Machine Tongues XV: Three Packages for Software Sound Synthesis

Stephen Travis Pope
The Nomad Group
P. O. Box 9496, Berkeley, California 94709 USA
stp@CNMAT.Berkeley.edu
http://www.cnmat.berkeley.edu/~stp/

The origin of the technology and methodology of modern computer music is certainly the *Music V* family of software sound synthesis systems developed since the late 1950s. In the "old days," this consisted of batch computer processing of musical programs expressed in terms of instrument definitions (programs) and score note lists (input data), generating sampled sound output data to off-line storage for later performance. The noticeable rekindling of interest in programs and languages for software sound synthesis (SWSS) and software digital audio signal processing (DSP) using general-purpose computers is due to a number of factors, not least among them the dramatic increase in the power of personal workstations over the last five years.

There are currently three widely-used, portable, C-language SWSS tools: (in alphabetical order) *cmix* (Lansky 1990), *cmusic* (Moore 1990), and *Csound* (Vercoe 1991). This article will discuss the technology of SWSS and then present and compare these three systems. It is divided into three parts; the first introduces SWSS in terms of progressive examples. Part two compares the three systems using the same two instrument/score examples written in each of them. The final section presents informal benchmark tests of the systems run on two different hardware platforms—a Sun Microsystems SPARCstation-2 IPX and a Next Computer Inc. TurboCube machine—and subjective comments on various features of the languages and programming environments of state-of-the-art SWSS software.

This author's connection with this topic is that of extensive experience with several different SWSS systems over the last 15 years, starting with MUS10 and including all three compared here: Csound (in the form of Music-11 initially) at the CMRS studio in Salzburg (Pope 1982); cmusic in the CARL environment at PCS/Cadmus computers in Munich (Pope 1986); and more recently a combination of cmix, Csound, and various vocoder software packages with user interfaces written in Smalltalk-80 at the CCRMA Center for Computer Research in Music and Acoustics at Stanford University (Pope 1992).

Software Sound Synthesis

> Computer music as a field has been likened to a building with a sign on it saying "Best Eats in Town." Many people go into this building expecting to find an elegant restaurant with a parchment menu, formidable wine list, and pleasant, efficient, even charming service. What they find instead, to their surprise, is a shiny, enormous, extremely modern kitchen, with abundant supplies of every kind of foodstuff in voluminous, refrigerated storage. Indeed, the "Best Eats in Town" are available here, but only to those willing to learn to cook! (Moore 1983)

There have historically been three distinct classes of electroacoustic music instruments or systems: tape-based *musique concrète* studios; modular analog synthesizers of (e.g.,) Moog, Buchla, ARP, and EMS; and software-based sound synthesis systems as described in the landmark book *The Technology of Computer Music* (Mathews 1969), which described his Music V sound synthesis program developed at Bell Telephone Laboratories. The technology of Music V-style languages will be described below. A major factor in the wide use of Music V during the 1970s is was the fact that it is written almost entirely in FORTRAN; its predecessors were generally written in assembly language, and were therefore not portable among machine architectures.

This technology was widely used during the 1970s in the form of several Music V descendents that ran on mainframe computers of the day, most notably for portable FORTRAN Music IVBF written by Godfrey Winham and Hubert Howe at Princeton University, Music360 (developed by Barry Vercoe also at Princeton), which ran on IBM System 360-class machines, Mus10 (developed by David Poole and extended by Tovar at Stanford University), which was used on the Digital Equipment Corp. DECSYSTEM-10 family. The 1980s saw a further steady increase in the availability of SWSS systems in the form of systems based on DEC PDP-11 computers using Vercoe's Music-11, and later DEC VAX-11 machines running the CARL/cmusic system developed by F. Richard Moore and D. Gareth Loy at the University of California, San Diego.

More recently, we have seen the rise of the "computer music workstation," including diverse configurations using personal computers and DSP subsystems such as the York University Composer's Desktop Project CDP (Atkins et al. 1987), and of course, the plethora of less flexible but real-time-capable MIDI-based computer music systems. Figure 1 shows a partial genealogy of the Music V family of languages and related systems—defined as those based on a software implementation of the unit generator instrument model and the function/note list score model. Another article (Pope 1992) presents an in-depth discussion of the engineering aspects of computer music workstations using modern technology, and introduces several of the systems listed in Fig. 1.

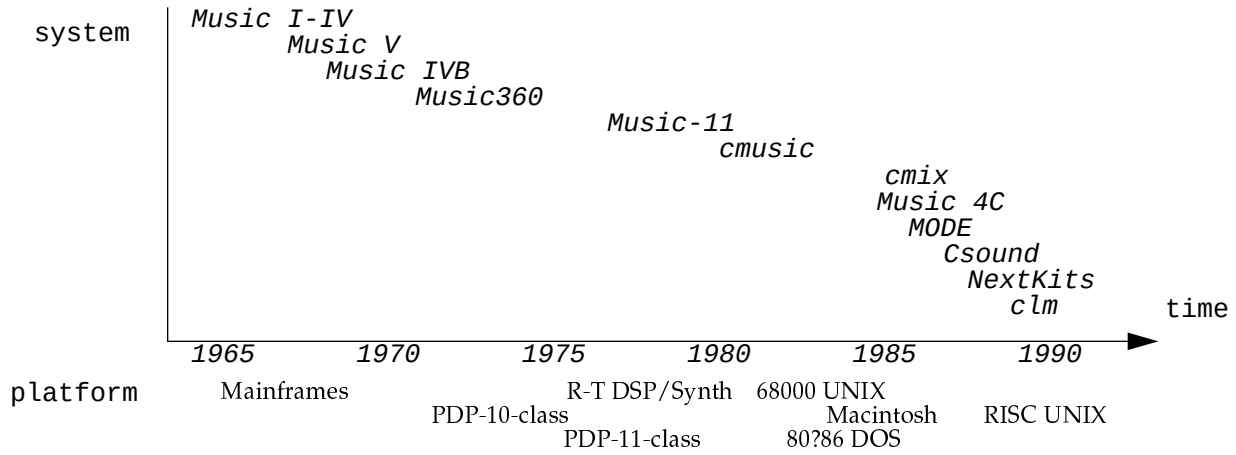

Figure 1: Time-line of Music V-like SWSS Systems—the figure shows a partial time-line of software sound synthesis packages, starting with the early languages developed at Bell Telephone Laboratories by Max Mathews. The systems named in the figure are described in the text.

Early SWSS Literature

The SWSS literature is generally said to have started with the abstract and publication of the "Acoustical Compiler" article by Max Mathews (1960, 1961) in the Journal of the Acoustical Society of America and the Bell Laboratories Technical Journal. The field gained much more visibility with the more widely circulated articles (Mathews 1963)—a popular and introductory essay—and (Tenney 1963)—a comment on the genesis of the field from a composer's point of view. The state of the art at the time of the development of Music V, in addition to the early experiments undertaken using it, are described in (Pierce, Mathews, and Risset 1965).

Current SWSS Packages

There are a number of other Music V-like SWSS languages in addition to the three discussed in this article, including the Music 4C system developed by Scot Aurenz at the University of Illinois at Urbana-Champaign (Beauchamp 1989), Bill Schottstaedt's Common Lisp Music, or clm (Schottstaedt 1992), and this author's MODE Musical Object Development Environment (Pope 1992). The NeXT computer also provides flexible C—and to some extent Objective C—programming libraries in its Sound Kit for unit-generator-based near-real-time development (Jaffe and Boynton 1989). All of these provide some higher-level score input languages, graphical tools, and easy-to-use extension languages.

The Technology of Computer Music

With reference to the design of Music V, Max Mathews wrote:

> The two fundamental problems in sound synthesis are (1) the vast amount of data need to specify a [sound] pressure function—hence the necessity of a very fast and effective computer program—and (2) the need for a simple, powerful language in which to describe a complex sequence of sounds. Our solution to these problems involved three principles: (1) stored functions to speed computation, (2) unit generator building blocks for sound-synthesizing instruments to provide great flexibility, and (3) the note concept for describing sound sequences. [...] [The composer] would like to have a very powerful and flexible language in which he can specify any sequence of sounds. At the same time, he would like a very simple language in which much can be said in a few words, that is, one in which much sound can be described with little work. The most powerful and universal possibility would be to write each of the millions of samples of the pressure wave directly. This is unthinkable. At the other extreme, the computer could operate like a piano, producing one and only one sound each time one of 88 numbers was inserted. this would be an expensive way to build a piano. [...] In a given instrument, the composer can connect as many or as few unit generators together as he desires. Thus he can literally take any position he chooses between the impossible freedom of writing individual pressure-function samples and the straightjacket of the computer piano. (Mathews 1969 p. 34-5)

A musical composition is programmed in two parts in Music V-style languages—the "instrument definition" describes the connections of signal generators and modifiers for the timbres that are to be used, and the "note list" is the score, described in terms of parameters sent to the instruments. The synthesis model is similar to that of an traditional analog synthesizer. One makes a "patch" among modules such as oscillators, amplifiers, mixers, and control function generators (the so-called *unit generators* or UGs of SWSS programs), and then sends trigger and control data to the patch to make sounds. Music V-style systems use various naming conventions to denote the use of scalar versus vector variables for inputs and outputs of unit generators in instrument programs. The parameters of note statements are generally referred to by number in instrument definitions, e.g., `p5` for the fifth parameter of the instrument (also called "p-field 5"). Oscillators, for example, write their output samples into buffers—arrays of data values that are managed by the system. These buffers are often referred to by number, e.g., `b1` and `b2` for sample buffers 1 and 2.

The parallels between the models of unit generators and the modular analog synthesizers developed after Music V have obvious advantages in terms of the ability of many composers to map their prior experience into the new realm (assuming that most composers coming to computer music have used analog synthesizers, which used to be true), and to make structured, scalable instrument descriptions. The disadvantage is that many types of sounds (e.g., those based on physical models, or involving complex time-

varying filters), are not easily modelled in terms of simple patches of the standard unit generators.

Instrument Definitions

A Music V punched card deck of the 1970's had the same structure as the input files used with SWSS packages today; it can be likened to the structure of a traditional batch data processing program. The "orchestra" file (like the program source code), consists of some header information—such as the sampling rate and output file format— (program header and declarations), followed by one or more instrument definitions (subroutines). The "score" file or "note list" initializes certain shared data, e.g., function wave tables, and then contains a (probably time-ordered) list of "note" commands that activate the orchestra's instruments at stated times with given parameters (like a batch data processing input file). The result of "executing" a Music V "program"—running the Music V "sound compiler" program with the "program" and "input" files—is a (possibly huge) output file of digital sampled sound, which can be listened to using a "play" program to send it to the output digital-to-analog convertors (DACs) of the system in real time.

An instrument definition is structured like a subroutine, macro, or procedure definition in any imperative programming language (e.g., ALGOL, PASCAL, or C); there are variable declarations, set-up expressions, and a "loop" of data manipulation statements that write into one or more global output buffers. Examples of this structure in several SWSS languages will be presented below. In the process of realizing medium-to large-scale compositions, composers often develop very many instruments. This "orchestra" may consist of variations of several common models (e.g., FM-based strings sounds, sound-file processing instruments, or bell sounds), and instruments may range from the very general—having many parameters and a wide range of musically-useful applications—to the very specific—having few parameters and a more concrete (less customizable and possible more dynamic) musical gesture. Instruments may generate output based solely on their input parameters (as in traditional oscillator-based instruments), or they may read real-time control data (e.g., via MIDI), or process pre-existing sound files (as in filtering or mixing instruments).

A number of graphical representations and visualization tools for instrument definitions have been used for the Music V family of languages. The most common ones use a flow-chart or data-flow style diagram to show the signal flow among unit generators where instruments generally "flow down" from input parameters through control signals to audio signals to the output. As an example, a note's parameters setting

the attack and decay times of a time-envelope would be is used to control the amplitude input of a signal oscillator, writing to the output. Data-flow block diagrams with (generally multi-input single-output) graphical icons representing instrument unit generator modules and connecting lines or arcs representing parameters or control or signal buffers will be used throughout the discussion of SWSS instruments below. A single statement (line) in the instrument definition program will translate into a single block icon whereby the arguments of the statement determine the connections between the icon's control and signal I/O ports.

Oscillator Unit Generators

The most basic unit generators in SWSS systems are stored-function oscillators—subroutines that read data values out of a stored table (the envelope function or wave table), at a rate determined by the given sampling increment. The increment is computed using a formula relating the size of the table, the sampling rate, and the frequency of the desired sound signal or duration of the control envelope (see [Mathews 1969 p. 53 ff.], [Moore 1990 p. 169 ff.], or [Vercoe 1991 p. 23-4] for details). An oscillator statement generally includes the command name (`OSC`, `osc`, `oscil`, etc.), the amplitude value (constant or function), the frequency value (constant, envelope or audio-rate signal), the wave table name or number, and the output buffer name—in Music V, this would be written

```
OSC amp, freq, out_buf_num, fcn_table_num, temp_place_holder;
```

for which realistic values might be

```
OSC B1, P6, B2, F1, P20;
```

This statement uses the vector of data in buffer named `B1` for the amplitude envelope, assigns the value of parameter field 6 to the oscillator's frequency, and writes output waves using data from function table 1 into the output buffer `B2`. The last parameter is the initial phase of the oscillator, which can be ignored in most applications and for which we use a "place holder" parameter field such as P20. The order and format of the parameters differs among SWSS systems, but the four basic parameters—output, amplitude, frequency, wave table—remain the same.

Function Generators

To create function tables for use as envelopes or wave-forms, generator commands fill data tables with values that depend on their parameters and which routine they use. There are usually several ways—`GEN` routines—to describe such vector data in SWSS languages, such as by interpolation between breakpoint values (as in sound envelopes),

by the summation of related sinusoidal components (as in wave table generation for additive synthesis), or by reading in external data files with or without some analysis and feature extraction. The length of function tables is generally fixed within an orchestra, often to a power of two such as 1024 samples. The `GEN` statement generally includes an action time—so that one can re-define a table during the performance of a score—a table number to create, a `GEN` routine selector, and parameters for the `GEN` routine, e.g.,

```
GEN action_time, table_num, routine, arguments
```

In a system where routine 1 generates tables by linear interpolation between breakpoint values, the parameter values `[0,0 1,1]` could be interpreted to generate a ramp-like function that increases from 0 to 1 over the duration of a note (between virtual times 0 and 1). The corresponding `GEN` command would be

```
GEN 0, 1, 1,   0, 0,  1, 1;
```

which defines function 1 at time 0 using routine 1 with the last four parameters as its arguments. Another common `GEN` routine uses the summation of sinusoidal functions of related frequencies and interprets its parameters as the relative amplitudes and phases of its components as we will see below.

Envelope Unit Generators

SWSS systems have several ways of providing the functionality of "envelope generators," unit generators that produce functions of time that step through a table once per "note." This can be achieved by setting an oscillator's sample increment to depend on the inverse of the length of the desired note, rather than the output frequency (i.e., read through the function table once per duration). In general, one can define line-segment, or exponential-segment functions, or use a stored envelope function, and read through them with control over the speed of the sections, usually used to control the "attack" and "decay" times of envelope functions. The choice in envelope generators ranges from special envelope generators (à la Csound's `linen` unit generator) and more general-purpose (but more complicated) segment readers (such as cmusic's `seg`).

Other Unit Generators

Modern SWSS systems provide many low- and high-level control and audio signal generator and modifier unit generators. These may include (e.g.,) variable waveform oscillators, noise and pulse generators, sound file input unit generators, multi-segment envelope generators, digital filters, digital delay lines, composite oscillators (e.g., reading phase vocoder data into an oscillator bank), composite all-pole filters (e.g., playing sounds through filters that read linear prediction data), or other musical, or DSP functions. Some manner of output unit generator is also required; this will read one or

several instrument buffers and write to formatted sound files, or unformatted sample streams. Some systems add room simulation unit generators allowing the user to declare a spatial configuration for a room and position sound sources in it.

Score Note Lists

The statements that describe which notes the instruments are to play, how, and when, are the "note list" part of the Music V SWSS music description. This "score" file includes the set-up of the function tables using `GEN` statements, the declaration of the input and output sound files and their formats, and the time-stamped note event data—a list of expressions that activate the instruments one-by-one at specified times with parameters supplied in the statement. The "note card" statement used for this consists of its keyword (`NOTE, not, instr`, etc.), the start time and duration of the event, the instrument number (or name), and the parameters of the instrument (e.g., amplitude, frequency, location, timbral properties). An exemplary Music V note would be

```
NOT start, instrument_num, duration, instr_args;
```

e.g.,

```
NOT 0  1  2  30000 0.0128 6.70;
```

which plays instrument 1 starting at time 0 for 2 seconds (or "beats," depending on the tempo model), passing the arguments starting at 30000 to the instrument subroutine as numbered p-fields.

There are generally facilities to use abstract time notations in SWSS scores, with the score's tempo defined as a beat-to-second map, and some way of setting and changing it. Most score languages also allow longer scores to be broken up into sections, each of which can have a separate clock and tempo. The purpose of sections is that they are sorted separately, and each start at relative time 0. The sections are computed in sequence by the program's scheduler. SWSS languages generally provide powerful facilities for generating and structuring note list files, and various kinds of short-hand for making the input and management of (possible very complex) instrument parameter field data for larger musical forms less laborious.

A Sequence of Tutorial Examples of Sound Generation

This section presents examples of Music V, cmusic, and Csound using two simple instruments. The first example is a trivial Music V instrument; the subsequent examples present more complex instruments in the other languages, comparing their features. We will discuss the instrument definition format, the score list syntax, and the process of executing the "compiler" package for each system.

A simple unit generator and an instrument definition patch are shown in Fig. 2; Fig. 2(a) shows the graphical symbol for an oscillator unit generator with its four relevant features: amplitude, frequency (or sample increment), wave form (timbre), and output. Figure 2(b) illustrates the use of this unit generator in an instrument; two oscillators are connected such that the inputs of the first control its amplitude and its sample increment based on the note's duration, and its output is connected to the amplitude input of an oscillator that has its sample increment derived from the note's frequency. The first oscillator thus functions as an envelope generator that generates a time-varying control function for the amplitude envelope of the second oscillator's notes.

(a) osc unit generator *(b) oscillator with amplitude control*

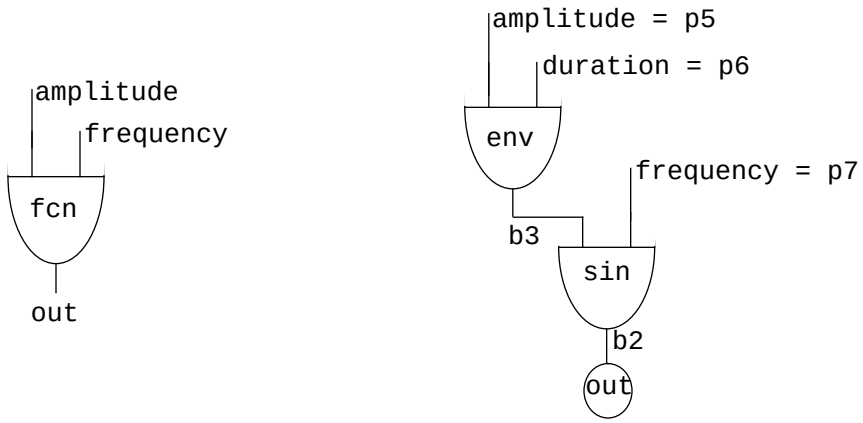

Figure 2: The Oscillator Unit Generator and a Simple Instrument Patch—Fig. 2(a) shows the icon used for a stored-function oscillator in instrument definition block diagrams, with its amplitude and frequency inputs, its stored (envelope or wave-form) function table, and its output. Figure 2(b) shows a simple instrument, in which one oscillator generates an amplitude envelope for another, the sine-wave output.

(a) amplitude envelope function *(b) wave table function*

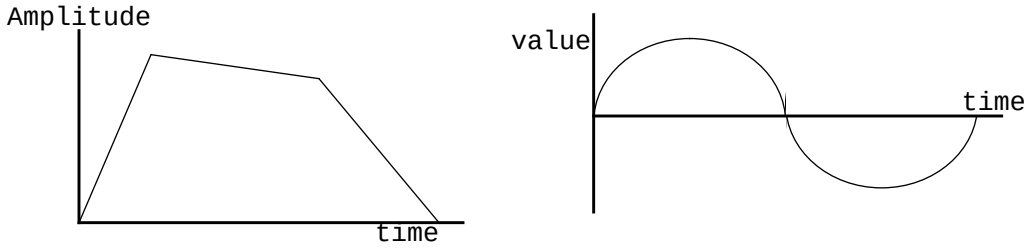

Figure 3: Function Table Values for an Envelope and a Wave Form Signal—the stored function tables for an amplitude envelope and an audio signal wave-form are shown; they will be created using generator statements.

The parameters of this instrument are the pitch, amplitude, and waveform of the output signal oscillator, the duration of a note, and the amplitude envelope function. In order to use this instrument, one must write Music V programs that define three components: the instrument, the function tables for the two lookup oscillators, and the desired note parameters. There will be two function tables: one holding values for the amplitude envelope (e.g., Fig. 3(a)), and the other the output oscillator's wave form function (e.g., Fig. 3(b)). The commands that declare and define function tables may be seen as being part of the instrument or the score, depending on the nature of the compile-run cycle (see below).

The Music V instrument definition for this example would read as shown in the first group of statements in Fig. 4. Comments start with the `COM` statement and go until the end of the line in Music V. The instrument definition uses the `OSC` unit generator for both the envelope function and the waveform oscillator by having them read different function tables. Note the use of two different buffer numbers for the signal buffers here; in some SWSS systems, one can reuse the same buffer number in several places in an instrument, as is possible here because no unit generator depends on *both* b1 *and* b2 for its input. The Music V `OUT` unit generator writes samples from its input to its standard output buffer `B1`, which is written to the output disk.

The specification of unit generator parameters and the translation of note command parameters varies greatly among SWSS systems. In Music V, for example, one typically passes amplitude and frequency parameters directly from the score to the unit generators, and is forced to translate from pitch values to oscillator sample increments and from loudness values to integer amplitudes in the score.

A Music V score file for this example instrument would first define the two function tables and then play notes on the instrument by providing values for its parameters, as in the second group of statements in Fig. 4. One declares function tables 1 and 2 using the `GEN` command and routines 1 (linear interpolation between breakpoints), and 2 (summation of sines). The parameter data in the note command p-fields signifies the notes' parameters—start time, instrument number, duration, amplitude, envelope duration increment, and oscillator frequency increment. The amplitude is given in absolute numbers (assumed in the range 0-32767 for this example, implying 16-bit linear samples). Note the increment parameters; p6 is the envelope increment—related to the table length, the inverse of the note duration, and the sample rate—p7 is for the frequency increment—related to the frequency, the table length, and the sample rate.

There may be one, a few, or many notes in the subsequent note list, depending upon whether the user is testing the instrument's parameters, developing small musical

textures or gestures, or performing an entire section or composition in the current "pass." Notes can overlap, and several notes may be active in the same instrument at the same time; the system's scheduler handles multiple instrument activations and output summation.

```
COM   Music V Instrument
COM   Note Parameters:
COM      p2=start, p3=instr_num,
COM      p4=duration, p5=ampl,
COM      p6=dur_incr, p7=freq_incr

COM   Definition for instr. 1
   INS 0 1;
COM       AMP FRQ OUT FCN TMP
      OSC P5  P6  B3  F1  P20;
      OSC B3  P7  B2  F3  P21;
      OUT B2  B1;
   END;

COM Generate Function Tables
COM Generate f1 as an envelope with GEN 1
COM Routine 1 takes x/y breakpoints
   GEN 0 1 1  0 0  0.99 50  0.8 480  0 511;
COM Generate f3 as a sine using GEN 2
COM Routine 3 takes partial num. and ampl.
   GEN 0 2 3  1 1;

COM Play Two Notes
COM p1  p2 p3 p4   p5    p6      p7
   NOT  0  1   2   30000 0.0128  6.70;
   NOT  2  1   4   8000  0.0064  8.20;

COM Terminate the score at time 6
   TER 6;
```

Figure 4: Music V instrument definition and note list for the instrument of Fig. 2(b)

To execute a Music V program, the instrument definition (possibly defining many instruments), is read, and possibly compiled into a compact and efficient internal format; then the note list is expanded and sorted, possibly applying score language preprocessors. The actual sample computation task consists of a "scheduler" reading through the score data in time order, activating instruments as appropriate, and summing their respective outputs into the output sound file(s). One "plays" the sounds using a program that sends the sample data (stored on disk or tape) to a DAC in real time. (In the early days, this was often a different machine from the one that computed the samples.) The steps of the process are illustrated in Fig. 5.

The actual interaction with the batch programs also varies widely among SWSS systems, and shows the generations of user interface technology since the 1960s, progressing from card decks to terminal edit-compile cycles, to window-based incremental interactive front-ends with graphical description of instrument definitions and score editing.

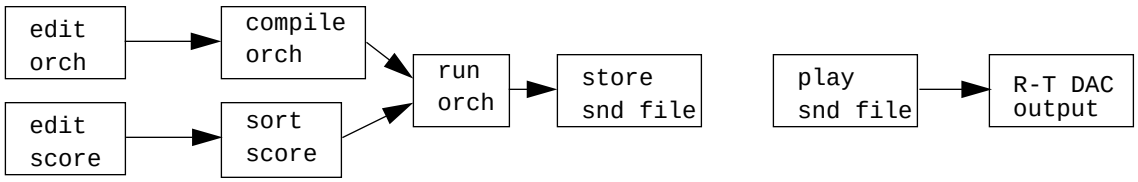

Figure 5: Steps of the SWSS "Compilation" Process

FM Example

The example shown in Fig. 6 is a simple frequency modulation (FM) instrument (Chowning 1973), in which a single modulator wave is used with exponential ADSR (attack-decay-sustain-release) envelopes for the amplitude and modulation index. It uses two line-segment envelope unit generators and sine-wave oscillators for modulation and carrier waveforms. The parameters of this instrument are:

p1 = note command;
p2 = instrument name or number;
p3 = start time (in seconds);
p4 = duration (in seconds);
p5 = amplitude (in the range from 0 to 1);
p6 = fundamental pitch (in symbolic pitch units);
p7 = carrier-to-modulation frequency ratio;
p8 = index of modulation;
p9 = amplitude attack time (in seconds);
p10 = amplitude decay time (in seconds);
p11 = modulation index attack time (in seconds); and
p12 = modulation index decay time (in seconds).

A graph of the generic ADSR envelope function, the unit generator symbol that will be used for it, and a block diagram of the FM instrument are shown in Fig. 6. The exponential function shown in Fig. 6(a) is to be applied to the amplitude and modulation index envelopes. The desired parameters are the duration *t3*, the attack time *t1*, the decay time (*t3* - *t2*), and the overall amplitude *a1*. The decay ratio (sustain level) *a2* is set to 0.8 * *a1* for the amplitude envelope and 0.6 * *a1* for the modulation index. This unit generator can be drawn as shown in Fig. 6(b), showing an envelope icon with four inputs, an envelope

function wave table, and one output. One generally specifies an exponential envelope's function table using a `GEN` statement that takes three values for each break-point: its x- and y-coordinates, and the exponential weighting—a multiplicative factor applied to the exponent to make the transition more or less sudden.

Figure 6(c) shows the instrument, with the data flowing from the note parameters into the envelope generators, then to the modulator and carrier signal oscillators, then to the output.

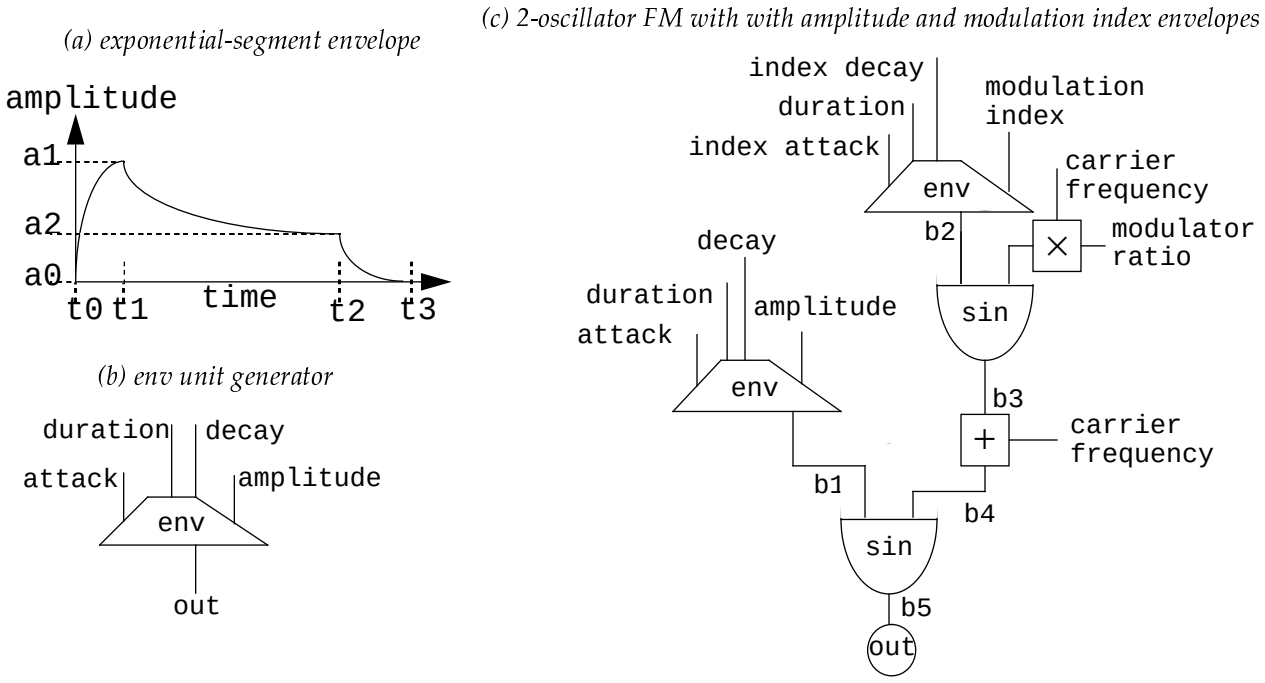

Figure 6: Exponential Envelope and FM instrument block diagram—(a) exponential-segment envelope; (b) env unit generator; (b) 2-oscillator FM with amplitude and modulation index envelopes

The note command will list the instrument's parameters in order

```
p1   p2     p3    p4    p5    p6    p7    p8    p9    p10 p11   p12

note name   start dur   ampl  pitch c:m  index att dec i_att i_dec
```

Figure 7 gives the cmusic language definition for this instrument; the instrument's format is much like that of a subroutine definition in an imperative ALGOL-like programming language. In cmusic one can write all three sections (instruments, functions and note lists) in one file (as shown), or put them in separate files and combine them using the C-preprocessor's `#include` directive to "include" the orchestra and/or function files into a score for execution. Cmusic comments are enclosed in (nesting) curly-braces. The

instrument is more readable because cpp preprocessor macro definitions (`#define name value`), are often used to map parameters onto symbolic names for use in unit generator statements. Note the different argument order of cmusic statements; the unit generator output always comes as the first argument, e.g., oscillators are written

```
osc out amp freq fcn_table_number temp_phase;
```

Place holder variables such as the oscillators' initial phase can be written as "`d`" in cmusic. Note also the different format for the `out` unit generator; in cmusic, it writes data from its argument (an audio buffer) onto the process's output (file or stream) directly, not needing a special buffer designation. The instrument definition can be followed directly by the function declarations and the note list. We will revisit this instrument and write it in Csound and cmix in below.

There are many flavors of single- and multiple-modulator FM instruments implemented in every possible SWSS language, ranging from small, fast, flexible fm "cliché sound" instruments to complex simulations of various instrument families such as voices, percussion sounds, brass instruments, or plucked and bowed strings.

```
{ Cmusic FM instrument     }
{ note format e.g., }
{ p1  p2 p3 p4  p5  p6   p7  p8  p9  p10 p11  p12}
{note 0 fm1 dur amp freq c:m ind att dec iAtt iDec}

   { include cmusic definitions }
#include <carl/cmusic.h>

   { set important globals }
set rate = 44100;
set channels = 1;

   { names make instruments easier to read }
#define  DUR   p4
#define  AMP   p5
#define  FREQ  p6
#define  RATIO p7
#define  MODF  b10
#define  IND   p8
#define  ATT   p9
#define  DEC   p10
#define  IATT  p12
#define  IDEC  p12

{ instrument named fm1 defined at time 0 }
ins 0 fm1;  { b1 = ampl env }
   seg   b1  AMP f2 d ATT 0 DEC;
            { b2 = index env }
   seg   b2  IND f2 d IATT 0 IDEC;
```

```
            { scale frequency }
   mult  MODF FREQ RATIO;
            { b3 = modulator }
   osc b3  b2 MODF f1 d;
            { add base freq and mod }
   adn b4  b3 p6;
            { b5 = carrier }
   osc b5  b1 b4 f1 d;
   out b5;
end;

   { function declarations }
SINE(f1);{ this is pre-defined }
            { GEN4 uses exponential segments }
GEN4(f2) 0,0,-1   0.1,1,0   0.8,1,-1   1, 0;

   { note list }
{    0 fm1 dur amp freq  c:m ind att dec iAtt iDec }
note 0 fm1 2  -2dB 100Hz 1   1  .2  .1  .05  .4;
note 2 fm1 2  -6dB 100Hz 1   3  .2  .1  .05  .4;
ter;
```

Figure 7: FM Instrument written in cmusic

Note that we could have written the cmusic version more easily if the modulator oscillator's frequency were put in the score rather than the c:m ratio. Sample buffer numbers are also not reused in this example; one could write the instrument using two buffers. We use the cmusic `dB` and `Hz` post-operators to provide amplitude and sample increment values in convenient units. The `sec` operator converts time values into sample increments and is often used in envelopes (e.g., `p5 = p4sec = 1/p4Hz`). Complex parameter-mapping expressions are handled in scores in cmusic (rather than in the instruments as in Csound); one can use parameter combinations such as `((C(0)*2^(p8/12))` for pitch values in a note statement.

Sound File Mixing Example

The third example will be an instrument that reads monophonic sound files from disk and mixes and writes them to a stereo output with variable fade-in and fade-out times and stereo positions, which we have implemented first in Csound. The parameters of a note for this instrument will be:

p3 = start time (in seconds);
p4 = duration (in seconds);
p5 = amplitude (in the range from 0 to 1);
p6 = input sound file name or number;
p7 = amplitude attack time (in seconds);

p8 = amplitude decay time (in seconds);

p9 = stereo position (0=left, 1=right).

The format of a note command for this instrument will be something like:

```
note instr start duration amplitude file attack decay position;
```

The instrument definition block diagram is shown in Fig. 8(b). Note the sound file reader unit generator in Fig 8(a); it takes a sound file name or number as argument (with optional skip time, duration, and sound file format fields), and writes samples from the file into an audio buffer in the instrument. The instrument in Fig. 8(b) reads sample from the file, applies an envelope to them, and multiplies the signal by left- and right-channel scale values.

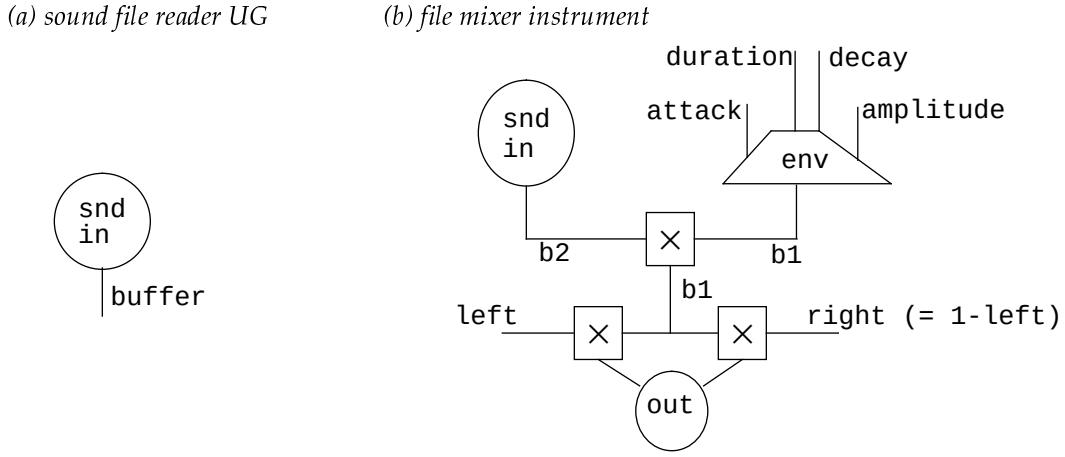

Figure 8: Sound file mixing instrument definition—(a) sound file reader UG; (b) file mixer instrument

The Csound orchestra for this instrument is shown in Fig. 9; it would normally be stored in a file with ".orc" as the file name extension. It starts with the header, which declares values for four important global constants. Comments start with semicolon and end with new-line in Csound. The instrument definition starts with the `instr i_number` statement and ends with `endin`. unit generator statements start with the result value and continue until the end of the line, i.e., `as1` is a buffer and `soundin` and `linen` are unit generators; an oscillator would read,

```
asig_out oscil amp freq fcn_table_num
```

The `soundin` unit generator reads sample data from the sound file named soundfile.*x* where *x* is the file number given as a parameter. This makes the scores numerical, but means it is necessary to copy or link sound files into a special directory with meaningless names. (The *make* utility makes this manageable under UNIX.) The Csound `linen` unit

generator reads an audio signal and applies an envelope to it, saving us an extra multiplication step. The stereo output unit generator takes two arguments that are the signals to be sent to the left and right channels, respectively.

The note list for the instrument is stored in a separate file with the file name extension ".sco." It is shown in the last five lines of Fig. 9 and has no `GEN` statements because the instrument uses no stored tables. Csound notes start with the character "i" followed by the instrument number and the note parameters.

```
; Csound orchestra file

; sound file instrument
;  p1 p2    p3  p4  p5       p6     p7    p8
;  i1 start dur amp f_number attack decay position
;  HEADER: Set 4 magic constants
   sr=44100    ; audio rate
   kr=441      ; control rate
   ksmps=100   ; == sr / kr
   nchnls=2    ; stereo
; Stereo soundfile mixing instrument
   instr 1
            ; i-rate variables
            ; (updated once per note--optional)
   idur  =  p3       ; duration
   iamp  =  p4       ; amplitude
   iatt  =  p6       ; attack time
   idec  =  p7       ; decay time
   il    =  p8 * iamp         ; left factor
   ir    =  (1 - p8) * iamp   ; right factor

            ; read from file p5 into as2
   as2   soundin p5, 0, 4
            ; apply envelope to as2
   as1   linen as2, iatt, idur, idec
            ; place in stereo output
   outs  as1*il, as1*ir
   endin

; Csound score file: notes for sound file instrument
;start  dur amp   fcn   att      dec   pos
i1 0      4 1.0   1     0.001    1.0   0.5
i1 4      4 1.0   1     2.0      2.0   0.1
end
```

Figure 9: Sound file mixing instrument of Fig. 8(b) in Csound

To give the reader an impression of a more involved instrument definition, Fig. 10 shows a multi-carrier FM instrument representing a model of a trumpet tone devised by Dexter Morrill in the MUS10 language. It is described in detail in (Morrill 1977).

(Fig. 10 not included here.)

Figure 10: FM trumpet designed by D. Morrill (from [Morrill 1977])

SWSS System Issues

Before the detailed comparison of the three subject SWSS packages, we will discuss several of the central design issues in programming languages for DSP and SWSS.

Oscillator Unit Generator Issues

There are several types of oscillator unit generators used in SWSS systems, the most important distinction being those between interpolating and truncating oscillators. This issue arises when the ratio between the sampling frequency, the function table length, and the desired output frequency causes the sample increment to be a non-integer (e.g., the frequency-to-oscillator-increment conversion formula mentioned above leads one to want to read every 1.5 stored data values). In this case, the system can either truncate or round the sample increment to be an integer (i.e., take the first or second table value), or it can interpolate between wave table values (i.e., compute a value that is half way between the first and second values in the table). The first option will be somewhat faster, while the second will produce better-sounding results—less quantization distortion—with shorter function tables. Good SWSS systems provide both options—interpolating oscillators and variable table length—so that the user can determine what the most important factors are. This is typical of the "speed vs. space" trade-offs often found in software engineering.

Audio and Control Rates

Unit generators, whether standard oscillators or not, may write their output values every sample, or they may be told to update less frequently, e.g., every 32 samples, to save computation. The differentiation between audio and control signals and update rates was an important issue in the design of analog systems in the 1960s—witness the difference in the treatment of control vs. audio between Buchla and Moog analog synthesizers. Several SWSS languages allow flexible, run-time specification of control and audio rates and/or control and sample buffer sizes, though this provides neither a fully type-safe programming environment, nor a flexible model of multi-rate signal processing, and introduces myriad opportunities for aliasing problems and artifacts.

SWSS Usage

Among the reasons for the rekindling of interest in SWSS systems in the 1990s is the increasing computational power of workstations and networks, which means that the issue of modelling generality often outweighs that of non-real-time performance. This trend can be predicted to continue in the near future. The generality of SWSS input languages and the continued popularity of the modular synthesizer model has meant that there is widespread interest in the construction of interfaces to real-time MIDI/DSP systems using this paradigm. Hardware/software packages that merge features of SWSS

and real-time performance systems are exemplified by (e.g.,) Accelerando (Lent, Pinkston, and Silsbee 1989), Kyma (Scaletti 1989), and the IRCAM Musical Workstation (Lindemann et al. 1991). Advanced front-end tools for SWSS systems will be discussed below. The three systems presented here provide "batch" tools accessed through the UNIX shell command-line interface by default, adhering to the "edit, compile, compute, play" development cycle. Cmix also provides an interactive usage mode.

Sound Compilers

SWSS programs have often been referred to as "sound compilers," and while their input languages and usage do bear distinct similarities to programming language compilers, there are also important differences. Using the model of programming for sound synthesis is very compelling and dangerous at the same time. SWSS systems generally try to "feel" like programming languages with batch compilers, but the model is known to break down in many cases when composers try to "overuse" the procedural programming style in instrument definitions or note lists. This can mean needing to add many new unit generators, or doing everything "by foot" with very low-level unit generators, or writing scores or score-generating programs that exceed the practical limits of the system. SWSS systems whose orchestra language is easily accessible and extensible in the same language used in instruments and/or scores are too rare; historical examples are MUS10 (where the extension language was SAIL) and Music 4BF (based on FORTRAN); modern C-based examples are Music 4C and cmix, though neither of these is completely consistent with C in orchestra and score formats. Some SWSS packages (e.g., Csound and cmix), "compile" instrument definitions into a compact optimized internal format, while others (e.g., Music V), are actually interpreters that get their speed through the use of vector programming. It is interesting to note that non-C-based SWSS tools such as Common Lisp Music, Kyma, and MODE are able to provide more uniformity of model between the instrument, score list, score structuring, and interactive front-end languages than seems to be possible in the C/UNIX environment.

Sound Utilities for SWSS Systems

Any engineer who set up an SWSS studio before about 1988 had two interfacing issues to deal with that were more complicated and error-prone than the SWSS software itself—the provision of a sound file storage management system, and of real-time I/O programs such as "play" and "record." Before the most recent generation of storage disks and operating systems, it was assumed that computer music studios would have special disks for sound file storage, and that these disks would use a special file system (see e.g., [Loy 1982], [Gross 1982], and [Fichera 1991]). Utility programs were then needed to get sounds

onto and off of the sound file disks (e.g., *fromsnd* and *tosnd* in the CARL system), and to manage sound file space (e.g., *lsf* commands in several flavors to list sound files). The DAC and/or ADC hardware issue could be solved using off-the-shelf components for standard computers only after about 1980 (with the introduction of Digital Sound Corp's DSC-200 DAC/ADC system for DEC PDP-11 and VAX-11 computers).

All three of the SWSS systems presented here come with some type of sound file utilities that involve formatted files that have special data structures in their first few blocks (the sound file headers), followed by (possibly very large) sample data arrays. These header data structures generally include the sound's sample rate, number of channels and storage format, and may also include such information as the maximum amplitude of the file, a comment, documentation or annotation text, or even a list of named entry points or "cues" within the sound. Most modern SWSS systems perform all their internal calculations in floating-point numbers, and provide the possibility of writing out integer or floating-point sound files. The use of floating-point storage with external gain programs—as described below by F. R. Moore—is gaining wider acceptance as disk storage gets less expensive. This frees the composer from having to pay such close attention to instrument amplitudes to avoid "samples out of range" errors when the summed signal exceeds the range of integers used for sample representation (typically 16 bits).

Advanced User Interfaces for SWSS Systems

One of the immediate problems in building sophisticated scores is the complexity of specification of break-point time envelopes and of wave tables via Fourier overtone summation. Graphical editors for envelopes and overtone spectra emerged soon after appropriate graphical displays and input devices were developed, and now come in many flavors. None of the systems discussed here, however, include any but the most rudimentary graphical tools, and the available "third-party" extensions leave much to be desired in most cases.

There are several interactive applications that offer a graphical user interface for flexible modular software-only or hardware-assisted sound synthesis. The Kyma system includes a mix of graphical and text-based facilities for instrument and score description. The Digidesign Sound Tools application, the MaxDSP program by Miller Puckette (part of the IRCAM Musical Workstation [Lindemann et al. 1991]), The Berkeley Ptolemy system, and the Accelerando system's patch editor (Lent, Pinkston, and Silsbee 1989), are data-flow-oriented graphical programming languages for DSP and disk-based sampled sound processing. This author's MODE system provides front-end tools for instrument

definition and sound file mixing, along with an array of score generation possibilities. Adrian Freed's MacMix (marketed by Studer Dyaxis), is a representative example of the various cue-sheet-based sound mixing front-ends.

Csound, Cmusic and Cmix

The format for the comparison below will be to describe each system's programming language model, specific instrument and note list description languages, and system architecture and music programming environment in turn. The language issues will include the level of (data and control) abstraction, flexibility, and programming style of the instrument language and standard note list preprocessors. The development tools and utility programs will be evaluated in terms of the handling of the edit/compile/play cycle and the integration with external programming and DSP tools. The following sections will present the three systems in roughly the order of their development. The FM instrument shown in Figs. 6 and 7, and the sound file mixer program of Figs. 8 and 9 will be presented for each language.

Csound

The heritage of Csound goes back to Music360 developed by Barry Vercoe while he was at Princeton University in the late 1960's. A descendent of this was Music-11, developed by him at the MIT Electronic Music Studio, which ran on DEC PDP-11 computers and was very popular during the late 1970s and early 1980s. Csound is an upward-compatible reimplementation of Music-11 which is written entirely in C for portability. It is now distributed for free by Vercoe and his colleagues at the MIT Media Laboratory. The upward compatibility means that old Music-11 instruments and scores can be run unchanged in Csound. (This author had a chance to test this by resynthesizing a composition originally realized on a PDP-11/44 in Salzburg in 1983 [at 16 kHz sample rate], in 1991 on a NeXT machine at CCRMA in Stanford [at 44.1 kHz sample rate].) Csound is also known for being quite efficient and portable; it runs on several UNIX workstation as well as on IBM PC-compatible, Atari, and Apple Macintosh personal computers.

Csound uses a mix of assembly language—e.g., `goto` as only control structure—and FORTRAN—e.g., variable type being determined by the first character of a name—as its programming language model, and has subtly different instrument definition and score syntaxes. The advantage of this is very low learning and programming overhead for non-programmers (as with FORTRAN); the disadvantage is poor scalability to complex instruments and rather difficult debugging (as with assembly language).

A Csound instrument file includes a header that defines the sample rate and number of channels. Here one also defines the "k-rate," which is the sample rate used for control functions as described above (i.e., it may be many times less than the audio sample rate). One must, for some reason, also give the ratio of the a- to the k-rate, the size of the control blocks. Any number of instrument definitions follow this header; instruments are numbered. The Csound definitions for our two example instruments are shown in Figs. 11 and 9 (above), which include copious comments in the hope that they will be readily understandable to most readers. The arguments to Csound's oscillator statement are:

```
out oscil amp freq, fcn_table_num
```

Note the use of arithmetic expressions in the argument fields of unit generator statements—a valuable feature. Csound score files are simple numerical data files that define any function tables used in instruments, set the global tempo, and then have one or more sections of note commands. Figures 12 and 13 reproduce sample score files for the two example instruments.

```
; Csound FM instruments

;  p1 p2    p3  p4  p5   p6  p7  p8  p9  p10  p11
;  i1 start dur amp freq c:m ind att dec iAtt iDec

;  HEADER: Set 4 magic constants
   sr=44100    ; audio rate
   kr=44100    ; control rate
   ksmps=1     ; == sr / kr
   nchnls=1    ; mono

;  version 1--two envelopes and two oscillators
instr 1
   idur  = p3     ; duration
   iamp  = p4     ; amplitude
   ifreq = octcps(p5)   ; carrier frequency
   imodf = p5 * p6      ; modulator frequency
   indx  = p7     ; index of modulation
   iatt  = p8     ; attack time
   idec  = p9     ; decay time
   iiatt = p10    ; index attack time
   iidec = p11    ; index decay time
               ;amplitude envelope
   kamp  envlpx iamp, iatt, idur, idec, 2, 0.8, 0.01
               ; index envelope
   kind  envlpx indx, iiatt, idur, iidec, 2, 0.6, 0.01
               ; modulator
   amod  oscili  kind*ifreq, imodf, 1
               ; carrier
   acar  oscili kamp, (ifreq+amod), 1
   out acar
endin
```

```
;  version 2--use foscili unit generator
instr 2        ; k1 = amplitude envelope
   k1 envlpx p4, p8, p3, p9, 2, 0.8, 0.01
               ; k2 = index envelope
   k2 envlp p7, p10, p3, p11, 2, 0.6, 0.01
               ; a5 = fm oscillator
   a5 foscili k1, p5, 1, p6, k2, 1
   out   a5
endin
```

Figure 11: FM instrument of Fig. 6(c) in Csound

```
; Csound Score for FM Instrument
; Generate Sine in F1 using GEN 10
f1 0 1024 10 1
; Function 2 is the attack envelope using GEN 7
f2 0 1025 7 0.0001 1024 1.0

;  p2   p3  p4    p5 p6 p7 p8 p9 p10 p11
; start dur amp   freq  c:m   ind   attdeciAtt iDec
i1 0       220000 7.02  1     1     .1 .2 .05.4
i1 2       220000 7.00  1     6     .1 .2 .05.4
i1 4     2  20000 7.07  1     4     .1 .2 .05.4
end
```

Figure 12: Score file for the FM instrument in Csound

```
; Csound Score for Soundin Instrument
; p2   p3  p4 p5 p6 p7  p8
;start dur amp fileattdecpos
i1 0   41.010.001 1.0   0.5
i1 4  41.01 2.0   2.0   0.1
i1 7  40.11 0.0   4.0   0.8
end
```

Figure 13: Score file for the mixing instrument of Fig. 8(b) in Csound

Running a Csound "orchestra" involves the `csound` shell command (on command-line-oriented user interfaces such as those for UNIX or MS-DOS), which requires the names of the score and instrument files and may be used with various option flags. List- and menu-oriented front-ends to the process exist on several systems (e.g., Macintosh and NeXT), which support the interactive selection of files, the setting of global variables, and the viewing of function tables. The Csound command carries out several steps of processing that are visible to the user through the package's verbose output messages. The first of these steps is score expansion, where the useful "carry" short-hand feature is expanded and other score processing takes place. Step two is score sorting, where the notes in each section of the score are sorted into time-ascending order. The score extraction feature may be used to select or pick out specified intervals, sections, or instruments from a large, multi-section or multi-part score. The instrument file (the

orchestra) is "compiled" next into an internal format (a "virtual machine language") for fast execution. Function generation and optional display is followed by the final step—to "execute" the compiled orchestra giving it the expanded, sorted, extracted score. The output data can be written to a sound file from within Csound if the option `-o outfile_name` is given, as in the first command in Fig. 14. The sound file can then be played with the host system's real-time DAC interface program *play*. If the file name in the Csound command happens to be "`stdout`," Csound writes samples to its UNIX process's standard output stream, which can then be "piped" to the input of a program that creates a sound file with the proper header information, such as *tosnd* as shown in the example in the last line of Fig. 14, where floating-point samples are written through a UNIX pipe to `tosnd`. Other command-line options allow users to set the sample format (including compressed 8-bit Mu-law), set the level of processing verbosity, or ask for test processing only—allocating instruments and checking arguments but generating no samples. One cannot, however, set the sample or control rates from the shell command in Csound; they are fixed in the orchestra file's header.

```
csound -o fm1.snd fm1.orc fm1.sco
play fm1.snd
csound -o stdout -f fm1.orc fm1.sco | tosnd -f fm1.snd
```

Figure 14: Steps in Csound processing expressed as UNIX commands

Two features of the Csound instrument language distinguish it from Music V and most other SWSS systems: the use of k- vs. a-rate signals; and of complex expressions in generator arguments. The differentiation between k- (control) and a- (audio) sample rates means that when one declares internal variables in an instrument, one specifies whether their values should be updated once per note, once per control period, or once per sample period (based on the first letter of their names as in FORTRAN, using buffer names such as *ivar1, ksig1* or *asig1*). The orchestra file header defines the audio sample rate, the *a-rate*, and the control sample rate, the *k-rate* (and the ratio between them). This can be used to save time when developing an instrument in that one can set the k-rate so that control functions such as envelopes are computed less frequently than every audio sample period, which can result in significant compute ratio improvements.

This feature qualifies as "a blessing and a curse" because while orchestras often run much faster (e.g., with a-rate=44.1 kHz and k-rate=441 Hz for an *a*-to-*k* ratio of 100), many instruments sound significantly different when run at k-rate = a-rate than they do in "debugging mode," a feature which has caused this author many hours of consternation. Another unfortunate detail (that this author only became aware of very recently), is that the ratio of the a-rate to the k-rate (the size of the a-rate inner computation loop, if you

will) is also used as the unit generator computation loop size, so that if you run your instruments at k-rate = a-rate the sound really good, but run *extremely* slowly. The recommended work-around it to re-write the instruments after they're debugged so that they use only a-rate signals, and then run with a large k- to a-rate ratio (to get large inner computational loops without slowly-sampled control functions).

Csound's orchestra language also allows one to embed arbitrarily-complex expressions in the fields of instrument definition statements (e.g.,

```
a1 oscil (p5/p6*21.402), p9, 2
```

This allows for quite terse and compact instruments. A number of useful shorthand notations for musical pitches and loudness values are also provided, such as *octave.semitone* notation (where 8.2 would mean *d* in the 8th octave), and *octave.cents* notation (where 8.250 would mean *d* + one quarter tone in octave 8). These notations are provided by the `octpch` and `pchcps` conversion statements that can be used in instrument definitions. The `ampDB` operator converts positive dB values in the range 0-96 dB to integer amplitudes in the range 0-32767.

Csound has both truncating and interpolating oscillators in terms of the `oscil` and `oscili` unit generator statements. A special unit generator for single-carrier FM is also provided in the form of the `foscil` and `foscili` statements (see instrument `i2` in Fig. 11). Sample-and-hold or interpolating random value generators are available to generate control or audio noisy signals. Generators are provided that can read analysis data for additive synthesis (phase vocoder or Fourier resynthesis) or subtractive synthesis (linear predictive vocoder or filter-based resynthesis). A powerful and easy to use set of filters statements (low-pass, band-pass, delay lines, etc.) allows users to build various filters and resonators with relative ease. Csound also has very flexible envelope generator facilities in the `linseg` (linear line segment function) and `expseg` (exponential line segment function) generators. The `GEN` routines allow many description modes for table data.

One can also use limited assembly-language control structures in instruments, having `goto` statements with named labels that can be executed on a per-note, per-control-period, or per-sample-period basis, or special variables that are used when notes are "tied into."

Because of the efficiency of Csound, it is possible on some machines—fast RISC workstations—to run simple orchestras in real time—possibly under MIDI control—writing sample data directly to the DAC output rather than to disk.

As mentioned above, in the version described here (as in Music-11), data in Csound scores are purely numerical; this is a feature that this author has always considered more of a problem. The advantage of this is the easy construction of score processing libraries

and score preprocessors, but it also means that even external files must be referred to numerically, e.g., soundfiles must be named "soundfile.*x*" where the *x* is an integer that can be passed in a score. All parameter translation information is embedded in the instruments, which will often be needed in several different versions for use with different score formats. These conversions are compiled into the instrument to avoid needing units in the score file. Several higher-level score preprocessors exist for Csound, most notably scot (a compact and flexible music input language), SCORE-11 (Alexander Brinkman's implementation of Leland Smith's SCORE language), and cscore (a wonderful C library for writing score manipulation programs). Csound also implements a "carry" feature in scores whereby all parameter values that remain unchanged between two note commands can be written as "+" in the second one. This often makes it much easier to see what is relevant in a score by making unchanging parameters "invisible."

The multi-pass `Csound` command allows users to save sorted scores, and there are "user friendly" versions of it on systems with more modern user interfaces than the UNIX shell. It is not, however, possible (as it was in Music-11), to save compiled orchestras for faster processing. There are interfaces to a number of external interfaces to DSP tools such as spectral analysis packages, and phase and linear prediction vocoders, but these tools are not integrated into Csound, and they generally have different user interfaces. The Csound source code comprises approximately 20,000 lines of C source, including real-time output and graphical display routines for several systems. The comments are of varying usage and several apparent programming styles. It is not recommended for non-experts to attempt to extend or customize this package.

The Csound system comes with excellent documentation, including a very useful beginner's tutorial and a quick-reference guide. There is also a significant collected literature in terms of existing orchestras included with the distribution (many of which are not commented well enough to be of any use, however). Csound is known as a fast, relatively simple, and very portable (even to personal computers as in the Composer's Desktop Project's CDP port), SWSS package that is widely used in computer music centers and in the homes of many composers. The Csound system is available free for UNIX, NeXT, and Macintosh systems via anonymous network file transfer (Internet ftp) from the machine *media-lab.mit.edu*. Any user at a university or major corporation, or connected to a commercial network should be able to get it via some ftp remote file transfer program.

Since the writing of this manuscript, a new version has ben released that incorporates the ability to use file names in scores, and other enhancements. The system has also been

ported to new platforms, such as recent commercial UNIX workstations from Silicon Graphics and Digital Equipment Corp.

Cmusic

The cmusic language was developed by F. Richard (Dick) Moore at the Computer Audio Research Laboratory of the Center for Music Experiment (now the Center for Research in Computing and the Arts), of the University of California, San Diego (that's CARL at CME [now CRCA] at UCSD), starting in the late 1970s. The cmusic language is a reimplementation of a very Music V-like language (Moore had worked with Max Mathews at Bell Labs) for use within the CARL software environment, and was originally developed as an example only, finding its way into regular use at a number of centers only gradually.

The CARL software distribution is a large and sophisticated suite of C-based SWSS and DSP software tools originally used on DEC VAX-11 computers, and later ported to the UNIX-based workstations of the early- to mid-1980s. It includes the "Csound" sound file system (not to be mistaken for the Csound SWSS language discussed above), and uses C as its primary model and extension language. Moore described the motivations for the CARL software design this way:

> "Aside from its UNIX and Music V heritage, cmusic was an attempt to create as flexible and general a software synthesis system as possible, sometimes sacrificing speed for generality or ease of user extension. The unit generators, for example, are all coded in C, allowing additions or modifications to be made as easily as possible. This is based on the premise that improvements in speed are much more a function of hardware than software. This premise also makes it desirable to be as hardware-independent as possible. The environment of the CARL software was chosen, therefore, to be UNIX and C rather than any particular hardware. Some portions of the CARL software—such as the Csound soundfile system (yes, the CARL soundfile system has had that name since 1979, much to the confusion of many) written primarily by Gareth Loy—are necessarily less hardware independent. Cmusic, on the other hand, is just a program, so it has been relatively easy to implement on many different systems. Note, however, that cmusic uses the UNIX pipe mechanism not only for its output signal (and optionally for its score file input), but also to intercommunicate with function generators. This again allows easy user extension of available generators (such programs are used to fill cmusic wave tables with data). It also prevents cmusic from running (without modification) on platforms that support C but not UNIX.
>
> The entire CARL package contains many dozens of programs that all are designed in a way that allows them to be used together in a relatively coherent way by anyone who understands and is comfortable with the UNIX environment. [...] A CARL program called "gain" that reads a digital signal from its standard input and writes a signal on its standard output that has been multiplied by a scale factor given as an argument to the program. All piped signals are normally in floating-point format, so clipping is not performed by the gain program. The following UNIX command, then, reads a signal from a soundfile called

> *loud_sound*, reduces its amplitude by a factor of about 8 (18 dB), and writes the result into another soundfile called *soft_sound*:
>
> ```
> sndin loud_sound | gain -18dB | sndout soft_sound
> ```
>
> The `sndin` and `sndout` programs [now called *fromsnd* and *tosnd*] are part of the Csound soundfile-management environment. [...] This environment was later improved and extended in collaboration with other centers into what is now known as the "BICSF" (for Berkeley-IRCAM-CARL Sound File) system, which is now part of the CARL software." (Moore 1991)

The cmusic language model is a mixture of C, csh (the cshell interpreter), and cpp (the C preprocessor); it is not fully compatible with any of them, but offers a welcome "raising" of the model when compared to Csound's orientation towards assembly language and FORTRAN. When using cmusic, one can write instruments and scores in the same file or use cpp's `#include` directive to include an orchestra file into the score when running the program. Compile-time macros can be built in via cpp's `#define` directive (a flexible macro definition facility). The cmusic program runs all input files through cpp, so users can have very sophisticated predefined functions as macros, for example a note name convention that turns note names into frequencies. Cmusic input files start with any `#include` statements (such as one that reads in the main shared CARL header file), followed by optional statements that set certain system variables. The syntax of these statements is simple (e.g., `set stereo;` to tell the system to create a stereo sound file), and reasonable defaults are assumed. The typical variables are the sample rate, the number of output channels, the shape and size of the space used for the reverberator models, and the file name(s) for the output sound file(s). After this, one customarily has one or more macro definitions, which set up useful translations (e.g., parameter-maps), followed by the instrument definitions (or #inclusion of the orchestra source file). This can be followed in the same file by the function definitions and the score in one or more sections. The cmusic instrument definitions and scores for the two example instruments are shown in Figs. 7 (above) and 15.

```
{  Cmusic Sound File Reader Instrument
   sndin--Mono-input to stereo instrument with att/dec
      p1    p2    p3    p4    p5    p6    p7    p8
      note start instr  dur   ampl  att   dec   pos
}

#include <carl/cmusic.h>
set rate = 44100;
set channels = 2;

#define  AMP   p5
#define  ATT   p6
```

```
#define  DEC   p7
#define  POS   p8
#define  ENV   b2
#define  SND   b3
#define  LEFT  b4
#define  RIGHT b5

ins 0 sndin; { mono to stereo with fade in and out }
                        { soundfile reader }
   sndfile b1 AMP 1.0 s1 0 0 -1 d d d d d;
   seg   ENV 1.0 f1 d ATT 0 DEC; { envelope in f1 }
   mult  SND b1 ENV;    { envelope mult }
   mult  LEFT SND POS;  { left mult }
   inv   b8 POS;        { right factor }
   mult  RIGHT SND b8;  { right mult }
   out   LEFT RIGHT;    { stereo out }
end;

               { exponential attack/decay function }
gen 0 gen4 f1 0, 0 (2)  1/3, 1 (2)  2/3, 1 (4)  1, 0;
var 0 s1 "../bell.snd"; { file name variable }

#define BDUR 4 { define duration of sound }
set verbose;{ verbose score execution }

{  start instr dur   ampl  att   dec   pos; }
note 0   sndin BDUR  1.0   0.001 0.5   0.5;
note 2   sndin BDUR  1.0   1.0   0.5   0.1;
note 5   sndin BDUR  0.2   2.0   2.5   0.9;
ter;
```

Figure 15: Sound processing example and sample score in cmusic

To run this file, one uses the shell command `cmusic`, which takes as its arguments the orchestra/score file name and optional command-line switches to set the sample rate, block/buffer sizes, and various processing options. Cmusic can write the output samples to the UNIX "standard output" (for use with an external `tosnd` program), or directly to a sound file (if the orchestra or score includes a `set outputfile = file_name;` statement). The same kind of *play* program used with Csound is needed here. Because the cmusic compiler can write samples to UNIX pipes, it can easily be connected to other programs, such as the "outboard" reverberator used (in good CARL style) in the third line in Fig. 16. Variables set from the cmusic command line override values set in the score file (*very* handy).

```
cmusic fm1.sc | tosnd -h fm1.snd
play fm1.snd
cmusic -f fm1.sc | reverb 0.994 | tosnd -f fm1.reverb.snd
```

Figure 16: Steps in Csound processing expressed as UNIX commands

The CARL score processing and DSP software environment includes a wide variety of programs for sound spatialization, analysis/synthesis, etc. These programs generally read and write floating-point sound samples to/from their UNIX standard I/O streams, and can therefore be "piped" together in many ways. The various sound sampling, analysis/synthesis, mixing, localization, reverberation, and DSP development tools are also available as C functions in the function library "libcarl.a," and the development of new DSP programs is well-supported for a "roll-your-own" approach that is in good keeping with the UNIX (and more recently, GNU) style.

The csound sound file system, and its descendant the BICSF system, define a header format for sampled sound files, and provide a set of utility programs for (e.g.,) format conversion, recording and playback, file management and annotation, and analysis and feature extraction.

The cmusic instrument definition language supports named (rather than numbered) instruments and symbolic units via post-operators (e.g., `p5 Hz` to convert parameter 5 from Hertz into a sample table increment). Variables can be declared with types by name as being audio-rate (b-variables are buffers), note-rate (note p-fields), or constant (instrument v- or s-variables). There are however, no variable expressions allowed in unit generator fields, so cmusic has special unit generators for addition, multiplication, etc. according to the Music V model (a minor nuisance). One can write a constant expression such as `440*2^(v6/12)Hz`, but not one that uses a p-field such as `p6*440Hz`. Truncating and interpolating oscillators are available in cmusic. There are post-operators for many common units (e.g., `Hz`, `sec`) available for use in cmusic scores, which make instruments much simpler and place more information in the score (relative to the Csound convention of having the conversions be part of the instruments). The operator for dB conversion scales 0 dB to a floating-point sample of 1.0, with -6 dB being 0.5, -12 dB 0.25, etc. (much more natural to this user than the positive scale).

Cmusic provides a sound spatialization unit generator called `SPACE`, with which one can define sound paths in arbitrary simulated rooms. The cmusic `GEN` programs are run as separate UNIX programs that are called—and whose output is read—by the cmusic driver program. This means that the set of generators is extensible without changing the cmusic program itself. The list of available generator routines includes all of the standards as well as a Chebyshev polynomial table generator (for wave-shaping or nonlinear distortion synthesis), a Shepard tone envelope generator, a quadraphonic sound file path interpreter, and more.

One of the most important features of cmusic (from the point of view of the user) is the integration of the sound compiler with all the other CARL software tools, such as the

Mark Dolson vocoders, the large collection of DSP utilities (including an implementation of Andy Moorer's tapped delay line/low-pass reverberator), and several composition and music description languages. Cmusic also includes simple general-purpose filters (for use with the CARL filter design programs). The more one works in this environment, the less one seems to use the sound compiler, resorting more and more to the standard CARL sound file and DSP utilities or custom-built tools. The extensibility of the system is excellent; there are documents about "Writing CARL Programs," source code for sample programs, and documentation of the library functions. The cmusic source itself is rather difficult to read due to the heavy use of non-obvious macros (defined in a wide array of header files), and the use of macros that include partial control structures (e.g., `#define OPEN int f; {`), something last seen in the Bourne shell or the various C obfuscation contests that have become popular of late. The source for the main CARL library and the cmusic program itself are each about 5000 lines of C (with some FORTRAN for the IEEE DSP libraries). The utilities make up approximately 25,000 lines of C source.

The CARL environment has been ported to many UNIX-based computers, and is easily compiled and extended onto new platforms. The CARL system is documented in several forms; first, there are the early 1980s reports and papers by Moore, Loy, Dolson, et al. that describe the various facets of the system and the tools developed under it. Moore's book *The Elements of Computer Music* (1990) is an impressive and exhaustive introduction to computer music, SWSS, and DSP, and is heartily recommended to anyone interested in the field (see John Snell's review of it in *Computer Music Journal* 14:4). It introduces cmusic and provides many usage examples and a reference manual. A related introduction to C programming on UNIX systems can also be found in *Programming in C with a Bit of UNIX* (Moore 1985), a good prerequisite to the *Elements* book.

The CARL software distribution can be obtained for a small licensing fee from the CARL laboratory; for more information, contact the Center for Research in Computing and the Arts, Q-037, University of California, San Diego, La Jolla, California 92093-0037 USA; electronic mail frm@ucsd.edu.

Cmix

The third SWSS system (and the one with the most direct "New Jersey connection"), is cmix, developed in several stages since the mid-1980's by Paul Lansky at Princeton University. Cmix is a new version of Lansky's soundfile mixing programs that trace their heritage back to the days when he programmed on punched cards and stored samples onto 7-track magnetic tape. The original MIX program (written in FORTRAN), was used

for sound file mixing and voice processing. Cmix developed over the years as Lansky gained access to powerful UNIX workstations.

Cmix is different from typical Music V-style programs in two important respects. First, the cmix package is not really a program at all, but rather a collection of C functions that can be used in user-written C programs (see the examples below). The second difference is that cmix has no scheduler; there is no need to sort events because a cmix program is free to write samples to one or more output files at any points and in any order. The system calculates one event at a time and may either write its samples directly to the output file or add them to the values that are already there in a read/modify/write operation. Paul Lansky described the reasons for the differences between cmix and Music V-style SWSS tools in the following manner:

> "I had gotten tired of rerunning huge jobs, just to correct one note here or there and I wanted to model my computer work on the workings of an efficient chamber music rehearsal, where the conductor will fix spots and instruments without having to restart everything whenever something went wrong. At that time I felt that the way I was working was analogous to a situation in which when the clarinet played a wrong note in bar 349, the rehearsal stopped and you began again from the beginning with everyone playing, hoping that he would get it right. This way of working on that system, which was overwhelmingly cumbersome to begin with, was just too time consuming. So I wrote a program I called MIX, which was essentially a 20 track mixer. I could perfect each part, lay it out in a file or on a track, and then do mixes, fixing a note here or there until I got it right. At a certain point it became evident to me that I could easily do synthesis and processing in this program, as well as mixing, and began to do instrument design, etc. Then I decided that it was going to be a while until these things were real-time, so I abandoned a scheduler altogether and just used random access to address the disk. This eliminated a lot of the headaches associated with a multi-pass system such as Music IVBF." (Lansky 1991)

Cmix orchestra files are regular C programs that use the library's powerful built-in functions, or any user-defined C functions. The "programming language" model for SWSS introduced above is interpreted literally here; unit generators are C-language functions that are configured into higher-level instrument definition functions. The score is read as a sequence of function calls to instruments that take formal parameter lists and have random read/modify/write access to one or more sound files. There are cmix procedures that correspond roughly to all of the unit generators and `GEN` routines of Music V-style languages, as well as a large collection of extended functions. The user program's `main()` function is provided by the cmix system library and reads note statements from the program's standard input (possibly a score file), calling instrument functions as appropriate. One builds new instruments or unit generators by writing functions that generate or manipulate sound buffers or sound files. The file format for a cmix orchestra is the same as for standard C programs, with the exceptions that one

includes the cmix system's header files, and does not define the `main()` function. Instrument functions accept arrays of floating-point numbers (the note p-fields) as arguments. The `profile`() function (supplied by the user), associates these functions with score file statements so that the input reader can use them to interpret score files. This format is shown in Fig. 17, the outline of a cmix program and MINC score. Figure 18 shows the program for the FM example, a C-function named `fm` that uses the library's oscillator function as a unit generator, and a `profile` function. Note that comments in C are delimited by the strings "`/*`" and "`*/`" (non-nesting).

Cmix scores are written in the *MINC* (a recursive acronym for "MINC is not C"), language. MINC is very C-like and allows typed variable declarations, score function calls, and a variety of control structures. This is by far the most powerful score language this author has seen in a Music V-like SWSS system. MINC functions (instruments), can return values, and score statements can contain programming language operators and control structures such as branching and loops. Paul Lansky said of MINC, "I have found that the front end, which is like an interpreted C language is also a very powerful composition language and this has allowed me to expand far beyond the note list-type environment. I have quite frequently used the MINC front-end to drive a variety of programs" (Lansky 1991). A cmix score file (in MINC format) looks like a cross between a C program and a csh script, and can contain its own function definitions. The `system()` command executes its argument (a string) as UNIX shell command. In the figure one creates an empty output file (using the `sfcreate` utility, part of cmix that creates sound header files given a sampling rate, the number of channels and the storage format), sets it as the default output file for the score, declares temporary variables for the score, and calls instrument functions in any order. A typical MINC data file for our FM instrument is shown in Fig. 19, which creates an output file header with the desired properties, then creates the function tables and calls the FM instrument (function) several times. Note that one does not typically create empty sound file header with each execution of a score, but tends to "reuse" files in the read/modify/write pattern described by Paul Lansky above. The fact that cmix reads the sound file format from the pre-existing header provides a very nice type-safety in that the file format is always maintained, providing, e.g., automatic recognition of integer or floating-point input or output files.

```
/* cmix C file */
   include cmix header files
   global declarations
   C-functions for "unit generators" or "instruments"
   profile() function
```

```
   /* cmix MINC file */
      global variable declarations
      create and/or open output sound file
      play notes (i.e., call C functions)
```

Figure 17: Format of a cmix program

```
/****** cmix fm instrument file ******/
#include <cmix/ugens.h>    /* include cmixheader files */
#include <cmix/macros.h>

int NBYTES = 32768;        /* set the I/O buffer size */

/* fm:   p[0] = start, p[1] = dur, p[2] = amp, p[3] = freq,
 *       p[4] = c:m, p[5] = index, p[6,7] = ampl. att/dec
 *       p[8,9] = index att/decay
 */
double fm(p,n_args) float *p; int n_args; {
   float sicar, simod, *sine, evals[6], ivals[6], modphs, carphs,
         *risef, *decayf, indFcn, ampFcn, diff, mod, car, ampl;
   int len1, nsamps, i;

   sine = floc(1);         /* fcn 1 is sine wave */
   decayf = risef = floc(2);  /* fcn 2 is att/dec shape */
   len1 = fsize(1);        /* lookup size of f1 */
                           /* scale Hz to sample inc. */
   sicar = cpspch(p[3])*(len1/SR);
   simod = sicar * p[4];   /* scale the mod. freq */
   modphs = carphs = 0;    /* initialize phases */
   ampl = ampdb(p[2]);     /* convert p5 from dB */
   nsamps = setnote(p[0], p[1], 1);    /* set up the note */
   evset(p[1], p[6], p[7], 2, evals);  /* set ampl envelope */
   evset(p[1], p[8], p[9], 2, ivals);  /* set index envelope */

                           /* inner sample loop */
   for(i=0; i<nsamps; i++) {
                           /* update amplitude envelope */
      ampFcn = evp(i, risef, decayf, evals) * ampl;
                           /* update index envelope */
      indFcn = evp(i, risef, decayf, ivals) * p[5];
                           /* modulator oscillator */
      mod = oscili(indFcn, simod, sine, len1, &modphs);
                           /* carrier oscillator */
      car = oscilni(ampFcn, sicar+mod, sine, len1, &carphs);
      ADDOUT(&car, 1);     /* additive writes to file 1 */
   }
   endnote(1);             /* save note to disk after loop */
}

/** profile function--introduce function name to MINC scanner **/
profile() {
   UG_INTRO("fm", fm);
}
```

Figure 18: FM instrument in cmix C

```
/* cmix MINC data file for fm instrument
 *    p0 = start, p1 = dur, p2 = amp, p3 = freq, p4 = c:m ratio,
 *    p5 = mod index, p6&7 = attack and decay times,
 *    p8&9 = index attack and decay times
 * function 1 is sine wave, function 2 is index guide, 3 is envelope
 */
           /* create an empty 44.1 kHz mono integer sound file */
system("sfcreate -r 44100 -c 1 -i fm1.snd")
output("fm1.snd")    /* set "fm1.snd" as the output */

makegen(1, 10, 1024, 1)          /* f1 = sine wave */
makegen(2, 7, 1024, 0, 1024, 1)  /* f2 = expon. attack/decay */

           /* declare names for p-field variables */
float start, dur, pch, amp, ratio, att, dec, i_att, i_dec;
start=0        dur=2    amp = 30000    pch=100     ratio=1
ind=1          att=.1   dec=.2         i_att=.05   i_dec=.4

/* p[0] p1 p2   p3   p4     p5 p6   p7   p8p9*/
fm(0,dur, amp, pch, ratio, 1, att, dec, i_att, i_dec)
fm(2,dur, amp, pch, ratio, 3, att, dec, i_att, i_dec)
```

Figure 19: MINC data file for the cmix FM instrument

The steps to execute a cmix program involve using the C compiler "cc" to compile the orchestra as a normal C program, and then running the resulting program with its standard input taken from the score file as shown in Fig. 20. The `-o filename` option in the `cc` command line tells the compiler to place the executable output of the compilation in the file named `filename` (`fm` in the example). The `-O` option tells the C compiler to optimize the program, and the source file name (`fm1.c`), is given next. The last two file names in this command are the general cmix function library and the table generator library. The next two steps are to execute the resulting program with the input taken from the data file (named `fm1.m`)—i.e., run the orchestra with the score as its input—and to play the output file, as shown in Fig. 20.

```
cc -o fm1 -O fm1.c ../lib/cmix.o ../lib/genlib.a
fm1 < fm1.m
play fm1.snd
```

Figure 20: Steps to compile and run the cmix FM instrument

The soundfile mixing example will look different in cmix than in the more Music V-like Csound and cmusic. This type of sound file layering instrument is what the cmix system was originally developed for, and there is a stand-alone mixing program provided with the cmix distribution. This mixer reads MINC files (or interactive input), and responds to commands that open various input files, and read and distribute sound onto multichannel output files. A script for this in MINC is shown in Fig. 22, where one can see the use of the C-preprocessor's macro definition facility to make a single-line note

command according to our requirements. One could of course write a special C function for this mix command, and compile it in cmix for possible better performance—an exercise that is left for the advanced reader.

```
/***** cmix MINC data file for mix instrument *****/
output("snd.snd")

/* make a cpp macro for note command */
#define NOTE(ST, DUR, NAME, ATT, DEC, POS) \
   input(NAME); \       /* open input file */
                        /* set envelope times */
   setline(ST, 0, (ST+ATT), 1, (ST+DUR-DEC), 1, (ST+DUR), 0); \
                        /* play input file */
   stereo(0, ST, (ST+DUR), AMP, POS);

NOTE(0, 4, "../bell.snd", 0.001, 0.5, 0.2);
NOTE(2, 1, "../voice.snd", 0.0001, 0.1, 0.8)
```

Figure 21: Sound file mixer written for mix in MINC

There are important advantages and disadvantages to the use of C as the base programming language for cmix. The flexibility and extensibility of cmix is enhanced, while there is more "cognitive overhead" involved in using cmix relative to SWSS tools that are not real programming languages. With respect to the definition of SWSS given above, cmix definitely offers the "Best Eats in Town" in terms of easiest extension and customization of both the palette of unit generators and the score language. The connection of cmix programs with other programs written using the system (such as the *rooms* reverberation and spatialization package and Paul Lansky's famous *lpc* linear prediction vocoder tools) is excellent. These are available as stand-alone programs (e.g., for lpc analysis, analysis data manipulation and re- or cross-synthesis), or (of course) in source code formats. For advanced users, the custom of "Use the Source, Luke" is quite a powerful mechanism. There are also a number of separate tools, e.g., for filter design.

The cmix system also includes an extended version of the BICSF sound file system for use on Sun SPARCstation and NeXT computers. In this system, the sound file headers start with standard NeXT/Sun headers, which are followed by a "comment" that contains a BICSF sound file header. This allows the files to be manipulated by the host machine's sound utilities (e.g., record, play, and lsf), but also lets the user store such information as the file's maximum amplitude (and the sample it occurs on), and documentation comments in the BICSF header. There are cmix utilities for creating empty sound files (with header information filled in as described above), for listing and sorting them, and for editing the comments they hold.

MINC is indeed a very powerful score language, which includes control structures (e.g., loops) and other "real programming language" features. This is much preferable to

score input formats that are less flexible (e.g., raw note lists). The subtle differences between MINC and C will, however, eventually "bite" every serious user.

Cmix really isn't a Music V-style "sound compiler" but rather a "kit" for building DSP and SWSS programs, some of which may look and behave like Music V-style instrument definitions. This is, however, not what it was designed for, and not the typical mode of using it. It is much better suited for a more *musique concrète* style of working (which is getting more popular as it becomes more practicable on general-purpose hardware). The fact that cmix has no sorter or scheduler is also a mixed blessing; while this makes the system much simpler (and therefore more easily extensible), it also makes for much higher I/O overhead, e.g., when mixing many soundfiles together, since only one "note" is computed at a time. The system must then make many passes over the same set of samples, performing read/add/write operations to each output block for each sound that is active during it.

As with both of the other systems described above, the source code for cmix is a bit "idiosyncratically punctuated, formatted, and commented" (that's being polite), and I advise any user to write a shell script that will run all of the header and code files through the "C beautifier," "auto-indenter," or similar tool before diving in. The central cmix libraries comprise approximately 5000 lines of source code. The advanced applications (e.g., the lpc vocoder or sound file mixer) are quite compact due to the power of the library. The cmix documentation consists of a series of manual pages for the various built-in functions and utilities, and a few annotated examples; what is missing is a good tutorial or better example set, but this is partially mitigated by the choice of C as a base language.

Cmix is very portable among UNIX workstations and various personal computers, though it requires a Berkeley UNIX-style file system (i.e., one that allows files to have "holes" in them). Several of the add-on tools (such as the lpc editors) are written with NeXT user interfaces (a disadvantage for this user). The system, and several other applications, are available via anonymous InterNet file transfer from the directory pub/music on the network file server princeton.edu.

The Future of SWSS

There are several technical trends in SWSS systems seen in the three systems discussed above. The first is the move towards greater portability using the high degree of standardization of the C programming language. Actually, this trend started with Music V, whose portability was supported by the portability of early FORTRAN systems. This is, however, being undercut by the construction of ever more sophisticated non-portable user interfaces such as the X Windows-based tools for Csound or the NeXT-based tools

for cmix. Another trend is toward the integration of ever more powerful analysis-resynthesis and score generation tools into SWSS systems. All three systems presented in this article have interfaces to additive and subtractive vocoders, for example. An obvious trend is the inclusion of new synthesis techniques into SWSS programs. Examples of this are the Karplus-Strong plucked string algorithm, for which unit generators exist in several systems. This can be expected to continue, with, e.g., the provision of facilities for the construction of synthesis methods based on physical modeling.

The "higher level" trends are less clearly defined, but the integration of SWSS systems into higher-level programming languages can be hoped for. Recent developments in this area are Common Lisp Music (Schottstaedt 1992), and the MODE (Pope 1992)—two systems that run on several platforms and provide for optional DSP coprocessing support. The clm system seems very interesting for several reasons; first, it is written in Common Lisp, a high level, abstract programming language (relative to C, at least), which supports interpreted or incrementally-compiled execution. The orchestra and score languages in clm are the same—in Lisp, programs and data have the same format. Clm also makes use of DSP coprocessors (one or more Motorola DSP56001s supported on any of several platforms), so that users can enjoy the abstraction of a high-level programming tool along with the performance of a parallel multi-DSP system. Heinrich (Rick) Taube's Common Music package (Taube 1991), is an extensible high-level score description language written in Common Lisp for clm. The standard development environment for clm and Common Music is still, unfortunately, a multi-window text-based front-end using the emacs text editor in Lisp interpreter view—a fact that is being addressed by its developers at present. The MODE system supports unit-generator-based timbre description and hierarchical score generation in the Smalltalk-80 language and programming environment. It provides a variety of multi-paned text-based and graphical user interface tools for interaction with signal and event data.

When compared to other music software tools (e.g., hardware synthesizers or digital mixers), SWSS systems come the closest to being "all things to all people" in terms of their support for a wide range of musical activities and their applicability in a number of different situations. Relative to software libraries for sound support (e.g., those of the Macintosh or NeXT computers), SWSS systems provide a higher level of abstraction (in most cases), and easier start-up for novices.

Comparisons and Benchmark Tests

This section presents several benchmark tests of, and subjective comments about, the three software sound synthesis (SWSS) systems Csound, cmusic and cmix. A series of

performance benchmarks were run in order to measure the relative performance of the systems using the two example instruments introduced above—one single-modulator frequency modulation (FM) instrument, and one mono-to-stereo sound file mixing/layering instrument—and several different scores. The tests were run on two different machines in order to average out some of the effects of special architectural features or compiler optimizations. The eight benchmark tests were chosen to illustrate the performance of the systems under a mixed set of conditions. The second comparison of the three systems is based on a set of subjective criteria related to the instrument definition and score languages, the system usage, and the integration of the system with the programming environment. It is more informal, and is written in the first person.

Benchmarks

The benchmark tests use the two example instruments presented above. These tests are not truly rigorous, and represent approximate comparisons of the three SWSS packages under discussion, rather than as relative benchmarks of the two platforms. The tests were run in early 1992 on the then-current versions of all SWSS packages. Both of the instruments were run several times on two different workstation-class computers of the current generation (realizing that both manufacturers have come out with new, faster models since this article was written).

The benchmark platforms were: (a) a SPARCstation-2/IPX with 28 Mb of RAM memory running SunOS 4.1.1, and (b) a NeXT 68040 TurboCube with 16 Mb RAM, version 2.1 of the NeXT system software. Both machines had local disk storage, and were "relatively idle" so that the overhead of the windowing system and UNIX background daemons was minimized. In all cases the programs were compiled with the platform's standard C compilers, and optimized with the -O compiler flag. The tests were run (except as noted below), with a sampling rate of 44.1kHz, a control rate of 44.1kHz, function table length of 1024, using interpolating oscillators and writing sound output directly to sound files.

The benchmarks that were run for each system are listed in Table 1. They compare the start-up time, the per-note overhead, the raw computing speed, and the I/O performance of each system on each platform using the example instruments with various scores. The FM instrument was run with a one-note score in test 1—to measure the overhead involved in compiling and running a score—with one long note (test 2)—to gauge the average system compute performance—with many short staggered and overlapping notes (test 3)—to check the note start-up overhead—and with eight staggered overlapping long notes—to determine the sample buffer mixing processing time. The

sound file mixing example was used with two scores in tests 5 and 7, the first of which had many short notes from several sound files—to point out file opening, seeking, and closing costs—and the second of which used eight staggered long notes—to stress disk I/O overhead.

The final two tests (7 and 8), compare the options for improving compute ratios of SWSS systems. The effect of different control and audio rates (test 7 run on Csound and cmix only), and of truncating (versus interpolating) oscillators (test 8 run on all three packages), are measured here. Benchmark 7 was not run in cmusic because it does not provide control-rate variables. The results of the benchmarks are tabulated in Table 2, which shows the absolute real time for the commands as reported by the UNIX system's `time` utility. Table 3 shows this data expressed as "compute ratios," the ratio between the time need to calculate a sound and the sound's duration—the traditional unit of measure of SWSS performance. Compute ratios lower than 1.0 mean that sounds can be computed in real-time. Figure 1 shows the compute ratio data in bar-graph form. The horizontal axis in Fig. 22 shows the eight benchmark tests that were run; within each test, the three data values are for the Csound, cmusic and cmix packages respectively. The values shown in Fig. 22 are the harmonic mean of the SPARC and NeXT data—the square-root of the product of the compute ratios. Figure 22(a) shows this data on a linear scale, while Fig. 22(b) uses a logarithmic scale.

```
Standard Tests
   [1] FM example, one short note (0.5 sec)
      measures start-up time overhead
   [2] FM example, one long note (30 sec)
      measures "raw" compute speed for oscillators
   [3] FM example, many short notes (512 notes spaced over 43 sec)
      measures scheduling overhead for notes
   [4] FM example, 8 long notes (staggered over 37 sec)
      measures mixing overhead
   [5] Mixer example, many short notes (512 notes over 255 sec)
      measures sound file I/O overhead
   [6] Mixer example, 8 long notes (staggered over 8.25 sec)
      measures sound file I/O speed

Tuning Tests—using benchmark 2 FM example, one long note (30 sec)
   [7] k-rate = a-rate / 100 (csound, cmix only)
      measures tuning possible with k- vs. a-rate
   [8] truncating oscillators and 8k functions
      measures speedup when using truncating oscillators
```

Table 1: Benchmark list

	1	2	3	4	5	6	7	8
Csound								
SPARC	1.3	49.0	185.5	231.4	952.4	49.8	21.2	37.8
NeXT	2.6	64.0	344.3	477.8	1161.2	71.7	23.0	57.3
Cmusic								
SPARC	1.4	44.0	192.8	254.9	660.9	157.1		38.1
NeXT	6.0	274.0	1235.9	1685.7	1807.5	404.0		259.5
Cmix								
SPARC	4.4	47.3	245.1	323.6	690.3	25.8	177.9	36.4
NeXT	6.1	76.5	893.2	588.7	3033.9	82.2	96.1	67.8

Table 2: Benchmark results expressed as absolute execution times

	1	2	3	4	5	6	7	8
Csound								
SPARC	2.60	1.63	4.31	6.25	3.73	6.04	0.71	1.26
NeXT	5.20	2.13	8.01	12.91	4.55	8.69	0.77	1.91
Cmusic								
SPARC	2.80	1.47	4.48	6.89	2.59	19.04		1.27
NeXT	12.00	9.13	28.74	45.56	7.09	49.0		8.65
Cmix								
SPARC	8.80	1.58	5.70	8.75	2.71	3.13	5.93	1.21
NeXT	12.20	2.55	20.77	15.91	11.9	9.96	3.2	2.26

Table 3: Benchmark results expressed as compute ratios

(Not enclosed here.)

Figure 22: Graph of benchmark results

Discussion

There are several surprises in these results—based on expectations given published benchmarks for the two platforms, and the "common knowledge" in the computer music community about performance. First, I had expected a much larger difference between the three systems under test, and second, I had not expected such a large difference between the SPARC and NeXT platforms. As the data in Table 2 and Fig. 22 show, the three systems are within a factor of two of each other on most tests on both platforms. Cmix shows significantly worse results for benchmark 1 because of the overhead of using the multi-pass C compiler for instrument definitions. This is more than made up for by the fact that one can store compiled orchestras (as compiled and linked cmix programs), for use in the more laborious task of score debugging. I am at a loss to explain the relatively worse performance of cmusic on the NeXT platform, which seems equally slower on the compute-intensive tests such as benchmark 2 and 4—where the

measurements reflect the raw floating-point performance of the platform—and the I/O-intensive tests such as 5 and 6—where disk I/O speed is the dominant factor. Even more surprising are the benchmarks where the NeXT and SPARC platforms ran approximately equally fast, such as benchmark 7 where the NeXt was even faster than the SPARC for the cmix version.

Benchmark tests 7 and 8 demonstrate the effect of using the two main optimizations found in SWSS languages: using a lower control rate than the audio sample rate; and using truncating (vs. interpolating) table look-up in oscillators (with larger function tables to make up for the lost fidelity). Both tests use the same instrument and score as test 2. The results show that both Csound and cmix run about twice as fast with the k- to a-rate ratio set to 100—running in real-time with one layer of this trivial instrument—and that all three systems achieve a measurable (25 percent), speed-up with truncating oscillators. My evaluation is that this is insufficient to warrant the added programming effort of using control-rate variables or the fidelity considerations of truncating oscillators.

Getting back to the fact that the tests were run in order to judge the relative performance of the three SWSS tools under investigation, the SPARC data show all three systems to be generally comparable in compute ratio, with cmix having the highest start-up overhead and Csound being the overall performance winner by a small margin. Any of these systems can be run with acceptable performance on most current and next-generation workstation-class computers costing $10,000 or less.

One should mention that there are several other tuning options that were not attempted here. These include the use of block unit generators in cmix (which should accelerate the compute-bound tests such as 3 and 4 significantly). The effect of the use of the `foscil` FM unit generator in Csound was tested, and was found to contribute approximately 15 percent improvement for benchmark 2. Other tests that could provide meaningful insights would measure the speed of the systems when writing their output through UNIX pipes to an external tosnd program.

A more formal benchmark test would separate out the system times from the user times in the result analysis, and would include larger, multi-instrument orchestras and multi-section scores. This test could then be run on a wider range of platforms, ranging from personal computers to parallel super-computers. This author (and editor) invites readers to propose additional appropriate benchmark instruments for SWSS tests, to run the instruments given above on other platforms, to translate them into other SWSS languages and test them, and to comment on other languages in the format of this introductory article. The source code for the instruments shown in this article, and the

scores used for the benchmark tests, are available to readers via InterNet ftp file transfer from the directory pub/cmj on the network host ccrma-ftp.stanford.edu.

Feature Comparison

Table 4 shows this author's (very subjective) evaluation of the three SWSS systems; each of the points in the table are described below. The items in the table are rated on a scale of 0 to 10 with 10 meaning "excellent" and 0 meaning "nonexistent." Note that many of these items are issues of taste, musical intention, or computer science religion and that this data reflects many of my own biases or preferences.

Feature	csound	cmusic	cmix
Language model			
level	2	6	8
expressiveness	6	7	8
simplicity	8	7	5
learnability	8	8	5
Orchestra language			
low overhead	8	6	2
control structures	4	3	10
scalability	4	6	8
debuggability	4	5	8
terseness	8	6	4
Score language			
pitch notation	5	6	8
time notation	6	6	6
macros	2	10	10
control structures	2	2	10
score preprocessors	8	6	6
System			
portability	8	5	5
documentation	8	9	5
development speed	6	4	8
execution speed	8	4	8
ability to pre-compile	4	2	8
integration with DSP tools	5	8	8
integration with score tools	6	6	8
integration with graphics tools	4	4	4
extensibility	4	6	8
Totals	128	132	160

Table 4: Feature comparison

The first group of items in Table 4 compares the programming languages used by the three SWSS systems surveyed here. I have rated them according to their "beauty" (to me as a Smalltalk, LISP and [thirdly] C programmer), their expressive power, and their simplicity. The main information here is that Csound is simpler and has less syntactical overhead (i.e., better for non-programmers and novices), but that I find it uglier and less flexible as it is based on assembly language and FORTRAN (e.g., variable typing based on the first character of its name, very limited control structures). Cmix scores high here because it is a "real" programming language (albeit C), and has a powerful language interpreter for scores as well (though not precisely C). With respect to the issue of SWSS systems as "sound compilers," I have a strong bias towards systems that let the user extend the system (e.g., add new unit generators) in the same language in which he/she writes instruments and scores (e.g., cmix, clm, and MODE). The "learnability" rating is a function of each language's simplicity and of the existing tutorial and reference documentation; cmusic and Csound definitely score the highest for this.

The second collection of criteria rates the orchestra languages in terms of their scalability to large instruments or complex algorithms, the ease of debugging instruments, and the compactness or terseness of the instrument definition language. The important factors here are the low programming overhead and terseness of Csound, and the excellent scalability of cmix instruments, meaning that one can use C as a structured language to modularize large instruments. Cmusic suffers here because of the complexity of large instruments due to the necessity of using unit generators such as `add` and `mult` for the addition and multiplication of run-time variables and buffers.

The score languages are compared next as to their models of pitch and time, their provision of a flexible macro facilities and control structures, and the availability of "middle-level" music input languages. All three systems support simple models of pitch (e.g., octave.semitone representation), and support time expressed in "beats" with flexible tempo declaration facilities. Csound has several useful pitch notations, but these are hard-coded into the instrument to avoid units in the score. Cmusic allows the use of C preprocessor macros for named notes in any intonation, and allows post-operators in the score. The provision of macros and control structures in scores is very important in larger scores, and cmix and cmusic win out here. Note that Csound users often use Scot, SCORE, or other score-generation programs (without which large scores are rather unmanageable). The fact that cmusic uses the C preprocessor on all scores is an advantage here; cmix includes MINC—an interpreted C-like language with functions and control structures in scores—an important asset as described above.

The system environments of the three packages are compared last with respect to their portability, development tools, run-time speed, and integration with external score synthesis and signal processing tools. All systems try to provide some portability within/across base implementations in that they include sound file systems and sound storage and I/O utilities, and all can be used as stand-alone sound compiler drivers (vs. porting the whole environment). Csound wins here because of its high degree of portability (including non-UNIX platforms), and fast execution speed. The better integration of the "mix'n'match" tool kits/environments for both cmusic/CARL and "cmix + utilities" are noted here in the integration criteria. Each package includes some collection of add-on score and DSP tools and input languages—e.g., scot and cscore for Csound, the CARL DSP libraries and utilities for cmusic, and mix, rooms, and the lpc vocoder for cmix). All three tools also offer some link to graphical output programs (e.g., for wave table functions or filter parameters). These facilities are more highly developed to run with modern windowing systems on Csound and cmix, for which basic graphical instrument design front-ends ("block diagram compilers") also exist on some systems (see above). The sound file systems also vary greatly, and the cmix version of merged NeXT/SPARC and BICSF file headers is the most portable and flexible.

The interaction modes of the environments and compiler usage, as well as the basic development tools, editors, and data file management functions are comparable for all three systems, being based on the file system and batch compiler front-end used with a text editor, the UNIX shell and the `make` utility. Some menu-based interfaces exist for Csound, but even these are rather crude when seen as computer-aided software engineering (CASE) tools. Only cmix provides any interpreted score input. The speed of the basic development cycle, and the capacities for helping composers cope with large scores are somewhat aided by the C-oriented software tools (e.g., the SCCS source code control system), which are applicable with cmusic and, to a greater extent, cmix. One particular interest to me is the ease of integration of SWSS tools with some rather idiosyncratic Smalltalk-80- or Lisp-based systems (a precondition for my use of any of these in the first place). The experiences with the msh/ARA (in C used as a front-end with Music-11 and later Csound), DoubleTalk (in Smalltalk-80 for cmusic), the HyperScore ToolKit (in Smalltalk-80 for cmusic and MIDI), and the MODE (for the Moore phase vocoder with cmusic, and the Lansky linear predictive vocoder with cmix) are relevant here (see references). All three systems provide orchestra and score formats that can easily be generated and/or parsed by other tools, so that instruments and note lists can be imported or exported from other software packages, such as "block diagram compilers" or higher-level music description languages.

One can see how the systems address the needs of different users by selecting the relevant items from Table 4. A learning user would value features such as the system's simplicity and the quality of the documentation. A user doing "traditional" SWSS (i.e., score-based composition synthesis) on personal computing platforms is likely to be more concerned with system performance and available score input tools. The important comparison criteria for a "power user" doing DSP-based processing are more likely to be the scalability and level of system integration, which can often outweigh the raw compute performance of the system.

Resume

There are several features of "historical" (i.e., mostly obsolete) SWSS systems that are still missing from these three modern packages. Most of these are related to the development/delivery environments in which they are housed. UNIX has a history of weak interactive tools, and all three packages are essentially "batch" systems. The "good old" MUS10 system had a "load-and-go" mode which offered very fast turnaround from within the text editor for simple instruments and scores (extremely useful in debugging). None of the systems presented above provide "Music V interpreters" for the development of instruments of the kind provided on the Kyma system, or Sound Tools. I still miss some sort of interactive DSP workbench in the style of my ancient (1982) mshell program, a "programmable desktop calculator for DSP." This package was an interpreter for a language that included mixed mode mathematics (where operands could be scalars or sample vectors) and built-in functions that resembled Music V-style unit generators. One could use the package either in interactive mode, or have it read pre-edited program files (instrument definitions).

The issue of "language versus program" is also still important, and raises valid criticisms of Csound and cmusic. Again, MUS10 (as extended by Tovar), provided SAIL (an ALGOL-variant) for use within instruments or scores. Of the modern systems, cmix, Music 4C, MODE and Common Lisp Music all provide (to varying extents), unified implementation/orchestra/score languages. The MINC front-end—which allows the design of functions in the score language—and the fact that cmix can read MINC interactively from its UNIX standard input (i.e., the user's terminal keyboard), suggest the development of higher-level graphical or text-based user interfaces to cmix that would offer "the best of both worlds." One can (and should) build modern, graphical interfaces to compiled, extensible SWSS languages.

In comparison with personal computer-based hardware-assisted tools for sound synthesis and processing (e.g., sound samplers, disk-based mixers or MIDI sequencers),

SWSS systems generally offer Dick Moore's definition of "The Best Eats in Town," meaning that while they may involve more overhead to get started with, they scale better to complex timbres and large scores. It should be noted however that for single tasks, dedicated hardware/software solutions are often more effective than general-purpose systems.

I'm sad to say that none of the systems I've used fulfill my wishes completely, or live up to with what I believe we have the technology and the experience to implement. Solutions such as Common Lisp Music and Kyma seem to be living up to this potential more fully, but each suffer from their own problems. Given my preference, and the experience of evaluating these three systems in light of the history of the technology—as if this article were entitled "Software Sound Synthesis Languages I've Known and Loved"—I'd vote today for MUS10 in SAIL on a DEC PDP-20, or MODE in Smalltalk-80 on a network of fast workstations. The modern approach would be to put the system in a studio together with a sampling synthesizer, a disk-based hardware-assisted mixing system, several outboard effects devices, and connect it all via MIDI, computer, and audio networks.

As a way of avoiding the final question of which of these systems I would recommend to users, the answer is, "it all depends who you are." As is evident from selective weightings of the data in Table 4 mentioned above, cmusic is probably better as a teaching language, there is a better tutorial, and it is easier for novices, experimenters, and other infrequent users to use. For composers who have limited computational resources and require its fast execution, even at the expense of readability, Csound is most likely to be the best choice because of its higher degree of portability (among UNIX systems and personal computers), and its score input languages. I would recommend cmix without reservation for "real programmers" who want to do sound synthesis, DSP, or psychoacoustics, and are more concerned with development time or scalability than the simplicity of small instruments.

Conclusions

Software sound synthesis was the first "technology of computer music," and is still dominant in our field. The range of systems based on current software engineering technology is broad and the differences between systems great. The three systems discussed here—Csound, cmusic and cmix—represent three very different approaches to "mainstream" software sound synthesis. Diverse strengths and weaknesses were discovered for each of the systems, and they were found to provide powerful music processing environments that can be applied to compositional tasks with varying musical

intent in a wide range of studio configurations. As hardware and software technology continue to advance, we can hope for ever more flexible, better integrated, more abstract, and faster software sound synthesis platforms.

The author hopes that this article can serve as an introduction for those new to the technology of software sound synthesis, and that it can grow and be extended by *Computer Music Journal* readers with new example instruments and scores, new SWSS language descriptions, and more comparative benchmark tests.

Acknowledgments

The author is very grateful for the responses of F. Richard Moore, and Paul Lansky to a questionnaire sent to them in preparation for the writing of this article. Both of them, together with Barry Vercoe, Max. V. Mathews, Curtis Roads, D. Gareth Loy, Bill Schottstaedt, James Beauchamp, Adrian Freed, Glendon Diener, and Bruce Pennycook also read and commented on earlier versions of this text. Their input is invaluable. All mistakes and misrepresentations are mine.

References

Atkins, M., et al. 1987. "The Composer's Desktop Project." *Proceedings of the 1987 International Computer Music Conference.* San Francisco: International Computer Music Association. pp. 146-150

Beauchamp, J. 1989. "The Computer Music Project at the University of Illinois at Urbana-Champaign: 1989." *Proceedings of the 1989 International Computer Music Conference.* San Francisco: International Computer Music Association. pp. 21-24

Chowning, J. 1973. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation." *Journal of the Audio Engineering Society* 21(7): 526-534. Reprinted in *Computer Music Journal* 1(2): 46-54 and in C. Roads and J. Strawn, eds.1985. *Foundations of Computer Music.* Cambridge, Massachusetts: MIT Press. pp. 6-29.

Fichera, S. 1991. "Machine Tongues XIII: Real-Time Audio Conversion under a Time-Sharing Operating System." *Computer Music Journal* 15(3): 27-40.

Gross, R. 1982. *CCSS Cylinder-Contiguous Soundfile System*. Reference Manual, Rochester, New York: Electronic Music Studio, Eastman School of Music.

Jaffe, D., and L. Boynton. 1989. "An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal* 13:(2): 48-55. reprinted in S. T. Pope, ed. *The Well-Tempered Object*. Cambridge, Massachusetts: MIT Press.

Lansky, P. 1990. *CMix Release Notes and Manuals*. Department of Music, Princeton University.

Lansky, P. 1991. Private Communication.

Lent, K, R. Pinkston, and P. Silsbee. 1989. "Accelerando: A Real-Time, General-Purpose

Computer Music System." *Computer Music Journal* 13(4): 54-64.

Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* 15(3): 41-49.

Loy, D. G. 1982. "A Sound File System for UNIX" *Proceedings of the 1982 International Computer Music Conference.* San Francisco: International Computer Music Association. pp. 162-171

Mathews, M. V. 1960. "Computer Program to Generate Acoustic Signals." Abstract in *Journal of the Acoustical Society of America* 32: 1493

Mathews, M. V. 1961. "An Acoustical Compiler for Music and Psychological Stimuli." *Bell System Technical Journal* 40:677-694.

Mathews, M. V. 1963. "The Digital Computer as a Musical Instrument." *Science* 142: 553-557.

Mathews, M. V. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: MIT Press.

Moore, F. R. 1985. *Programming in C with a Bit of UNIX*. Englewood Cliffs, New Jersey: Prentice-Hall.

Moore, F. R. 1990. *Elements of Computer Music*. Englewood Cliffs, New Jersey: Prentice-Hall.

Moore, F. R. 1991. Private Communication.

Moore, F. R., D. G. Loy, et al. 1983. *CARL Release Notes, CARL Start-up Kit.* Computer Audio Research Lab, Center for Research in Computing and the Arts. La Jolla: University of California in San Diego.

Morrill, D. 1977. "Trumpet Algorithms for Computer Composition." *Computer Music Journal* 1(1): 46-52. reprinted in C. Roads and J. Strawn, eds. *1985. Foundations of Computer Music.* Cambridge, Massachusetts: MIT Press. pp. 30-44.

Pierce, J. R., M. V. Mathews and J.-C. Risset. 1965. "Further Experiments on the Use of the Computer in Connection with Music." *Gravesaner Blaetter* 27/8: 92-97.

Pope, S. T. 1982. "An Introduction to *msh* the Mshell." *Proceedings of the 1982 International Computer Music Conference.* San Francisco: International Computer Music Association. pp 194-201.

Pope, S. T. 1986. "The Development of an Intelligent Composer's Assistant: Interactive Graphics Tools and Knowledge Representation for Composers." *Proceedings of the 1986 International Computer Music Conference.* San Francisco: International Computer Music Association. pp. 131-144.

Pope, S. T. 1992. "The Interim DynaPiano: An Integrated Tool and Instrument for Composers." *Computer Music Journal* 16(3): 73-91.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13:(2): 23-38. reprinted in S. T. Pope, ed. *The Well-Tempered Object.* Cambridge,

Massachusetts: MIT Press.

Schottstaedt, W. 1992. *Common Lisp Music Documentation*. Reference manual available via Internet ftp from the clm directory on the host machine ccrma-ftp.stanford.edu.

Taube, H. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS." *Computer Music Journal* 15(2): 21-32.

Tenney, J. C. 1963. "Sound Generation by Means of a Digital Computer." *Journal of Music Theory* 7: 25-70.

Vercoe, B. 1991. *CSound Manual and Release Notes*. available via Internet ftp from the music directory on the host machine media-lab.mit.edu.