' and cls p'pitch' == cls'c' and p'm_chan' == 3)
Only events of type *mn* (MIDI note) which play a C (any octave) on MIDI channel 3.

commands by writing their own "editing" programs and groups of these programs can be bundled together to serve as "command sets" for specific applications. The problem of auditioning the event-list is handled easily in the same manner by implementing the play command as an external program that receives an event-list to perform. We have implemented our record and overdub commands by means of an external program that simultaneously plays the given score and reads new events performed on a MIDI keyboard. The technical means for playing the score can transparently change from one application to the next, and other hardware input devices (either real-time or non-real-time) can be incorporated in a similar manner.

Selection of Events

eled makes use of a general purpose expression evaluation routine called *elex* (Ludwig 1985) for selecting events and modifying the contents of data fields within events. *elex* provides for the logical and arithmetic evaluation of expressions, and contains useful postoperators and string-matching routines. By adding variables that substitute the starting time, duration, field-descriptor type, group list, or the value for any given parameter field into these expressions, event selection can combine any number of conditional expressions involving any of these parameters. The editor allows the user to select a group of events by evaluating this logical expression for each event within a time window in the score with each event's starting time, duration, etc. substituted for any variables that appear in the expression. If the result is a logical FALSE, the event fails the constraint expression and no action is performed on it. Figure 7 shows a few of the operators supported by the *elex* program, the variables associated with event parameters, and a few examples of constraint expressions.

The use of *elex* as a means of selecting events provides more advantages than disadvantages. First, it supplies a means of "sieving" events through a number of criteria based on any given parameter of the events. In practice, it is more flexible than simply passing events through sieves as it allows for arbitrary AND and OR expressions to be constructed. Second, the ability to use string matching, arithmetic and relational expressions, and a host of other operators improves the flexibility of our sieve

Fig. 8. Example of eled *session. The user input is in* **boldface** *and* eled *output is in roman. The right-hand column contains descriptions of the editing session.*

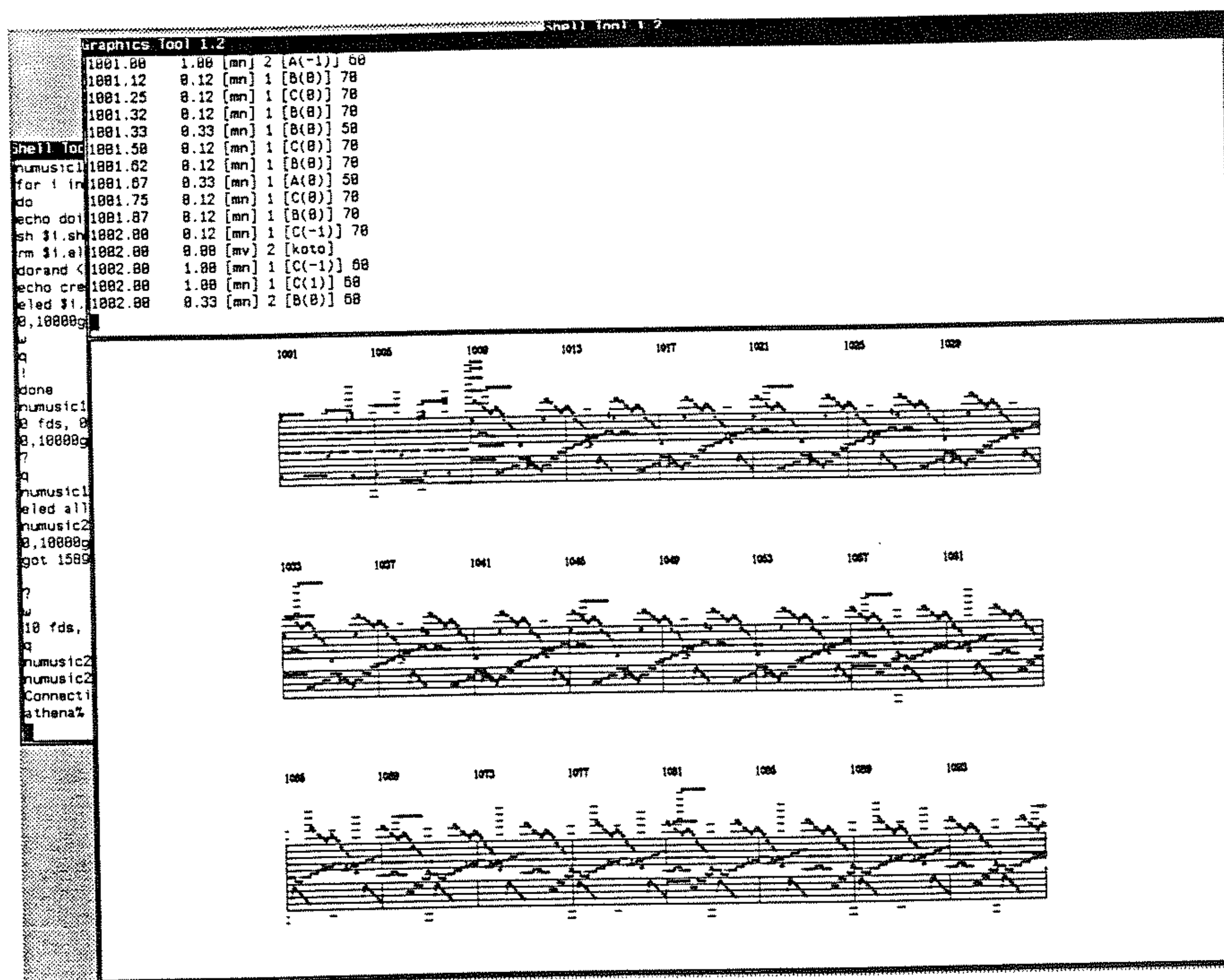
Session	Description
t	t is the *type* command. Type the entire *event-list*.
0.00 3.00 mt 65 75	
0.00 0.00 mv 1 tub_bells	
0.00 0.00 mv 2 voices	
0.50 0.50 mn 1 Ds(-1) 60	
1.00 4.00 mn 2 Cs(1) 80	
1.67 0.66 mn 1 C(0) 65	
2.33 0.66 mn 1 B(0) 63	
3.00 7.00 mn 2 Bf(0) 95	
done	
2,3d	d is the *delete* command. Delete the *events* between beats 2 and 3.
done	
a	a is the *add* command. Add new *events* to the *event-list*.
2.50 4.00 mn 2 Gf(2) 95	
3.50 0.70 mn 1 B(2) 60	
.	The '.' terminates the add command.
done	
0.5,$t[p'velocity' >= 65]	Type the *events* from beat 0.5 to the end of the list ($) that have a key velocity ≥ 65.
1.00 4.00 mn 2 Cs(1) 80	
1.67 0.66 mn 1 C(0) 65	
2.50 4.00 mn 2 Gf(2) 95	
done	
1.67e[p'm_chan' == 1](p'velocity' -= 10)	e is the *evaluate expression* command. Reduce by 10 the (key) velocity of the midi-note *event* on channel 1 at beat 1.67, by evaluating the expression within the paranethuses.
done	
r[f'mn' and p'm_chan' == 1](trans 6)	r is the *filter/replace* command. Filters the selected *events* though an external program specified within the parethses. In this case, the program "trans" will transpose all the midi-note *events* on channel 1 up six semitones.
done	
t[f'mn' and p'm_chan' == 1]	Type all the midi-note *events* on channel 1.
0.50 0.50 mn 1 A(-1) 60	
1.67 0.66 mn 1 G(0) 55	
3.50 0.70 mn 1 F(3) 60	
done	
1,3r[p'm_chan' == 2](rpts 3 4)	Repeat the midi-note *events* between beats 1 and 3 on channel two with three repeats (including the original), each starting 4 beats after the previous. New *events* are marked below with a ⇐.
done	
t	Type the entire *event-list*.
0.00 3.00 mt 65 75	
0.00 0.00 mv 1 tub_bells	
0.00 0.00 mv 2 voices	
0.50 0.50 mn 1 A(-1) 50	
1.00 4.00 mn 2 Cs(1) 80	⇐ (original)
1.67 0.66 mn 1 G(0) 55	
2.50 4.00 mn 2 Gf(2) 95	⇐ (original)
3.50 0.70 mn 1 F(3) 60	
5.00 4.00 mn 2 Cs(1) 80	⇐
6.50 4.00 mn 2 Gf(2) 95	⇐
9.00 4.00 mn 2 Cs(1) 80	⇐
10.50 4.00 mn 2 Gf(2) 95	⇐
done	
p	Play the entire *event-list*.
done	
p[f'mt' or p'm_chan' == 2]	Play only channel 2.
done	
q	Attempt to quit.
?	Error - Cannot exit *eled* when the *event-list* has been modified but not saved. Either write the *event-list* (w) or quit without writing (q!)
w(new.el)	Write the file *new.el*.
done	
q	Quit

scheme even further. Third, because *elex* is an independent program, improvements to *elex* can be made without interfering with *eled*. On the negative side, the syntax of *elex* is certainly not the most straightforward means of specifying musical groupings. It is likely to be more meaningful and intuitive to a mathematician than to a musician. However, the user may never use these expressions if the actual selection of events is performed with the aid of an external program. For example, if the events are selected graphically, the actual constraint expression will be constructed by this external program rather than by the user. Generality has been favored over ease of syntax for this reason.

eled Examples

Figure 8 illustrates a simple editing session using *eled* without either an input or an output filter. Each command is a single letter, which is preceded by an optional starting and ending time. These times can be used to limit the events affected by the command to those between the specified times. The left-hand column of Fig. 8 represents the dialogue between the user and the editor, the right-hand column contains a brief explanation of each command. The event types themselves are from a field-descriptor library of events for controlling a Yamaha TX816 and a number of Yamaha DX7 synthesizers in real time using a MIDI interface. The example shows the syntax for typing, deleting, and adding events. Also, two external commands are shown: trans, which transposes MIDI events by the number of semitones it is given as an argument; and rpts, which repeats the events it receives the number of times and at the interval given to it as arguments. Note that in the second use of the r command (which stands for "replace") the events are sent through two UNIX programs piped together in series. On our current system, even with large scores, the entire execution of an external command such as these takes, at most, a couple of seconds.

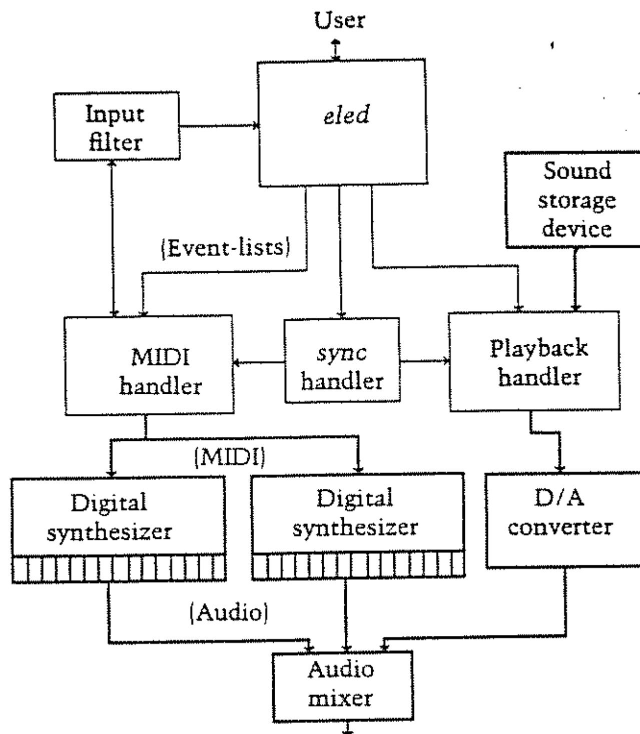
Fig. 9. Example of eled output with a graphic output filter. Output is being presented to the user both in its original ASCII format (top window) as well as in a "piano-roll" format (bottom window).



Users have developed a large number of external programs for editing these MIDI events. Examples of these programs include the transpose and rpts programs shown, programs which rhythmically quantize events entered at a keyboard (or vice versa add $1/f$ noise random jitter to starting times, durations, or key velocities), various algorithmic composition routines (typically written for a specific composition), and filters for mapping psychoacoustically scaled amplitude information into key velocities (Martens 1985).

In Fig. 9 the same event-list score is displayed simultaneously in both a piano roll and a character representation. In this example, the piano roll notation is created by an output filter that takes the ASCII representation used by *eled* and converts it into a graphic representation. Since the two representations are really of the same event-list, a change

Fig. 10. *Real-time example using a MIDI keyboard instrument as input, and using both MIDI and soundfile playback.*



made to one representation will be reflected immediately in the other, enabling the user to flip back and forth, depending on the need of the moment.

Real-time Synthesis with *eled*

One set of external programs developed for *eled* performs real-time recording and playback with MIDI-controlled devices. When configured with these programs, *eled* becomes a real-time, interactive MIDI score editor with playback, recording, and overdubbing capabilities. The external programs either perform an event-list provided by *eled* or record notes performed on a MIDI device as input to *eled*. For example, when a composer is performing on a MIDI keyboard to "overdub" a preexisting score, the hardware synthesis device, a Yamaha DX7, must perform the "overdubbed" notes while *eled* starts the performance of the original score. The overdubbed notes are transmitted back to *eled* and merged into the original score.

Figure 10 shows a typical software patch established by *eled* for real-time synthesis. The synthesis modules are all hardware devices. Because UNIX does not support real-time processing at the user level, each hardware device is accessed through a system-level device-driver. Synchronization of the real-time synthesis devices is performed by the *sync pseudo-device-driver*, which also passes real-time messages from the user-level program to the synthesis device-driver (Schmidt and Roth 1985). The *sync* module is designed to allow all of the synthesis and playback to be synchronized by external MIDI signals from MIDI musical devices and/or external SMPTE (Society of Motion Picture and Television Engineers) time-code signals from SMPTE-encoded video.

Additions to the *eled* Environment

As a computer music program like *eled* becomes more complex and sophisticated, there is a corresponding increase in the amount of expertise required to use it effectively. The combination of *eled* and *elex* provide a versatile but complex "language" for the selection and modification of information within a musical score. However, this language is not immediately accessible to a musician with little to no training in Boolean mathematics or programming languages. As a result, a beginning computer musician must spend a great deal of time learning the system rather than making music.

Many of the activities undertaken by the user with *eled* are fundamentally selection tasks that are best communicated by using the mouse to select menu options or notes in a score. These methods of communication can best be handled by the programs external to *eled*, which implement either specific command sets or specific representations of event-lists. But even when these methods of selection are available, a user's intentions often involve different kinds of manipulations than can be communicated by pointing.

One addition to the *eled* environment is a macro substitution facility with a set of definitions that substitutes macros for common expressions used in editing scores. The macro preprocessor, *elmac* (Schmidt 1985), has become a popular "front end" to *eled*; it captures the user's commands and expands macros before the revised command is sent to *eled*. For nearly a year, students have been using *elmac* in combination with *eled* without being told of its existence and without having to learn the full range of expression syntax interpreted by *eled* and *elex*. They have been able to perform complex score-editing tasks with "high-level" commands that they understand to be part of *eled* itself. For example, students have been able to add and delete notes, transpose motives, and create canons after an hour-long introduction to *eled*. We have gradually been able to expand the repertoire of macro definitions as users have creatively expanded their notions of how to use *eled*. This facility for high-level editing combined with real-time playback over MIDI-based synthesizers has provided an excellent entry-level environment for students as well as a very flexible environment for experienced composers.

No matter how successful *elmac* has been at enabling users to communicate and work in a more intuitive fashion, it has also served to point out the limitations of macro substitution in dealing with the complexities of editing and manipulating music. What is needed is an interface that understands the requests of the musician directly in a language that is familiar, namely a natural language. Thus, a musician should be able to say "Transpose the second half of the bass line up a minor third," instead of having to first translate his idea into a less intuitive representation.

Our second addition to the *eled* environment is a natural language system for musical applications being written in Franz Lisp (Schmidt 1986). It takes typed natural language input and creates appropriate *eled* commands for the access and manipulation of musical structures. Its function is to read user input and verify that it is syntactically and semantically correct in the context of score editing. The system then writes and executes a program based on the analyzed input. For example, for the request, "play the first 10 beats of the flute and bass," the verb "play" carries a specific semantic meaning for *eled*. "The first 10 beats" creates a *time constraint* and "of the flute and bass" creates an *instrument constraint*. Ultimately the program creates the desired *eled* command line and sends it to the score editor. Contextual information is retained during the use of the system. This allows users to specify their requests in a continuing dialogue. For example, the request "add the bass" following "play the piano part" would play the bass and piano.

Conclusion

Our concept of a modular synthesis environment evolved out of many practical concerns. Not the least of these was that it would be implemented within UNIX. We were concerned that real-time and non-real-time synthesis programs would develop without any clear coordination, and that each new piece of synthesis hardware would induce a major programming change. It seems that the rapid advance of hardware has forced us to adapt to being in a state of constant change and to design our software environment with the few stable standards we can ensure. Event-lists provide a standard format for communication among programs that may constantly change. Just the fact that this standard exists allows us to pursue compositional software like the *eled* editor as part of a long-term plan. Without such planning, compositional software is the first component in a computer music system to become obsolete.

The *eled* program has been developed over the last two years and has now stabilized in a mature form. While there is a certain loss of efficiency whenever external programs are used extensively, it is made up for by the extreme flexibility of the editor. Already, several uses of *eled* have arisen out of user-written external filters and commands that were unanticipated by the authors. In particular, real-time MIDI playback from the editor has motivated users to expand the range of compositional manipulations undertaken directly within *eled*. The amount of real-time musical exploration is far greater than we would have thought possible within

a timesharing operating system. Now that most of the work has been done in establishing the core editing routine, we anticipate that much of our energy will be devoted to designing a number of menu-driven graphic editors, using input and output programs and external command sets, for a wide range of applications. Many of the new programs will be written in Franz Lisp.

Acknowledgments

Much of the inspiration for both the modular nature of the environment and specifically the event-list data format came from work done by William Buxton and his colleagues in designing the SSSP system at the University of Toronto (Buxton 1978; Buxton et al. 1978; 1979). In addition, many of the ideas for the editor underwent much invaluable fine-tuning during our numerous brain-storming sessions with the staff at Northwestern Computer Music. Special thanks must be given to Charles Smith and the System Development Foundation whose support for Northwestern Computer Music has made this work possible.

References

Buxton, W. 1978. "Design Issues in the Foundation of a Computer-based Tool for Music Composition." *Technical Report CSRG-97.* Toronto: University of Toronto.

Buxton, W. et al. 1978. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." *Computer Music Journal* 2(4): 10–20. Revised and updated version in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge, Mass.: MIT Press. pp. 433–466.

Buxton, W. et al. 1979. "The Evolution of the SSSP Score Editing Tools." *Computer Music Journal* 3(4): 14–25. Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge, Mass.: MIT Press. pp. 376–402.

Decker S., and G. Kendall. 1984. "A Modular Approach to Sound Synthesis Software." In W. Buxton, ed. 1984. *Proceedings of the 1984 International Computer Music Conference.* San Francisco: Computer Music Association. pp. 243–250.

Decker, S., and G. Kendall. 1985. "A Unified Approach to the Editing of Time-ordered Events." In B. Truax, ed. 1985. *Proceedings of the 1985 International Computer Music Conference.* San Francisco: Computer Music Association. pp. 69–78.

Dolson, M. 1983. "Musical Applications of the Phase Vocoder." In C. Harris, ed. 1983. *Proceedings of the 1983 International Computer Music Conference.* San Francisco: Computer Music Association.

Freed, D. J. 1986. "inslib Manual." Unpublished manuscript. Evanston, Ill.: Northwestern Computer Music, Northwestern University.

Karplus, K., and A. Strong. 1983. "Digital Synthesis of Plucked-String and Drum Timbres." *Computer Music Journal* 7(2): 43–55.

Ludwig, M. D. 1985. "Exprsn Manual." Unpublished manuscript. Evanston, Ill.: Northwestern Computer Music, Northwestern University.

Martens, W. 1985. "Palette: An Environment for Developing an Individualized Set of Psychophysically Scaled Timbres." In B. Truax, ed. 1985. *Proceedings of the 1985 International Computer Music Conference.* San Francisco: Computer Music Association. pp. 355–366.

Moore, F. R. 1981. "Musical Signal Processing in a UNIX Environment." *Proceedings of the International Music and Technology Conference.* Melbourne: University of Melbourne.

Moore, F. R. 1982. "The Computer Audio Research Laboratory at UCSD." *Computer Music Journal* 6(1): 18–29.

Schmidt, B. L. 1985. "elmac Manual." Unpublished manuscript. Evanston, Ill.: Northwestern Computer Music, Northwestern University.

Schmidt, B. L. 1986. "A Natural Language System for Music." Manuscript in preparation. Evanston, Ill.: Northwestern University.

tion of Audio Production in Computer Music." In B. Truax, ed. 1985. *Proceedings of the 1985 International Computer Music Conference.* San Francisco: Computer Music Association. pp. 341–345.