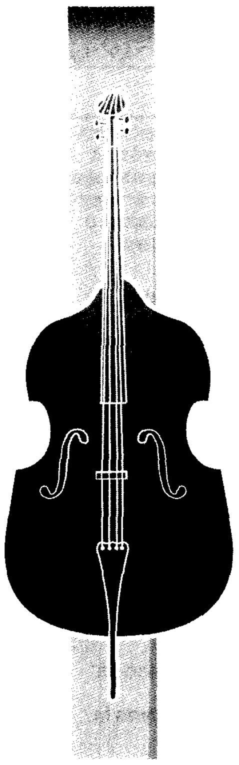


Formula: A Programming Language for Expressive Computer Music

David P. Anderson, University of California at Berkeley
Ron Kuivila, Wesleyan University

Formula, a language for controlling synthesizers, can model the expressiveness of a human performance. It supports algorithmic composition, interactive performance, and programmed interpretation of traditional scores.

Conventional programming systems have limited utility for composing and playing computer music because the languages themselves lack features relating to time and concurrency, and their runtime systems cannot provide the needed levels of output timing accuracy or input response speed.

Formula (an abbreviation for Forth Music Language) is a programming system that addresses these problems. It runs on Atari ST and Macintosh personal computers that control music synthesizers, typically via a MIDI interface.[1] Formula therefore does not specify waveform synthesis algorithms; its abstractions begin at the level of "note" and extend upward.

We based Formula on the Forth language (see the sidebar) and added many new functions and control structures. We designed Formula for the following applications:

- *Programmed score interpretation.* You can use Formula to represent static musical scores, as with Darms[2] or Score.[3] In addition, you can use Formula to express interpretive elements (for example, tempo and volume fluctuation) often not present in scores.
- *Algorithmic composition.* Formula's constructs are executable. Therefore, the features of the underlying programming language (for example, control structures and parameterized procedures) are available at any point. This lets you represent iteration, nesting, and randomness.
- *Interactive systems.* A Formula program plays its output, normally with very accurate timing while it executes. Also, Formula programs respond rapidly to input, so they can define "intelligent instruments" that interact with a performer using input devices such as MIDI instruments or a computer keyboard and mouse.

Formula uses concurrent processes that share a single address space; its runtime system schedules the processes. Formula offers several process types:

- *Note-playing processes* compute sequences of pitches and play these pitches as notes or chords. A separate note-playing process typically represents each "voice" of a piece of music. You can collect note-playing processes into *groups*.

0018-9162/91/0700-0012$01.00 © 1991 IEEE

• *Auxiliary processes* are attached to note-playing processes or groups to supply note parameters such as volume, duration, and articulation. There are several subtypes of auxiliary processes: *shapes* control volume and articulation, *time deformations* generate tempo fluctuations, and *timing sequence generators* produce rhythmic sequences.

• *Input-handling processes* execute when input arrives from a particular device (keyboard, mouse, MIDI). In response to input, they generate output themselves or create note-playing processes to do so.

Formula's factorization into processes provides several advantages. You can easily express concurrent musical voices. You can also exploit the structure in each musical parameter. For example, you can write a repeating rhythm as a loop, even if the corresponding pitches do not repeat.

Factorization into processes also facilitates the specification of musical "expression." For example, the timing of a piece is typically divided into the notated timing, specified by the timing sequence generator process, and the expressive fluctuations in tempo, specified by time deformation processes. These specifications are in separate sections of code, and you can modify them independently. You can factor tempo fluctuation itself into several time deformations and superimpose the deformations to produce the actual note timing.

In this article, we give an overview of the Formula language and describe two representative Formula programs. A complete description of the Formula language is available elsewhere.[4] We have described the process scheduling techniques used in Formula's runtime system in other articles.[5,6]

Basic features of Formula

In this section, we describe Formula's basic facilities for playing notes, creating parallel note-playing processes, and grouping these processes.

Playing notes and chords. Figure 1 shows Formula's note-playing functions, or *words*, in Forth terminology. The basic note-playing word is \$. For example, 60 \$ plays middle C (60 is the MIDI pitch number for middle C). In addition to starting the note (analogous to pressing a piano key), the function stops the note (lifts the key). The default duration is a quarter note, but you can express other time intervals as fractions of whole notes or directly in milliseconds. Also, \$ pauses the calling process; the default is again a quarter note. Thus,

```
60 $ 64 $ 67 $ 72 $
```

plays a C major arpeggio in quarter notes, with each note ending precisely when the next one begins.

```
$    (pitch - - - ; play a note of the given pitch)
m$   (p1 ... pm m - - - ; play m notes in sequence)
$n   (p1 ... pn n - - - ; play n notes as a chord)
```

Figure 1. Formula's note-playing words.

You can name pitches using a, b, ..., g. These words return a pitch number within a current octave, initially the octave beginning at middle C. Prepending a plus or minus sign specifies a pitch in the next higher or lower octave. Appending a plus or minus sign specifies a sharp or flat. For example, +g– specifies the G-flat in the octave above the current octave. *n* oct sets the current oc-

The Forth language

Charles Moore originally developed the Forth[1] language for control applications using microcomputers. Forth is well-suited to computer music for several reasons. Its execution model is simple and can be implemented efficiently on small machines. Moreover, it is extensible and can serve as a basis for defining new language constructs, imposing few requirements on their syntax. Forth also provides a shell-like interpreter, allowing programmers to enter and dynamically test definitions.

Forth functions, called *words*, are invoked using a postfix notation without delimiters. A data stack stores intermediate results. Most words take their arguments and return their results on the stack. Variables and constants are also words (a constant, when executed, leaves its value on the stack; a variable leaves its address). Words are defined as follows:

```
: print-sum (n m - - - ; add two numbers and print their sum)
   +  .
;
```

This example defines a word *print-sum* that adds its two arguments using "+" and prints the result using "." (these words are predefined). The parenthesized comment shows the word's effect on the data stack as *before - - - after*; print-sum expects two numbers (*n* and *m*) and removes them both.

Forth words are stored in *vocabularies*. Forth maintains a search list of vocabularies, and looks up names by scanning vocabularies in this list. The programmer interacts via the interpreter. Normally, the interpreter scans keyboard input, looks up each word in the list of active vocabularies, and executes the corresponding function. If the interpreter does not find a name, it scans the text as a number (if possible) and pushes it on the data stack. When the interpreter is in compile mode (for example, while scanning a word definition), it looks up words and compiles their addresses into the current definition.

Forth provides a mechanism for specifying the compile-time as well as the runtime behavior of a word. Programmers can use this mechanism to define words that implement control structures. For example, the words *do* and *loop* are defined this way and provide Forth's iteration construct:

```
10 0 do (print the numbers 0 .. 9)
   i  .
loop
```

Reference

1. L. Brodie, *Starting Forth*, Prentice Hall, Englewood Cliffs, N.J., 1981.

```
:ap cadence
  g r a r b r 6$
  c e g +c $4
;ap
```

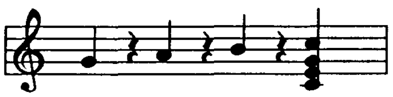


Figure 2. The definition of a note-playing word and the score fragment it plays.

```
:ap scale              (high low - - - ; play a chromatic scale)
  ::ap                 (create a new process to make this asynchronous)
    [ 2 params ]         (pass loop limits to new process)
    do i $ loop
  ;;ap
;ap
```

Figure 3. *[n params]* moves entries from the stack of the parent process to the stack of the child process.

```
:ap trio
  ::gp                 (create a group)
    ::ap soprano ;;ap  (create processes within group)
    ::ap alto ;;ap
    bass
  ;;gp
  ending               (continue here when group is empty)
;ap
```

Figure 4. Creating a group.

tave to *n* (middle C is octave 3). The word r returns a special pitch number (zero) that represents a rest.

:ap and ;ap delimit the definition of a note-playing word. They are like the colon and semicolon, but also add to the search list a vocabulary *ap-defs* containing words relevant to note-playing processes. Figure 2 shows the definition of a note-playing word and the score fragment it plays.

Note-playing processes. The following construct creates a note-playing process:

```
:ap asynch-scale
  ::ap
    c d e f g 5$
  ;;ap
;ap
```

Formula executes the code within the ::ap ... ;;ap construct as a separate process. The parent process continues after ;;ap (it does not wait for the child to finish). The child process exits when it reaches ;;ap. Thus, if you run *asynch-scale* from the Forth interpreter, it will play in the background while the interpreter handles further terminal keyboard input.

This type of construct, called an *embedded process definition*, is used throughout Formula. It lets you embed the code for a process within the procedure that creates the process, rather than in a separate (and perhaps hard-to-find) place.

When you create a process, often you must put initial parameters on its stack. In the example shown in Figure 3, *[2 params]* tells Formula to move the top two entries from the stack of the parent process to the stack of the child.

For many quantities, such as the "current octave" used in pitch-naming, you need a separate copy in each process. Formula provides a mechanism for defining per-process variables (called *pquans*). By default, a pquan name refers to its instance in the currently executing process.

Note-playing process groups. A *group* is a collection of note-playing processes and can contain other groups as elements. You can control (suspend, resume, or kill) a group as a unit. If you attach auxiliary processes to a group, they affect the tempo or volume of all notes generated by processes in that group and its descendants. The ::gp ... ;;gp construct shown in Figure 4 creates a group.

When a process *P* executes ::gp, it temporarily becomes a group. Formula creates a new process *Q*, which executes the code following ::gp and becomes the sole member of the new group. *Q* may create additional processes in the group using ::ap. *Q* exits when it reaches ;;gp. When all the processes in the group have exited, *P* becomes a process again and resumes execution after ;;gp. In the example in Figure 4, *trio* temporarily changes the calling process *P* into a group containing three processes that execute *soprano*, *alto*, and *bass*, respectively. *P* resumes and executes *ending* when these three processes have finished.

Interactive process control. A typical Formula program creates many processes. Usually you need to interact with only a few of them, the *visible processes*, and can ignore the rest. Each visible process has a unique ID (a small integer) and a symbolic name. The following variant of ::ap creates a visible process:

```
:ap trio
  ::ap" trio"
    (trio
  ;;ap
;ap
```

The string following ::ap" (*trio*, in this case) is the symbolic name. Formula assigns the ID. The word *.all* prints a list of all existing visible "objects" (processes and groups), including their names, IDs, and states. Figure 5 lists the words that manipulate visible objects.

Time control structures. As a note-playing process plays notes, it advances through a sequence of time positions (the start and end times of the notes). Formula provides control structures to specify limits on the time consumed by a block of code. The construct

```
maxtime (n - - - )
  ...
maxend
```

specifies that the enclosed code is to consume at most *n* units of time in the process's virtual time system. (We describe virtual time systems in a later section.) If the code exceeds the limit, the system unwinds the stack and transfers control to the statement following *maxend*.

Figure 6 shows code that plays a sequence of random-length notes. The sequence lasts exactly as long as eight whole notes. Other constructs specify a lower time bound on a code block, enforced by repetition or waiting. You can nest these structures; outermost structures have priority.

```
suspend   (ID - - - ; suspend an object)
resume    (ID - - - ; resume a suspended object)
kill-all  (- - - ; kill all mortal objects)
kill      (ID - - - ; kill a specific object)
immortal  (- - - ; make the calling process immune to kill-all)
```

Figure 5. Words to manipulate visible objects.

```
:ap foo
  ::tsg
    begin 1|4 irnd & again   (irnd generates random numbers)
  ;;sg
  8|1  maxtime               (do the following for 8 whole notes)
    begin c $ again          (play C's in a loop)
  maxend
;ap
```

Figure 6. Playing a bounded-length sequence of random-length notes.

```
:ap crescendo
  ::sh1                 (install new volume shape in local context)
    0 100 1|1 oseg      (volume shape process executes this code)
  ;;sh
  c  e  d  f  4$        (play some notes)
;ap
```

Figure 7. Using a shape for a linear volume increase.

Auxiliary processes

The $ words take only pitch arguments. You specify other parameters (such as volume and duration) using per-process variables and *auxiliary processes*. You can also use auxiliary processes to change the tempo of note-playing processes. An auxiliary process can be attached to either a process or a group. If you attach it to a group, it affects all processes in the group and its descendants.

Attaching auxiliary processes. A note-playing process *P* has *local* and *global* contexts. By default, *P*'s local context is itself, and its global context is the top-level group containing it (or, if *P* is top-level, *P* itself).

Every note-playing process and group contains *slots*, each of which can contain an auxiliary process (the set of slots is extensible). Table 1 shows how embedded auxiliary process definitions create new processes in the slots of the calling process or its local or global contexts. The semantics of these constructs are as follows: When a note-playing process reaches the start of the construct (say, ::sh1), the process currently in the *sh1* slot of its local context is killed. Formula creates a new process, executing the embedded code, and installs it in that slot. The constructs also add vocabularies (*sg-defs*, *sh-defs*, and *td-defs*) relevant to the type of process being defined. You can specify that parameters be passed to the new process using *[n params]*. For example, in Figure 7, the volume shape causes the notes to be played with a linear volume increase of 0 to 100 over a whole-note period. (We modified Forth's parser to accept rational-number notation. For example, 3|16 represents a duration of 3/16 — a dotted eighth note, if time units are equated with whole notes.)

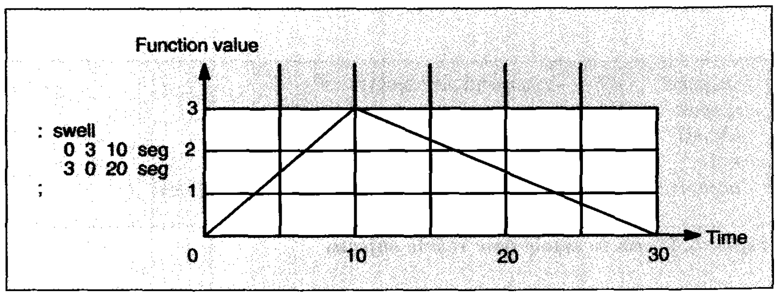
Table 1. The syntax for defining auxillary processes.

Defining Syntax	Location	Purpose	Process Type
::tsg ... ;;sg	Self	Note duration	Sequence generator
::sh1 ... ;;sh	Local context	Volume control	Shape
::sh2 ... ;;sh	Local context	Volume control	Shape
::gsh1 ... ;;sh	Global context	Volume control	Shape
::gsh2 ... ;;sh	Global context	Volume control	Shape
::ash ... ;;sh	Local context	Articulation	Shape
::td1 ... ;;td	Local context	Tempo control	Time deformation
::td2 ... ;;td	Local context	Tempo control	Time deformation
::gtd1 ... ;;td	Global context	Tempo control	Time deformation
::gtd2 ... ;;td	Global context	Tempo control	Time deformation

Procedural concatenation functions. A procedural concatenation function (PCF) is a process that defines a function of time by calling PCF primitives. Each primitive represents a function defined over a time interval. The function defined by the PCF process is the concatenation of the primitive functions. Figure 8 gives an example.

A PCF definition can have parameters, use control structures, and call other PCF definitions. For example, Figure 9 shows the concatenation of *m* instances of *swell*.

Shapes. Shapes are PCFs whose values control volume or articulation. (Shapes might also be used to control timbre, spatial location, or other parameters, but we have not yet implemented these uses.) Figure 10 lists the primitives for shape definitions that Formula provides; others are easy to define. The primitive *oseg* represents a segment ranging



```
: swell
  0 3 10 seg
  3 0 20 seg
;
```

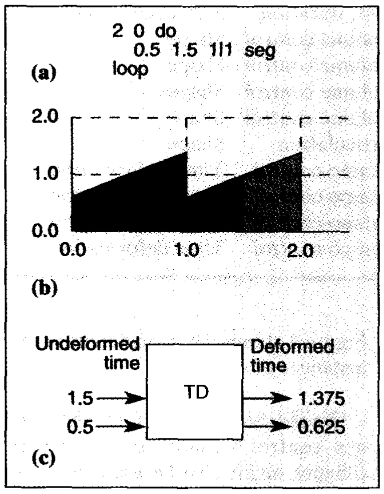
Figure 8. An example of function definition by procedural concatenation. The word *swell* defines the function shown; seg (y1 y2 dt - - - ;) is a procedural concatenation function primitive representing a linear change from *y*1 to *y*2 over an interval *dt*.

```
: many-swells      (m - - - ; m concatenated swells)
  0 do
    swell
  loop
;
```

Figure 9. Concatenation of instances of *swell*.

oseg	(y1 y2 dt - - - ; open linear segment)
cseg	(y1 y2 dt - - - ; closed linear segment)
ocon	(y dt - - - ; open constant segment)
ccon	(y dt - - - ; closed constant segment)

Figure 10. Primitives for shape definitions.



```
2 0 do
  0.5 1.5 1|1 seg
loop
```

Figure 11. A time deformation process executes a procedural concatenation function (a). The resulting tempo function is integrated over its domain (b) to deform a sequence of time intervals (c).

from *y*1 to *y*2 over an interval *dt*. A shape primitive is either open or closed. The shape value at the boundary between two primitives is determined by the first primitive if the first primitive is closed; otherwise, by the second.

The volume of notes played using \$ words is the sum of up to two local volume shapes, up to two global volume shapes, and a per-process variable *\$volume*. The sum must lie in the range [–128, 127]; it is then converted to MIDI's [0, 127] scale. The actual loudness function depends on the synthesizer.

Shapes also control articulation (staccato, legato, and so on). A note's "delay until release" (denoted D_r) can differ from its "delay until next note" (denoted D_n). D_r may be longer (causing note overlap or legato) or shorter than D_n. We call this timing relationship *articulation*. In Formula, an articulation shape that determines D_r, possibly as a function of D_n, controls the articulation for each note-playing process.

The value returned by an articulation shape includes both a number X and an *absolute*, *relative*, or *ratio* mode. An articulation shape generates X using shape primitives such as *oseg*, and can call *absolute*, *relative*, or *ratio* to change its mode. Table 2 shows how the mode and value determine a note's delay until release. Depending on the mode, the numerical value of an articulation shape gives the release time of a note, the release time relative to the start of the next note, or the release time as a multiple of the time until the next note. Hence,

```
relative
.1 1.5 1|1 oseg
```

generates a linear transition from staccato to legato over a whole note. This articulation shape can be applied to any sequence of note durations within that period.

Time deformations. Tempo fluctuations are defined by time deformations (TDs). A TD defines a *tempo function* by procedural concatenation. Formula applies a TD to a time interval X by integrating the tempo function over X (starting from the TD's current position), then advancing the position by the duration of X. For example, if the tempo varies linearly from 1 to 2 over an interval of duration 1, the interval is mapped by the TD to a duration of 1.5. Figure 11 shows a more complex example.

Figure 12 lists the available TD primitives (others are easy to define). The primitives *lpause* and *rpause* insert a "pause" in the tempo function; they map a single time point to a pause of duration *t*. With *lpause*, events (note starts or ends) scheduled for this time occur after the pause; with *rpause*, they occur before the pause.

If you attach two TDs to the same object, their effects are multiplied. If TDs are attached to an object and its parent, they are combined in series: The output of the first is the input of the second. Hence, each object has its own "virtual time system," which is mapped to real time by the clusters of time deformations attached to it and its containing groups.

Timing sequence generators. A timing sequence generator (TSG) is a process that generates a sequence of note durations for a note-playing process. The \$ words get note durations from TSGs. TSG word definitions are delimited by :sg and ;sg, and embedded TSG definitions are delimited by ::sg and ;;sg (both constructs add a vocabulary *sg-defs* to the search list). A TSG returns a sequence element using

```
& (n - - - ; return a sequence
    element n)
```

Table 3 lists commonly occurring rhythmic patterns and their naming conventions. The words in the table are in the *sg-defs* vocabulary because they are called only from TSG definitions. Identical names are used in the *ap-defs* vocabulary for words to install a TSG that generates an infinite sequence of the given durations. This permits convenient notation in note-playing processes:

```
/4 c d e 3$
/8 f g a 3$
```

The first line plays quarter notes; the second, eighth notes.

Two Formula programs

In this section, we illustrate some of Formula's features using two Formula programs. (Readers may order an accompanying compact disk or cassette tape to hear the two selections we describe in this section. See the order form on page 9.)

A programmed interpretation. The first example is an interpretation of the Prelude in F-sharp minor for piano, Opus 28, No. 8, by Frederic Chopin. Figure 13 shows the first four measures of the score. The specification of the score is straightforward because the piece has a repetitive rhythmic structure. Like most of Chopin's work, this piece is generally

seg	(r1 r2 dt - - - ; linear tempo change from r1 to r2 over time dt)
con	(r dt - - - ; constant tempo of r over time dt)
lpause	(t - - - ; "left-justified" pause of duration t)
rpause	(t - - - ; "right-justified" pause of duration t)

Figure 12. Time deformation primitives.

Table 2. Determination of a note's delay until release, D_r.

Mode	Value Determination
Absolute	$D_r = X$ (does not depend on D_n)
Relative	$D_r = \max(0, D_n + X)$
Ratio	$D_r = XD_n$

Table 3. Notation of commonly occurring rhythmic patterns.

Name	Definition	Length of Note
/1	1|1 &	Whole
/4.	3|8 &	Dotted quarter
/4..	7|16 &	Double dotted quarter
2/8	/8 /8	Two eighths
/4.8	/4. /8	Dotted quarter and eighth
/2-3	2|3 & 2|3 & 2|3 &	Triplet half
/8+	/8 /16 /16	Eighth and two 16ths

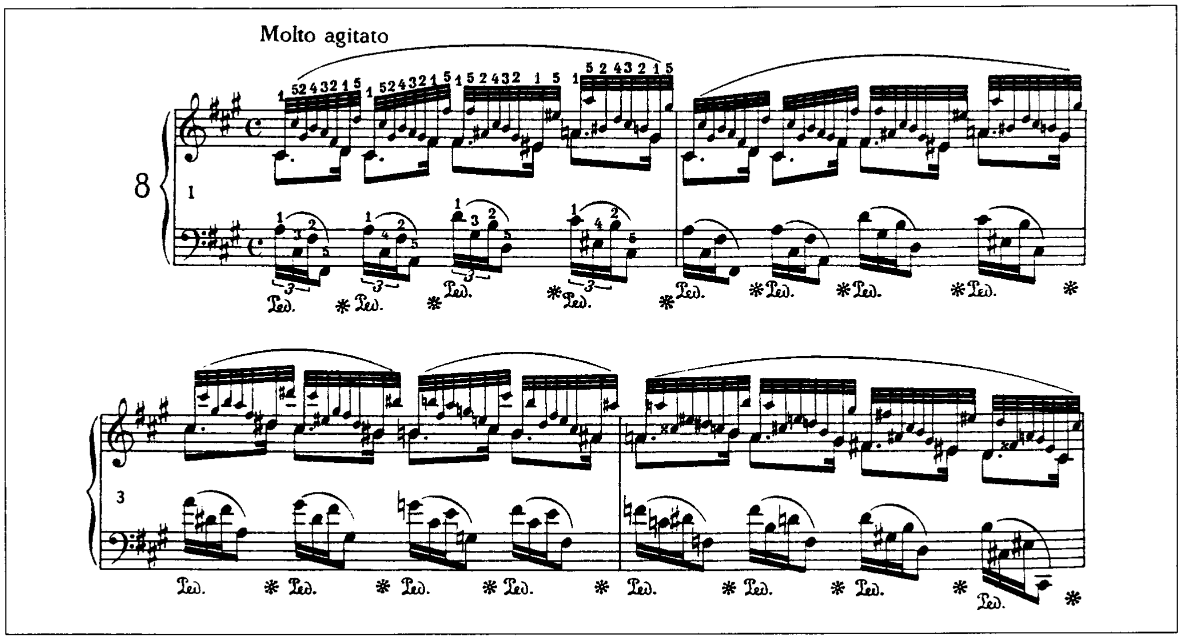


Figure 13. The beginning of the score for Prelude No. 8 by Chopin.

played with significant tempo and volume fluctuation.

Figure 14 shows the process structure of the program. A top-level group contains note-playing processes for the left- and right-hand parts, each of which has an articulation shape, a volume shape, and a timing sequence generator. Two time deformations are attached to the group, affecting both note-playing processes uniformly. (In some interpretive styles, the timing of the left- and right-hand parts is deformed independently. This is possible in Formula, but it requires writing time deformations that synchronize — have equal integrals — at the desired points.)

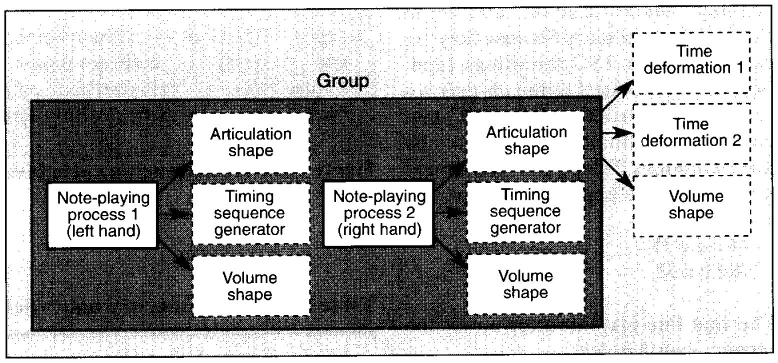

Group
Note-playing process 1 (left hand)
Articulation shape
Timing sequence generator
Volume shape
Note-playing process 2 (right hand)
Articulation shape
Timing sequence generator
Volume shape
Time deformation 1
Time deformation 2
Volume shape

Figure 14. The process structure of the Formula program for Chopin's Prelude No. 8. Lined boxes show note-playing processes; dashed boxes show auxiliary processes.

```
:ap  rhb        (6 pitches - - - ; do 1 beat of right hand)
  dup $ 12 + $ $ $ $ $ dup $ 12 + $
;ap

:ap  rh         (right-hand process)
  ::sh1         (volume shape within each beat)
    begin
      0 1|32 ocon -20 1|32 ocon
      -40 1|16 ocon
      -20 1|32 ocon -40 1|32 ocon
      -10 1|32 ocon -20 1|32 ocon
    again
  ;;sh
  ::ash         (articulation control)
    absolute
    begin
      6(32 1|32 ocon 5(32 1|32 ocon
      1(32 1|8 ocon
      1(16 1|32 ocon 1(32 1|32 ocon
    again
  ;;sh
  3 oct /32
(measure 1)
  2 0 do
    d f+ a b g+ c+ rhb f+ g+ a b g+ c+ rhb
    e+ g+ b +c+ a+ f+ rhb g+ b +c+ +d b+ a rhb
  loop
(measure 3)
  +d+ +f+ +a +b +g+ +c+ rhb b+ +d+ +f+ +g+ +e+ +c+ rhb
  +c+ +e +g +a +f+ b rhb a+ +c+ +e +f+ +d+ b rhb
  b +c +d+ +e+ +d a rhb g+ b +d +e +c+ a rhb
  e+ g+ b +c+ a+ f+ rhb c+ e+ g+ a g d rhb
(measure 5)

(160 lines omitted)

;ap

:sh  1bar           (swell over 4 beats, peak on 4)
  0 70 3|4 cseg
  70 0 1|4 oseg
;sh

:sh  halfbar        (swell over 2 beats, peak on 2)
  0 40 1|4 oseg
  40 0 1|4 oseg
;sh

:sh global-volume         (long-term volume control)
(1) 2 0 do 1bar 1bar halfbar halfbar 1bar loop
(9) 0 127 6|1 oseg 127 2|1 ocon 0 2|1 ocon
(19) 1bar 0 127 2|1 oseg 127 1|1 ocon
(23) 127 0 4|1 oseg
(27) 0 1|1 ocon 1bar
(29) -40 1|1 ocon -40 0 3|4 oseg 0 -40 1|4 oseg
(31) -40 100 1|1 oseg 100 -40 1|1 oseg
(33) 30 2|1 ocon ;sh

:td tri          (n m k - - - ; triangular TD over
                  4 beats w/ peak on 3)
  >r dup >r 1|2 seg
  r> r> 1|2 seg
  1(64 lpause         (with a little pause at the end)
;td

:td tri/2        (n m k - - - ; triangle over 2 beats)
  >r dup >r 1|4 seg
  r> r> 1|4 seg
  1(64 lpause
;td

:td med 310 270 380 tri ;td           (some common
                                       rubato patterns)
:td med/2 310 270 380 tri/2 ;td
:td slow 320 280 480 tri ;td
:td quick 310 250 310 tri ;td
```

Figure 15. Programmed interpretation of Chopin's Prelude No. 8.

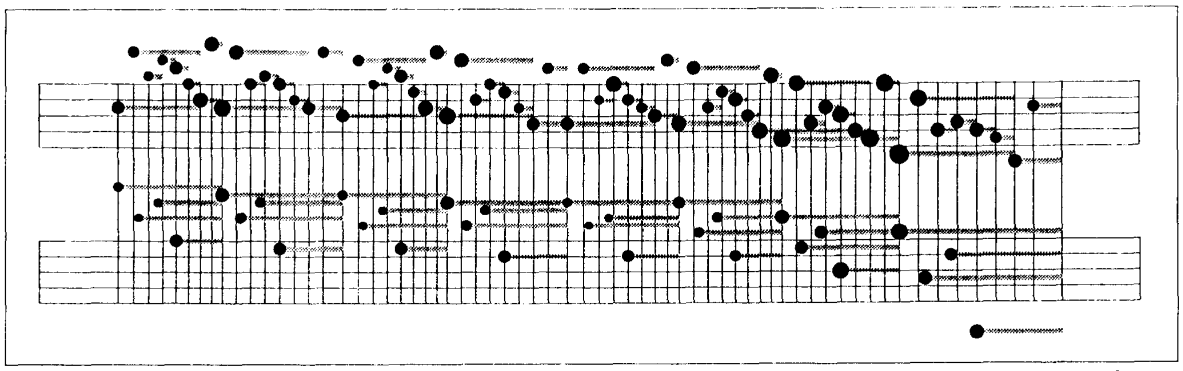


Figure 16. A graphic representation of measures 3 and 4 of the Formula program output for Chopin's Prelude No. 8. Each disk represents a note. Horizontal position is time, and the size of the disk represents the note's volume. The gray bar following each disk represents the time for which the note sounds. Vertical lines are ruled at each 32nd note.

```
232
233 :td phrasing-tempo          (long term tempo control)
234 (1) med med med/2 med/2 slow 1(64 lpause
235 (5) med med med/2 med/2 med
236 (9) quick quick med/2 med/2 med 1(64 lpause
237 (13) 300 260 2|1 seg 1(32 lpause
238 (15) quick quick 3(32 lpause
239 (17) med 320 300 600 tri 1(64 lpause
240 (19) med med
241 (21) med/2 med/2 300 400 1|1 seg 1(64 lpause
242 (23) 400 300 380 tri 380 320 340 tri
243 (25) 340 280 1|1 seg 1(64 lpause med
244 (27) med slow med 320 280 500 tri 1(32 lpause
245 (31) 300 260 1|1 seg 1(64 lpause slow
246 (33) 1(16 lpause 350 500 5|4 seg
247 ;td
248
249 :ap (prelude            (main program, synchronous version)
250   ::gp
251     96 beats-per-minute
252     ::gsh1 global-volume ;;sh
253     ::gtd1 phrasing-tempo ;;td
254     ::gtd2    (rubato at 1 beat level)
255       begin 270 200 1|4 seg again
256     ;;td
257     ::ap $left piano lh ;;ap
258     $right piano rh
259     $center
260     ending
261   ;;gp
262 ;ap
263
264 :ap prelude             (main program, asynchronous version)
265   ::ap" prelude"
266     (prelude
267   ;;ap
268 ;ap
```

In the Formula program shown in Figure 15, lines 1 through 3 define a word *rhb* to play one beat of the right hand. Because each beat contains two notes repeated at the octave, we need only six pitches to specify the eight notes. Lines 6 through 13 define the "micro-volume" shape for the right-hand part, while lines 14 through 21 define the articulation shape. The timing sequence generator is specified by /32 in line 22. Lines 24 through 90 specify the pitches for the right hand. Lines 196 through 214 define a global volume shape applied to both the left and right hands. We separately defined two commonly occurring patterns, swells of half- and whole-measure duration, with *1bar* and *halfbar* and used them repeatedly in global-volume.

Lines 216 through 247 define a global time deformation for long-term (phrase-level) control. This time deformation uses a set of words (*tri*, *med*, *quick*, and so on) for commonly occurring phrasing. Each phrase ends with a short pause. A second time deformation is defined in-line (line 255) for beat-level control: Within each beat, the tempo starts out slow and speeds up. Finally, lines 249 through 268 define the main program.

Figure 16 graphically represents part of the output of this Formula program and shows the effects of the auxiliary processes described above. The volume shapes accent the strong positions within each beat and provide a swell toward the high point of each phrase. The time deformations (manifested by the irregularity of the vertical lines) linger on the strong beats, rush toward the top of each phrase, and pause between phrases. The articulation shape in the right hand sustains the melody notes but not the passing tones.

An algorithmic composition. Figure 17 shows the source code for "Tuf-Stuf," an algorithmic composition programmed by Ron Kuivila. The composition demonstrates the power of Formula's multiple-process model: A short program generates a lengthy and complex piece. "Tuf-Stuf" consists of eight note-playing processes executing the same algorithm (*tuf*) but with different parameters (lines 46-53). The algorithm steps through a cyclical pitch group with a given increment (lines 34-38). For each process, the size of the pitch group varies over time. Each process is subjected to a triangle wave volume shape (lines 13-18). The processes start and end at different times, and use different instrument sounds and spatial locations.

The Formula language can produce expressive computer-generated music. It is related to music languages such as Moxie,[7] PLA,[8] Formes,[9] and Darms,[2] but it has a unique combination of attributes:

• *Programmability.* Almost all Formula features are executable statements. Thus, the power of the underlying programming language (for example, its control structures and parameterized procedures) is available throughout Formula.

• *Real-time interaction.* Formula is based on a real-time process scheduler.[6] Unless a program's computation time exceeds its performance time over long periods, the system executes output actions with highly accurate timing (typically within 5 milliseconds). Input-handling processes can very rapidly create new note-playing processes, and their output begins within a few milliseconds.

• *Lightweight processes.* Formula's unit of structure is the process, a thread of execution with its own stack of call frames and local variables. Processes can advance in time within arbitrary nested function calls. This is a natural and powerful way to maintain computation state over time. In contrast, languages such as Moxie[7] require the computation for a given logical instant to run to completion.

• *Separation of score and interpretation.* Formula makes it simple to separate a score (embodied in note-playing and timing-sequence-generator processes) from its interpretation (represented by shapes and time deformations).

Formula has many other music-related features beyond those we discussed in this article. For example, its synthesizer manager provides a uniform interface for synthesizer output and allows concurrent processes to share synthesizers coherently.[10] Formula provides a facility for defining and using nonstandard tuning systems and includes several predefined tuning systems, such as "stretched" equal temperament, just-intoned scales, and a Javanese Gamelan scale.

Forth's simplicity and its highly extensible nature made it well suited for developing Formula. On the other hand, Forth syntax often leads to hard-to-understand code, and it lacks some structuring features (for example, type declarations) useful for large-scale software development. For these reasons, we are currently reimplementing Formula using C++ as the base language.[11] ■

Acknowledgments

We are deeply indebted to Mitch Bradley for developing Forthmacs, the Forth system on which we based Formula. We also thank the people who have contributed to Formula, including Bruce Holmer, Henricus Holtman, George Homsy, Jim Horton, Marc Sabatella, Rob Vaterlaus, and Erling Wold.

```
 1  pquan cycle-size          (per-process variables)
 2  pquan base-pitch
 3  pquan inc
 4
 5  :ap tuf         (t0 t1 base-pitch cycle-size inc priority dur shape-dur - - - ;)
 6  (t0 and t1 are the start and end times in measures)
 7  (base-pitch is the starting pitch)
 8  (cycle-size determines the size of cycles)
 9  (inc is the increment in cycles)
10  (priority is the note priority for synth manager)
11  (dur is the reciprocal of note duration)
12  (shape-dur is the volume shape period in measures)
13      ::sh1 [ 1 params ]        (sawtooth-wave volume shape)
14         >r
15         begin
16            p ff r cseg ff p r cseg
17         again
18      ;;sh
19      ::tsg [ 1 params ]        (note duration is 1/n)
20         begin
21            1 over r>i &
22         again
23      ;;sg
24      ::ash                     (articulation is detached)
25         ratio
26         begin 0.5 1|1 ocon again
27      ;;sh
28      to inc
29      to cycle-size
30      to base-pitch
31      maxtime
```

Figure 17. "Tuf-Stuf," an algorithmic composition by Ron Kuivila.

References

1. *Musical Instrument Digital Interface Specification 1.0*, Int'l MIDI Assoc., North Hollywood, Calif., 1983.

2. R.F. Erickson, "The Darms Project: A Status Report," *Computers and the Humanities* Vol. 9, No. 6, June 1975, pp. 291-298.

3. L.C. Smith, "Score, A Musician's Approach to Computer Music," *J. Audio Eng. Soc.*, Vol. 20, No. 1, Jan./Feb. 1972, pp. 7-14.

4. D.P. Anderson and R.J. Kuivila, "Formula Version 3.4 Reference Manual," Tech. Report 91/630, Computer Science Division, Univ. of California at Berkeley, May 1991.

5. D.P. Anderson and R.J. Kuivila, "Continuous Abstractions for Discrete Event Languages," *Computer Music J.*, Vol. 13, No. 3, Fall 1989, pp. 11-23.

6. D.P. Anderson and R.J. Kuivila, "A System for Computer Music Performance," *ACM Trans. Computer Systems*, Vol. 8, No. 1, Feb. 1990, pp. 56-82.

7. D. Collinge, "Moxie: A Language for Computer Music Performance," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1984, pp. 217-220.

8. B. Schottstaedt, "PLA: A Composer's Idea of a Language," *Computer Music J.*, Vol. 7, No. 1, Winter 1983, pp. 11-20.

9. X. Rodet and P. Cointe, "Formes: Composition and Scheduling of Processes," *Computer Music J.*, Vol. 8, No. 3, Fall 1984, 32-50.

10. D.P. Anderson, "Synthesizer Management Based on Note Priorities," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1987, pp. 230-237.

11. D.P. Anderson and J. Bilmes, "Concurrent Real-Time Music in C++," *Proc. Usenix C++ Workshop*, Berkeley, Calif., 1991, pp. 147-161.

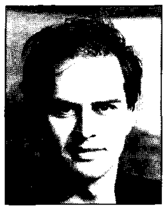
```
        time-advance
        base-pitch
        127 1 do
           i cycle-size 2/ * 0 do
              inc + j mod base-pitch + 127 and dup $
           loop
        loop
        drop
     maxend
  ;ap

  :ap  (tuf-stuf
     ::gp
        280 beats-per-minute
        ::ap 50 to $location nasty-bass 0 180|1 30 2 146 2 42|1 tuf ;;ap
        ::ap 20 to $location piano 10|1 180|1 50 4 73 4 84|1 tuf ;;ap
        ::ap 80 to $location piano 25|1 50|1 66 2 146 3 64|1 tuf ;;ap
        ::ap 80 to $location piano 50|1 180|1 66 2 146 3 64|1 tuf ;;ap
        ::ap 100 to $location vibes 50|1 180|1 70 4 73 6 128|1 tuf ;;ap
        ::ap 0 to $location xylophone 50|1 100|1 76 2 146 6 128|1 tuf ;;ap
        ::ap 0 to $location xylophone 100|1 180|1 76 2 146 3 128|1 tuf ;;ap
        127 to $location electric-piano 100|1 180|1 69 3 18 9 256|1 tuf
     ;;gp
  ;ap

  :ap tuf-stuf
     ::ap" tuf-stuf"
        (tuf-stuf
     ;;ap
  ;ap
```

David P. Anderson is an assistant professor in the Computer Science Division at the University of California at Berkeley. In addition to his work in computer music, he has done research in distributed operating systems, software support for digital audio and video, distributed programming, computer graphics, and protocol specification.

Anderson received his BA in mathematics from Wesleyan University, and his MA in mathematics and MS and PhD in computer science, all from the University of Wisconsin-Madison.



Ron Kuivila teaches in the Music Department at Wesleyan University. He composes music and designs sound installations to highlight the unusual electronic instruments he designs. He pioneered the musical uses of ultrasound, sound sampling in live performance, speech synthesis, and high-voltage phenomena. He has performed and exhibited throughout the US and Europe.

Kuivila received a BA in music from Wesleyan University and an MFA from Mills College.

Readers can write to Anderson at the Computer Science Division, Electrical Engineering and Computer Science Dept., University of California, Berkeley, CA 94720; or Kuivila at the Music Department, Wesleyan University, Middletown, CT 06457.