A Modular Approach to Sound Synthesis Software

Shawn L. Decker and Gary Kendall
Northwestern University Computer Music Studio

ABSTRACT

As our knowledge of timbre synthesis grows, it becomes increasingly clear that no single synthesis strategy can create the wide range of musical timbres wanted 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 large range of compositional styles and intentions. Thus, the attempt to satisfy this wide range of musical demands with a single general-purpose synthesis language fails, not only because such programs increasingly cannot meet the 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 our notions about synthesis and compositional interfaces keep changing year by year and that a great deal of software is constantly discarded.

Our approach to solving these problems has been to develop a highly modular software environment for sound synthesis within the constraints of the UNIX† operating system. Central to our approach is a standardized, time-ordered data structure called the *event-list* which enables compositional programs to communicate among themselves and with synthesis programs. The facilitator for this communication is a 'scheduler' program which is able to connect a variety of independent compositional programs, interactive interfaces, and synthesis algorithms. Our implementation of the 'scheduler' concept within UNIX is a program called *sched* that has been in use at the Northwestern University Computer Music Studio to interconnect various user-interfaces with synthesis programs and with real-time, *MIDI*-controlled digital synthesizers.

† UNIX is a trademark of Bell Laboratories.

ABSTRAIT

Plus nous approfondissons nos connaissances en synthèse audio, plus nous réalisons que différentes techniques de synthèse sont ńecessaires afin de créer une large gamme de sonorités. De même, plus nous appliquons l'informatique à la composition, plus nous comprenons qu'un langage unique ou une interface utilisateur ne peuvent s'adapter vraiment à tous les styles ou tous les objectifs de composition. Les languages de synthèse universels ne peuvent assurer tous les modes de synthèse actuellement utilisès et il est souvent difficile de les adapter aux besoins d'un compositeur donné.

Afin de rèsoudrer ces problèmes, nous avons développé un environs du logiciel très modulé. Très important pour notre approche est une structure de données standarisée, intitulé le *event-list*, qui permet les programmes de composition à communiquer entre eux-memes et avec les algorithms de synthèse. Cette communication est facilité par le program *scheduler* qui est capable de joindre plusiers programs de composition, interfaces utilisateurs, et les algorithms de synthèse. Nous l'avons employé sous UNIX avec un programme intitulé *sched.* A l'heure actuelle, ce programme est utilisè au Northwestern Computer Music Studio à connecter plusieurs interfaces utilisateurs avec des algorithmes de synthèse et avec des synthétiseurs temps réel conrolé par MIDI.

1. 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 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. In fact, there is plenty of 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 has been the 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 to today's computer composer is quickly diminishing as new, more complex algorithms are devised for synthesis and processing.

Monolithic synthesis programs suffer not just in conceptual terms but in practical terms too. They offer no potential strategy for bridging the gap between real-time and non-real-time synthesis. 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 FFT. Then too, the efficiency of an algorithm like Karplus-Strong is virtually destroyed when the working values of pointers and past samples have to be stored and restored upon every exit and return to the algorithm. More importantly, monolithic synthesis programs are awkward in interactive contexts. This is especially evident in processing pre-recorded sound samples and data supplied by analysis methods. Will some future version of MUSICX incorporate cross-synthesis by LPC (linear-predictive coding)? Whether the answer is 'yes' or 'no', the task of maintaining a single program so large as to incorporate all of these things is a difficult task. Prototyping of new synthesis algorithms must necessarily involve recompiling very large sections of code and necessitates considerable 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 must be especially reluctant to go back to the drawing board when problems in the design of a program do not become apparent until the program has been in actual use.

Another problem with general-purpose synthesis languages is that the means for specifying the 'score' is at the same time too general, and not general enough. The format of the score is often too general in that it constrains 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 of a given score. Because these 'score languages' provide the only standard for musical representation and are yet 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.

2. 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 operating system. Instead of providing composers with a single synthesis program, we have provided an environment which supports a collection of synthesis programs, each designed to address a specific synthesis task. Instead of providing composers with a single language for representing their score, we have provided an environment which supports a variety of possibilities for creating scores and for modifying them. This strategy has evolved out of experience with the UNIX operating system which has become a de facto standard operating system for major computer music centers. UNIX provides the necessary means for multi-processing and interprocess communication on which the modular software environment for sound synthesis is built. The most recent versions of UNIX also support interprocess communication across multiple CPU's in a local network.

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*. These *event-lists* generalize and broaden the concept of a score to include any chronological list of events and data such as commands to a synthesizer, mixing scripts, or descriptions of compositional processes. 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 this format is self-describing. The form and kind of events in any list is described within the list whether they be digital synthesizer voice definitions, mixer commands, or compositional directives.

The program that facilitates communication between compositional programs and synthesis programs is called the 'scheduler'. The 'scheduler' receives an *event-list* from the compositional program and 'schedules' the synthesis programs or devices that execute its commands. It must also establish the software patch between the compositional and synthesis programs, initiate and monitor execution, collect output, and synchronize the software and hardware components of the synthesis. Our implementation of the 'scheduler' concept within UNIX is called *sched*. *Sched* provides all compositional programs with an identical gateway into synthesis, whether it is performed by hardware, software, or both. It has been used at the Northwestern University Computer Music Studio to connect compositional programs to a range of synthesis programs including stand-alone synthesis programs, shell scripts which pipe sounds through various signal-processing programs, and real-time, MIDI-controlled synthesizers. The traditional synthesis approach of defining instruments with unit generators is still supported by a package called *inslib*. It provides composers with a library of very efficient C-callable routines for constructing stand-alone instruments.

A block diagram of a typical configuration of programs and their interconnections is shown in figure 1. Each box represents either a single program or a number of programs piped together. The 'user-interface' represents one of many possible compositional programs for displaying, editing, or generating an *event-list*. To realize the *event-list*, it is sent to the *sched* program which demultiplexes the events and passes them on to the individual synthesis programs. Each synthesis program executes the list directed to it by *sched* and the output samples are collected by a 'summing' process that most typically stores the combined result in a soundfile for later access. In the case of real-time synthesis, the synthesis hardware has its output directed to an audio mixer.

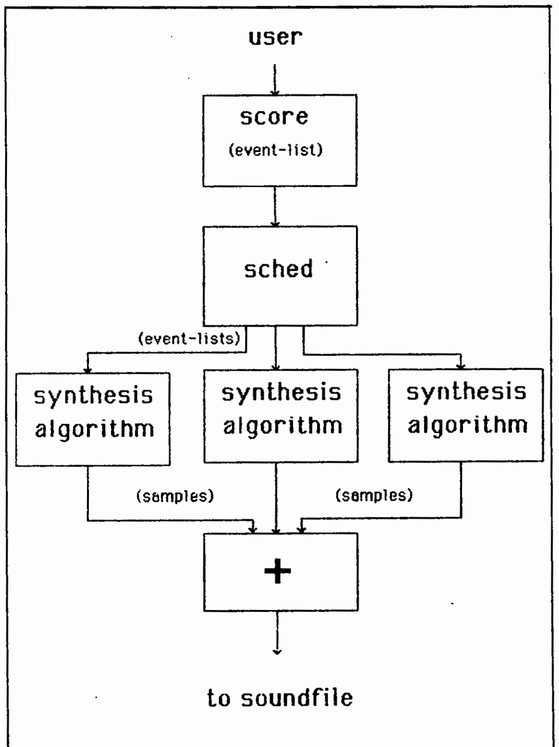

Figure 1. Configuration and data paths.

In addition to the conceptual differences that the modular approach to sound synthesis provides, there are numerous practical advantages. Some of these can be stated as follows:

1. Synthesis routines and compositional tools exist as independent programs implemented in the manner which is most most efficient and appropriate to the task.
2. Because synthesis and compositional programs are not bundled together in large packages, the general repertoire of programs can gracefully grow through the addition of programs which contribute new synthesis algorithms or new compositional tools.
3. Compositional programs are independent of 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. The individual composer with unique needs can modify the environment by supplying a few 'custom-made' programs and need not construct an entire software package from scratch.
5. Special categories of composers (for instance, beginners in computer synthesis) can be given software environments that are expressly tailored to their needs and capabilities.
6. Since any of the software modules can be replaced with computer hardware, it is possible to substitute hardware for synthesis programs or for compositional input routines. This allows new hardware devices to be easily integrated into the environment as they become available.

3. THE *event-list* DATA FORMAT

The primary obstacle to a modular software environment is the standardization of communication among the many different modules. The most common standard in the general UNIX environment is a stream of ascii characters. The standard for signal processing programs is a stream of binary, floating-point numbers. The communication standard for compositional programs is a more complicated issue because information can take on so many different forms and yet must be more structured than ascii or floating-point streams. Our solution to this problem is a standardized data structure called *event-lists*. *Event-lists* assume a notion of temporally organized '*events*' which are each characterized by a list of attributes. Since every program must make some sense of this information, the form of each type of *event* must be described by a *field-descriptor* which is included within the *event-list*.

Each *event-list* contains three types of structures which are organized as shown in figure 2. The *header* structure (figure 3*a*) contains the general attributes of the list, including the number of *field-descriptors* and

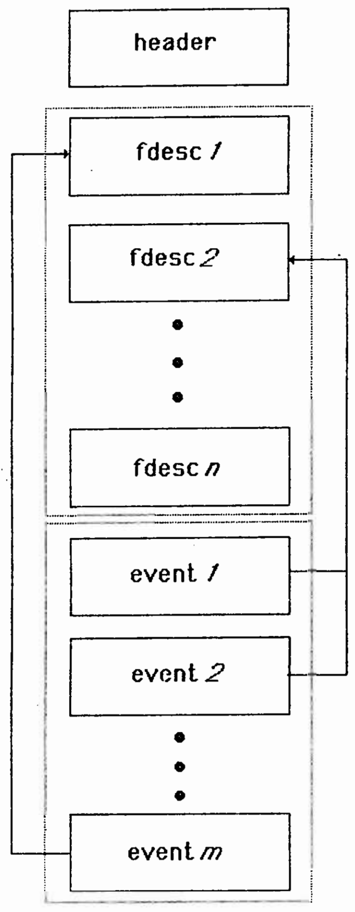

Figure 2. ***Event-list*** **organization.**

events which follow. In the case of the internal linked-list version, the header also contains pointers to a linked-list of *field-descriptor* structures and to a linked-list of *event* structures. Each *event-list* has exactly one *header* structure.

Following the *header* structure is a list of *field-descriptor* structures (figure 3*b*). Each *field-descriptor* is a template used both for defining 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 (either a binary byte, a binary integer, a binary floating point number, or an arbitrary-length, null-terminated character string). The *field-descriptor* for a specific type of event defines both the number of fields and the data type for each field. The *field-descriptor* includes a list of null-terminated strings, one per field, with user-defined labels describing the function or purpose of each field. The *field-descriptor* also contains the 'name' and the 'destination' of this particular type of *event.* The 'destination' is typically the name of the synthesis module for which this type of event is normally intended, and will be used by the *sched* program to route each event to the proper synthesis program.

Following the list of field-descriptors comes the list of *events* (figure 3*c*). Each *event* has a starting time, an optional duration, and the name of the *field-descriptor* (from earlier in the list) which describes the format of the event. The data for the event then follows as a list of 'fields', each 'field' containing data of the type 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 formula prescribed by the *field-descriptor.*

a. header:

- **number of field-descriptors**
- **number of events**
- **total duration of list (in secs.)**
- **time base (default: secs.)**
- **comments**

b. field-descriptor:

- **name**
- **destination**
- **number of fields**
- **data type for each field**

.
.

- **symbol for each field**

.
.

c. event:

- **starting time**
- **duration (optional)**
- **type of event (field-desc. name)**
- **data for each field in mixed format**

.
.

Figure 3. ***Event-list*** **structures.**

Figure 4 shows an ascii representation of the DX-7 *field-descriptors* followed by a list of events. It is important to keep in mind that the information represented here is normally in binary format with the exception of those fields which have been declared as type 'STRING'.

```
field-descriptors:

 (name, symbol, data type, ...)

 midi_note -
          m_chan       INT
          pitch       STRING
          velocity     INT

 midi_tempo -
          start_val    INT
          end_val      INT

 midi_voice -
          m_chan       INT
          voice_name     STRING

events:

 (st. time, dur, f.d. name, data ...;)

 0.00  0.00 midi_voice 2 voices ;
 0.00  0.00 midi_tempo 65 65 ;
 0.00  0.00 midi_voice 1 tub_bells ;
 0.50  0.50 midi_note 1 D#(-1) 60 ;
 1.00  4.00 midi_note 2 A(1) 80 ;
 1.00  4.00 midi_note 2 C#(1) 80 ;
 1.00  0.66 midi_note 1 E(0) 68 ;
 1.67  0.66 midi_note 1 C(0) 65 ;
 2.33  0.66 midi_note 1 B(0) 63 ;
 3.00  7.00 midi_note 2 Bb(0) 95 ;
 3.00  7.00 midi_note 2 D(0) 95 ;
 3.00  0.50 midi_note 1 D#(-1) 60 ;
 3.00  0.50 midi_note 1 F(0) 67 ;
```

Figure 4. Example DX-7 *event-list.*

A *field-descriptor* is typically developed along with a specific synthesis program. Once the *field-descriptor* has been finalized, it can be included in a library of *field-descriptors.* Individual or shared libraries can be loaded along with an *event-list* by any program manipulating events of that type. For example, we have established a number of *field-descriptor* libraries which are used for our most common types of synthesis. These include the Yamaha DX-7 *field-descriptor* library which contains templates for those types of events understood by the DX-7 under MIDI control, and an *inslib field-descriptor* library for those events understood by *inslib* programs.

Standard versions of i/o routines which support both sequential access and in-memory linked-list data are provided for composers and programmers who wish to read and write *event-lists* from within their programs. The sequential access routines are used for programs that sequentially access one event at a time (for example, most synthesis modules), and the linked-list versions are used when the entire list needs to be accessible in memory (for example, in an editing program). It often is useful to convert data in *event-list* format to and from a simple ascii representation. This is especially useful to users wishing to access the information in an *event-list* in a simpler fashion or with ascii-oriented UNIX programs. These conversions are supported by both 'C'-callable routines and by stand-alone pipes.

4. THE *sched* PROGRAM

The *sched* program plays a central role in executing, connecting, and coordinating the modular synthesis programs. As was illustrated in figure 1, the *sched* program receives an *event-list* from a composition or editing program or file and then demultiplexes this *event-list* stream into individual streams, one for each synthesis program. In the case of non-real-time synthesis, *sched* also starts a process which sums the outputs of all of the synthesis modules and either returns them to the user or writes them into a soundfile. *sched* communicates with this 'summing program' throughout the entire synthesis process in order to coordinate the reading of samples from each algorithm with the events given to it. In the case of real-time synthesis, *sched* controls the playback of each of the real-time synthesis modules through a single *sync* module, and each of the synthesis devices produces it's respective audio output which may then be mixed using an analog mixer.

The sched program executes the other software modules as completely independent UNIX process and connects the UNIX 'pipes' between processes. The exact configuration is provided to the scheduler as a script which contains an input program (from which events will be received), and a list of synthesis programs (to which events will be sent). Each pipe provides data flow in one direction: from the compositional program or *event-list* file into the scheduler, from the scheduler into each synthesis algorithm, and from each synthesis algorithm into the 'summing' process. The scheduler itself simply provides a means for initiating and connecting each process, for managing the data flow from the compositional program to each synthesis program, and for collecting the proper number of samples from each synthesis program. Beyond these functions, the scheduler is transparent, and in no case modifies the data that is passing through it. Thus, in the case of a single synthesis program, *sched* becomes a 'pipe' which simply passes the list of events received on its input to its output.

The UNIX 'shell' (or command interpreter) greatly expands the flexibility of our modular environment by providing the ability to combine multiple programs into single executable units. The composer can combine a number of separate synthesis programs into one 'synthesis program' by providing a UNIX 'shell script'. Each 'user-interface' or 'synthesis program' may actually consist of several pieces of software executed in series or, more importantly, connected in series. Thus, a synthesis script may pass a signal through a series of signal-processing programs piped together. The *sched* program then treats this 'synthesis script' as a single entity for the purposes of supplying *event-list* input and collecting output samples. One very important application of the 'synthesis script' concept is the adaptation of programs which do not use *event-list* input to this environment by constructing a script which places a 'filter' between the scheduler and the program. This filter translates the *event-list* from the scheduler into whatever input format the program expects. Thus, various programs which otherwise would be completely incompatible, and might even have been designed by two separate people for two entirely separate purposes, can be made compatible by supplying the appropriate input filters. This means then that any sound synthesis language which can be run under UNIX can easily be adapted to this environment, giving the user a wide range of possibilities from which to choose.

Figure 5 is an example of a 'software patch' made by *sched* with three synthesis programs which might be used either individually or in some simultaneous combination. In this example, a composer is using a simple *event-list* editor (*eled*) to edit a score, and is using *sched* to generate portions of the score directly from within the editor. There are several synthesis modules represented here. The first is a Karplus-Strong algorithm (Karplus, 1983) (written in 'C' and using the i/o-library to read *event-list* input), whose output is piped through a reverberator. The next program is a soundfile mixer which accepts mixing commands in *event-list* format and then passes the result on through a chorusing program. The mixer used on our own system was not written for this environment (but rather was written for the CCSS Soundfile System by Robert Gross (Gross, 1982)), and therefore requires an input filter to translate *event-lists* into its own input format. The third is an FM algorithm implemented as an instrument definition using the *inslib* program. The *pscale* filter provides a means for translating verbal descriptions of timbre into FM synthesis parameters. Details concerning the development of the psychoacoustic database allowing for this translation will be described elsewhere. One benefit of the *sched* environment apparent in this example, is that *sched* promotes treating soundfile 'mixing' and other signal processing of pre-recorded or analysed sounds in exactly the same manner as other sound synthesis methods.

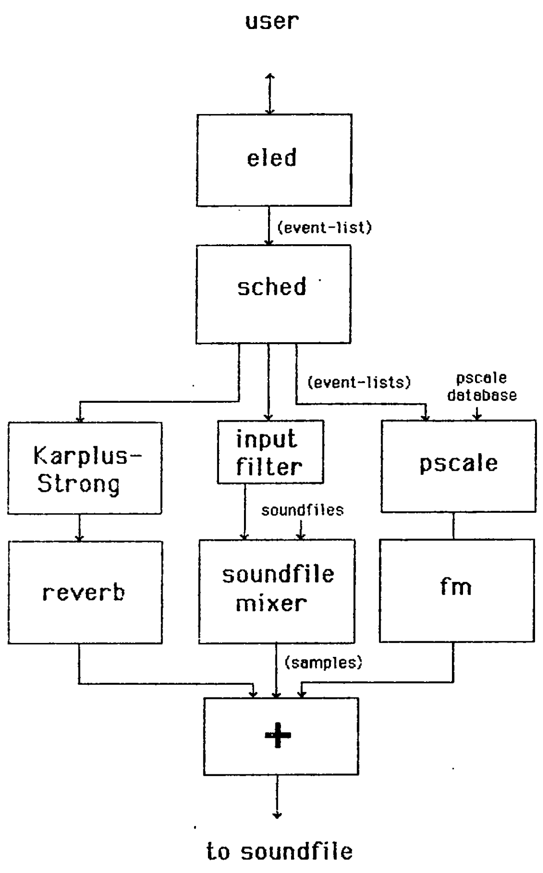

Figure 5. Example synthesis configuration.

5. THE *eled event-list* EDITOR

While the *eled* editor shown in figure 4 represents just one of the many possible user-interface programs which might be used within this environment, it is a good example of how our modular approach affects the design of a score editor. We chose to construct a very simple editor which could support prototyping of various score editing environments, rather than a complex program with prescribed capabilities. The editor is loosely based on *ed*, the UNIX line-oriented text editor, and like *ed* it is ascii-oriented and designed to be used on any terminal. The *eled* program supplies a few simple functions: First, it provides a 'constraint language' which allows the user to select certain events or groups of events from within the score on the basis of criteria such as starting time, event type, etc. Second, the program provides a simple syntax for basic operations such as typing, adding and deleting events, and for reading and writing *event-list* files. Third, *eled* provides a means for the user to pass groups of events through a UNIX program and merge the output of the program (assuming that the output contains valid events) back into the *event-list*. Thus, the composer may supply a collection of UNIX routines which function as the editor 'commands' and allow the editor to be easily reconfigured. Composers can supply their own compositional programs and tailor the 'editor' to their specific compositional needs. Finally, the editor allows one to attach input and output 'filters' (also separate UNIX programs) which map *eled's* ascii representation of the score into a screen-oriented or a graphic representation of the score and map graphic input back into ascii. With a small number of such filters, numerous graphic representations can be made of the same score.

Figure 6 shows a more detailed example of the *eled* editor, with a separate set of UNIX programs providing both the specific editor 'commands', and a 'graphic input-filter' providing the user with a graphical representation of the score. We also show a MIDI keyboard being used as an extra input-device to the editor. The keyboard information is first translated into *event-list* format by a software filter, and then is read as an extra input to the editor via a UNIX 'pipe' in the same manner that the editor pipes events through other UNIX commands.

The *eled* editor in some ways does for the user-interface environment what *sched* does for the synthesis environment: it provides a common gateway into many compositional programs. It also promotes prototyping and customizing of the environment for the individual composer. While this may not be the most computationally efficient approach to a given problem, we believe that the flexibility of the approach more than makes up for the loss. Then too, once a particular editing program becomes standardized, then that program can be incorporated into the editor. The details of this editing environment will be discussed at length elsewhere.

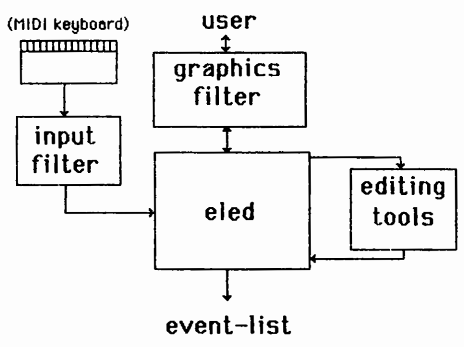

Figure 6. The *eled* program.

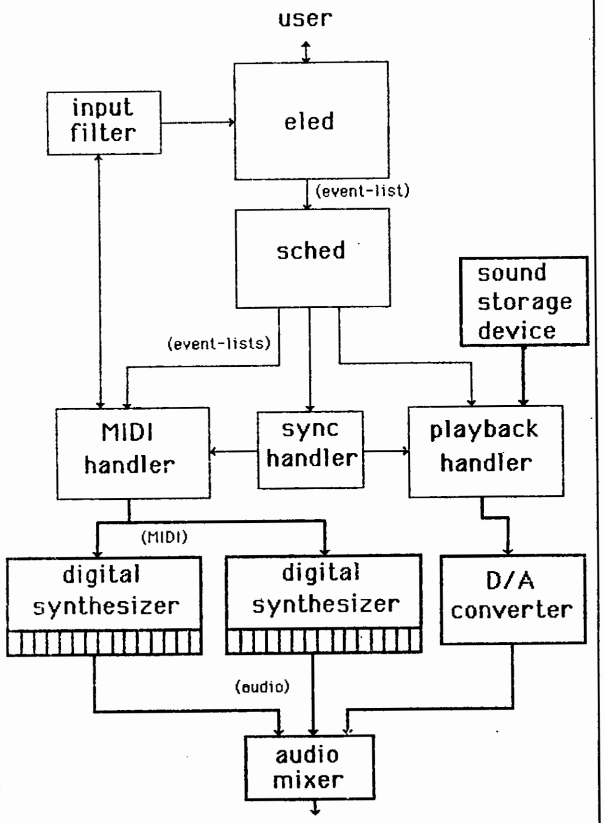

Figure 7. Real-time example.

6. REAL-TIME SYNTHESIS

From the perspective of the composer using compositional programs which pass *event-lists* to *sched*, real-time and non-real-time synthesis appear the same. *Sched's* real-time duties are identical to its non-real-time duties with the exception of not having to sum the output samples from each synthesis program. *Sched* always takes an *event-list* and separates it into separate lists for each synthesis module. In the case of soundfile playback, the *events* are segments of soundfiles to be played back from a disk; in most other cases, the *events* have actual synthesis parameters. The *event-list* that is passed to *sched* for real-time synthesis, must be from a preexisting score, because *sched* is not equipped to receive events in real-time. The user-interface or the synthesis hardware must be able to meet the real-time requirements themselves. For example, if a composer is 'overdubbing' a preexistent score by performing on a MIDI keyboard, the hardware synthesis device itself (in our case, a Yamaha DX-7) must perform the overdubbed notes while *sched* performs the existing score. The new notes are transmitted directly back to *eled* for merging with the original score.

Figure 7 shows a typical software patch established by *sched* for real-time synthesis. The synthesis modules are all hardware synthesis or playback devices. Because UNIX does not support real-time processing at the user level, each hardware audio device is accessed through a software 'device-driver' at the system level. Synchronization of the real-time synthesis modules is provided by the *sync* 'pseudo-device-driver' which also passes real-time messages from the user-interface and *sched* programs to the synthesis modules. The *sync* module is designed to allow all of the synthesis and playback to be synchronized by external MIDI signals to any MIDI musical device and/or external SMPTE time-code signals to SMPTE-encoded video. The specific details of both the *sync* module and other real-time aspects of the environment are beyond the scope of this paper, and will be discussed elsewhere.

7. CONCLUSION

Our concept of the 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 co-ordination. We were also concerned 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 into our software environment the few stable standards we can ensure. *Sched* provides a standard gateway to real-time and non-real-time synthesis while *event-lists* provide a standard format for communication among programs that may constantly change. Just the fact that these standards exist allows us to pursue compositional software like the *eled* editor as part of a long-term plan. Without planning, compositional software is the first to be obsoleted. A prototype version of the *sched* program has been in use at the Northwestern University Computer Music Studio for the last year with good results, and the *eled* program has been in use for approximately three months with equally good results.

8. ACKNOWLEDGEMENTS

Many of the ideas for this environment grew out of our experience using the modular signal processing tools contained within the CARL music software package written by F. R. Moore and D. Gareth Loy (Moore, 1982). In addition, 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 et al. in designing the SSSP system at the University of Toronto (Buxton et al., 1978a; 1978b; 1979). Special thanks must be given to Charles Smith and the System Development Foundation whose support for the Northwestern University Computer Music Studio has made this work possible.

9. 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., REEVES, W., BAECKER, R., and MEZEI, L. (1978) "The use of Hierarchy and Instance in a Data Structure for Computer Music," *Computer Music Journal*, 2.4, pp. 10-20.

BUXTON, W., SNIDERMAN, R., REEVES, W., PATEL, S., BAECKER, R. (1979) "The Evolution of the SSSP Score Editing Tools", *Computer Music Journal*, 3.4, pp. 14-25.

GROSS, R. (1982) "A Cylinder Contiguous Storage System", unpublished manuscript, Berkely: University of California Statistics Department

KARPLUS, K., STONG, A. (1983) "Digital Synthesis of Plucked-String and Drum Timbres", *Computer Music Journal*, 7.2, pp. 43-55.

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, pp. 18-29