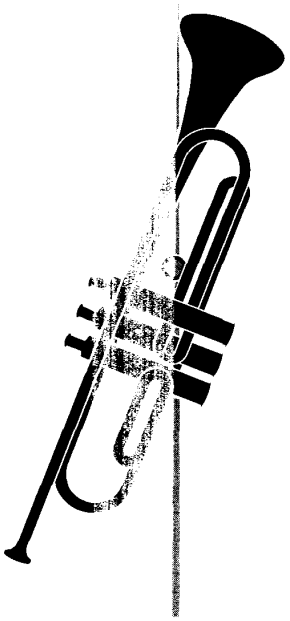

Fugue: A Functional Language for Sound Synthesis

Roger B. Dannenberg, Carnegie Mellon University
Christopher Lee Fraley, Microsoft Corporation
Peter Velikonja, West Virginia University

Fugue provides functions to create and manipulate sounds as abstract, immutable objects. The interactive language supports behavioral abstraction, so composers can manage complex musical structures.

Traditional software synthesis systems — which include Music V,[1] cmusic,[2] and Csound[3] — are based on the same principles. Composers use a score language to describe a list of notes or sounds to be synthesized. Each note specifies a starting time, a duration, an instrument name, and a list of parameters specific to the instrument (such as pitch, loudness, and articulation). A separate orchestra language defines a set of instruments, each of which specifies a particular signal processing algorithm. An instrument is defined as a set of interconnected signal processing steps known as unit generators. Typical unit generators are oscillators, filters, adders, and multipliers.

Each note in the score language invokes an instance (essentially a copy) of an instrument in the orchestra. This instrument instance computes sound for the duration of the note according to the parameters of the note statement. The synthesis system adds the resulting sound to that of the other notes in the score and writes them all to a disk file. After the computation completes, the system can read the synthesized music from the disk in real time and convert it into analog form for listening.

This approach has been used without much change for over two decades. It offers excellent and robust ideas, but it also has some weaknesses. An obvious problem is the separation of the score and orchestra languages.

In this article, we present Fugue. Unlike the traditional approach, which requires separate languages for different tasks, Fugue lets composers express signal processing algorithms for sound synthesis, musical scores, and higher level musical procedures all in one language.

Although it extends the traditional sound synthesis approach[2] with concepts borrowed from functional programming,[4] Fugue retains the advantages of the Music V class of systems. One advantage of Music V is that composers can specify the starting time and duration of each note in the score. These temporal attributes

0018-9162/91/0700-0036$01.00 © 1991 IEEE

are implicit parameters to each instrument instance that tell the system when to start and how many samples to compute. Fugue supports an extension of this technique.

Another important contribution of Music V is the notion of combining unit generators into a signal flow graph. Fugue implements this feature efficiently.

Programs as scores

Historically, musical scores have been static data structures created by composers. Traditional scores do not express computation beyond a few simple abbreviations to indicate repeats or alternate endings. There is a good reason for this. Traditional scores are data intensive and contain much detailed information because composers want to express the results of their creativity, rather than show the process of creation.

Consequently, lists of notes and their attributes have been the standard form of machine-readable scores for many years. As data structures, note lists can be transformed in time, in pitch, or along any other dimension defined by note attributes. It is generally easier to generate and manipulate data structures than programs. For example, making all the notes in a section louder is easy if the notes are represented as data. But note lists can suffer from the fact that they are not programs. In particular, there is a schism between the "score" and the "orchestra": The score controls while the orchestra executes; the functions are not integrated.

A common way to address this dilemma is to use programs to compute note lists. This presents at least two problems. Composers must deal with yet another language (as if two were not enough), and the ultimate result does not close the gap between the score and orchestra. For example, note-generating procedures cannot use the results of signal processing functions in the orchestra, and instruments in the orchestra cannot call on the note-generating capabilities of the score.

Fugue offers a new approach. Fugue scores are actually program expressions that, when evaluated, return audio or control signals. Fugue also defines instruments in terms of expressions that denote audio signals. Thus, composers can use one language for both instruments and scores.

With this unification, composers can express scores and instruments more flexibly. In traditional synthesis systems, it is difficult to alter a phrase of many notes by a single volume envelope, because this requires a hierarchical nesting of notes within the volume envelope. In Fugue, however, scores and signals are all nested expressions, making hierarchical structures very natural.

Composers can also use Fugue for signal analysis to determine aspects of a score. Signal analysis is a common practice in computer music, but signal analysis programs are rarely integrated into traditional sound synthesis languages.

Perhaps Fugue's most valuable feature is that it encourages composers to develop personal musical vocabularies, unencumbered by a language-specified model of how music or music computation should be structured. Of course, any language, including Fugue, is bound to influence how composers approach programming or composition. However, Fugue supports the Music V model of computation as a subset, so we can at least claim an improvement in flexibility and generality.

Fugue allows the composer to treat simple instruments and sounds as building blocks for more complex sound events. This development process is supported by its abstraction capabilities and an interactive language interpreter.

Behavioral abstraction

An important feature of Music V is that note starting times and durations determine when and how long the system instantiates an instrument. In addition to unifying the score and orchestra, Fugue also supports this approach. Otherwise, temporal aspects of compositions might be very difficult to express.

Starting times and durations in Music V provide a special kind of abstraction: Instruments define a class of behaviors that can have any starting time or duration. Abstraction is important because it is not always obvious what to do to make a note longer. For example, a violinist typically lengthens a note by drawing the bow across the string for a longer period of time, but if the note is a tremolo, with rapid back-and-forth bowing, the violinist adds more bow strokes to extend the duration. Clearly, the notion of stretching is abstract, and the software instrument designer must control its implementation. The user of the instrument, however, need not be aware of the implementation details.

To support nested expressions in Fugue, we viewed a starting time or duration as a transformation rather than as an absolute parameter. We extended the concept to allow transformations of articulation, loudness, and pitch. Composers can make additional qualities transformable and extend the system with new transformation operators.

Programmers or composers define *behaviors* that describe how to generate a sound within a *transformation context*. A context in Fugue reflects the cumulative effect of nested transformations on environmental parameters such as current time, stretch, transposition, and overall loudness. Behaviors can be hierarchical compositions of other behaviors. Once defined, a behavior can have many *instances*, each of which can be evaluated in a different context and/or with different parameters. In this way, composers can define concepts such as "drum roll" or "glissando" once and apply them in many different contexts.

A behavior realized according to a context is called a *behavioral abstraction*.[5] We present a few examples to show how Fugue uses behavioral abstractions. In a sequence of three sounds

```
(seq (tremolo A3) (cue wind) (osc Bf3))
```

cue is a behavior that simply plays a sound at a given time, and tremolo and osc play pitches (A and B-flat below middle C). Osc and cue are built-in behaviors, while tremolo is defined by the composer in terms of built-in behaviors. Wind is a sound, perhaps loaded from a sound file.

If we want to hear the same sequence half as loud and with the wind sound delayed by 2 seconds, we write

```
(loud 0.5
  (seq (tremolo A3) (at 2.0 (cue wind))
    (osc Bf3)))
```

Suppose we wish to change the pitched sounds of the sequence. We write

```
(transpose 3 (seq (tremolo A3)
  (cue wind) (osc Bf3)))
```

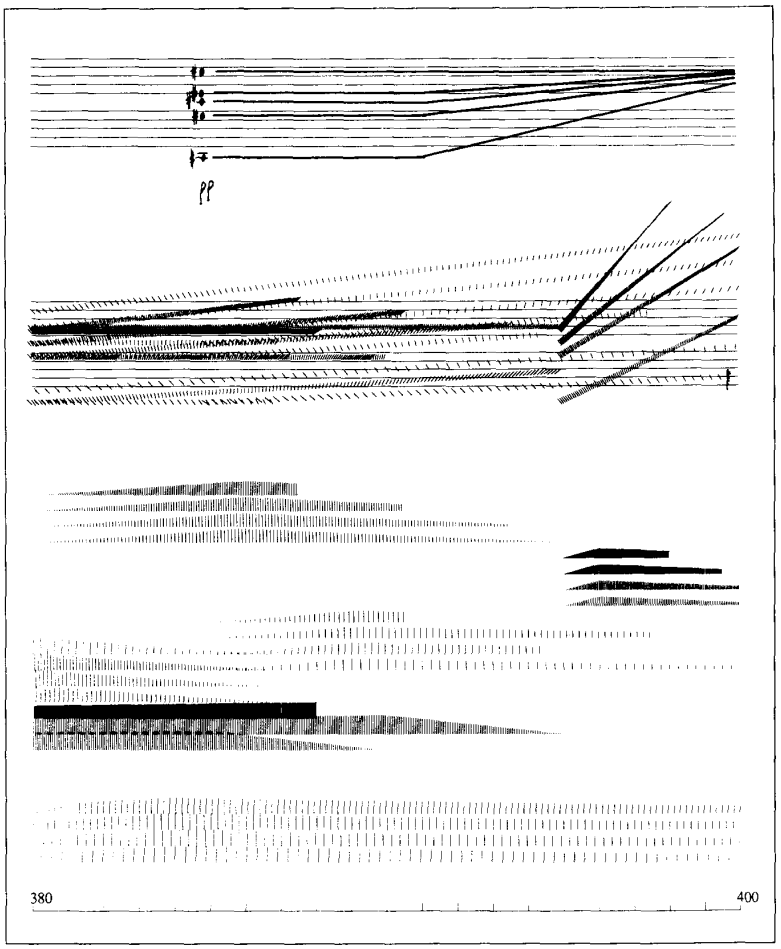

Figure 1. Page 25 from the score of *Spomin*. To generate the score, we replaced Fugue's sound generation modules with modules that output graphics commands. The modules ensure that the score accurately reflects the sound. The final score shown here includes manually added music symbols.

This says to transpose the sequence up by three semitones. However, the cue abstraction overrides and prevents the transposition of wind because cue is intended for use with unpitched sounds. Hence, only the tremolo and osc behaviors are affected. We could replace cue with a behavior that would allow the wind sound to transpose.

In general, we defined transformations and built-in behaviors to work like traditional unit generators. As a result, most instrument definitions in Fugue support the standard transformations implicitly. But composers can always customize the default behavior as needed.

An example

The composition *Spomin* (an unpublished score by P. Velikonja) uses thousands of transformations of a single human vocal utterance (see the box at right). The composer used Fugue sound-processing primitives to manipulate the source sound to varying degrees, so some sounds are clearly vocal while other more highly processed sounds bear little relation to their vocal source. Using Fugue, the composer extracted slices of the source, in some cases down to the level of an individual period. Once extracted, a period can be cycled to form a sustained tone (as in the technique used by sampling synthesizers). Quickly swapping periods during this cycling produces a tone with a time-varying spectrum, a technique used extensively in the latter half of the piece. In the first half, the composer used tones generated from extracted periods to create chords, glissandi, and chorus effects.

Spomin illustrates the advantage of using an integrated language for expressing score information as well as signal processing. Composers can delegate signal processing routines to low-level functions and then work more abstractly using high-level functions. Moreover, Fugue modules, modified to output printing information rather than digital samples, can produce the graphical portion of a score. Figure 1 gives an example.

The code in Figure 2 shows the layering of several levels of abstraction. In this example a short slice is cut from the source sound and cycled to form a grain. Grains of sound form pebbles, which are strung on a necklace. The second band from the top in Figure 1 shows necklaces modified at the grain level to rise in pitch over time. Each grain is represented by a short line, angled to indicate where (in the source) a slice was extracted. The third, fourth, and fifth bands show timing and amplitude information. The top band shows glissandi of tones built from extracted periods.

Without Fugue's abstraction capabilities and computational support, the top level of the score would consist of many thousands of notes, each corresponding to a tiny grain of sound. Such a score is technically feasible but impractical to construct or edit by hand.

Because Fugue allows composers to combine components to form complex structures, they can control large sound events with a few commands. The interactive environment provides rapid

Audio examples

Readers may order an accompanying CD or cassette to hear the selections discussed in this article by using the order form on page 9.

feedback, so composers have greater control over sound materials. Fugue's supportive framework helps composers explore new musical forms and compositional methods. Composers can define a musical syntax so that they can compose music as a process rather than as a simple series of notes. Some might object that Lisp is too great an obstacle for noncomputer scientists, so it is worth mentioning that *Spomin* is the first full piece generated using Fugue and was the composer's first exposure to Lisp.

Implementation

We implemented Fugue in a combination of C and XLisp to run on Unix workstations. (Written by David Betz, XLisp is an interpreter which is in turn implemented in C.) We used XLisp because it is fairly easy to extend with new data types, and it is also easy to interface with C programs for signal processing. Lisp provides convenient and powerful interaction, and C efficiently implements low-level functionality. It might be possible to use a more efficient compiled Lisp. However, most of the computation time is taken by signal processing primitives, so the Lisp interpretation represents only a small overhead.

We implemented Fugue's transformation context in Lisp. Operators such as Transpose are macros that bind an element of the context (*Transpose* in this case) and then evaluate the embedded expression. The binding is restored upon exit from the transformation. Composers can introduce new synthesis techniques into Fugue by combining existing behaviors or by writing new sound synthesis algorithms in C and calling them from Lisp.

Our implementation includes multiple sample rates and lazy evaluation to increase time and memory efficiency. Multiple sample rates allow for "control" signals at a low sample rate, as in the Music-11 system and Csound,[3] reducing time and memory requirements. When the system must manipulate two sounds with different sample rates, it uses linear interpolation by default to resample the lower sample rate signal to the higher sample rate. Composers can explicitly specify other types of interpolation. Because there is no distinction between control and audio signals, composers can use filters to modify spectra or to smooth envelopes, and use multiplication uniformly for gain control, amplitude envelopes, or audio-rate amplitude modulation.

```
(defun osc-slice                          ;a slice from time start
    (sound sndpitch start end pitch dur)  ;to end is cut from
  (s-mult (pwl (* dur .5) 1.0 dur)        ;sound, oscillated at
          (osc pitch dur 0                ;pitch with an
           (extract start end sound)      ;envelope of length dur.
           sndpitch)))

(defun bb-grain (period pitch dur)        ;a grain consists of one such
  (osc-slice bb 49.0                      ;oscillated slice. A grain
             (* period 0.008)             ;is selected by the number
             (+ 0.008 (* period 0.008))   ;period. The typical
             pitch                        ;length of a slice is 0.008
             dur))                        ;seconds.

(defun bb-pebble                          ;a pebble is formed from
    (ngrains period pitch dur)            ;several (ngrains) grains.
  (seqrep (g ngrains)                     ;In this example the grains
    (bb-grain (+ period g) pitch dur)))   ;between period and
                                          ;period + ngrains form
                                          ;the pebble.

(defun bb-necklace                        ;a necklace is made from a
    (npebbles ngrains pitch dur)          ;number (npebbles) of
  (seqrep (p npebbles)                    ;pebbles.
    (bb-pebble ngrains p pitch dur)))

(sf-od                                    ;a necklace made from 8
  (s-mult                                 ;pebbles, each containing 90
    (pwl 25.0 1.0 28.0 .5 35.0)           ;grains of 0.05 second
    (bb-necklace 8    ; npebbles          ;duration pitched at e2
                 90   ; ngrains           ;(tenth above middle c), has
                 76.0 ; pitch             ;a piecewise-linear envelope
                 .05  ; dur               ;applied to it. The sound is
  )))                                     ;written (by sf-od) to an
                                          ;optical disk.
```

Figure 2. An example Fugue program taken from *Spomin*, showing multiple levels of abstraction. Any level can be invoked interactively for testing, or from within a higher level expression serving as a score.

We implemented sounds in Fugue as an extension to Lisp. Sounds are immutable values, meaning that once a composer creates a sound, it cannot be altered. Therefore, the implementation cannot add several sounds directly into a single buffer. Instead, each addition of two sounds produces a new sound. Using lazy evaluation,[6] we avoid the inefficiency of immutable values. When composers perform additions (and many other operations), our implementation merely builds a small data structure describing the desired operation. Fugue doesn't actually compute any samples until absolutely necessary, and when it does, it can combine operations. For example, Fugue commonly allocates only one array to hold the result of many additions. This technique avoids many needless copy operations and is completely hidden from the composer.

Example of lazy evaluation

To demonstrate how Fugue's implementation of lazy evaluation works, we show what memory structures look like

```
(setf Mysound (sfload "mysound"))
(setf Demo (scale 2.0 (seq (cue Mysound)
                           (cue Mysound))))
(play Demo)
```

Figure 3. Sequence of operations for the lazy evaluation example.

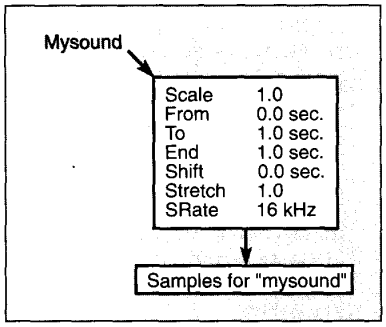

Figure 4. Memory structure of Mysound after the first operation in Figure 3, assuming the duration of Mysound is 1 second. The system loaded the samples from a file.

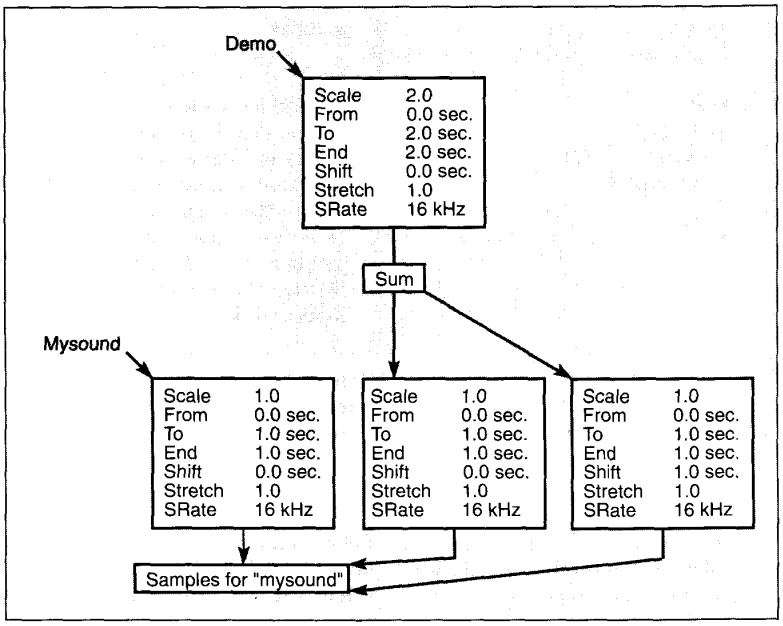

Figure 5. Memory structure of Demo after the second operation in Figure 3. The transformations and summation are reflected in the data structure, so the system doesn't need to compute new sound samples.

at each step of a sequence of operations. Figure 3 shows the operations. The first line sets the variable Mysound to a sound stored in the file "mysound." Figure 4 shows the resulting configuration. The second line evaluates a score consisting of two copies of Mysound in sequence. Because only time-shifting and addition are involved, essentially no computation takes place. Figure 5 shows the resulting configuration. The system has performed no addition; instead, Fugue represents the sum as a data structure. Finally, the third line forces the system to produce samples for Demo. It replaces the representation for Demo with one in which the actual samples have been computed and storage for samples has been allocated, as shown in Figure 6. This last step is the only time the system allocates new storage for sample data and actually computes a new sound sample.

Imagine a typical computation of the form

```
for i := 1 to 100 do
        myPiece := myPiece +
          MakeNote(i);
```

In Fugue, a roughly equivalent program would be

```
(seqrep (i 100) (make-note i))
```

where seqrep is a control construct that concatenates some number of instances of a behavior — in this case 100 copies of make-note.

Typically, MakeNote(i) generates a relatively short signal to be added to a much longer myPiece. Without lazy evaluation, each addition requires the system to

(1) allocate memory at least the size of myPiece to hold the sum of the two signals,
(2) copy myPiece into the new memory area, and
(3) add the result of MakeNote to form the new signal.

Allocating memory and copying signals make this computation costly.

With lazy evaluation, each "lazy" addition simply adds another level to a tree of sum nodes like the one in Figure 5. To produce the final result, the system traverses the tree to determine the size of the result, allocates memory, and adds the leaves of the summation tree. This technique eliminates memory allocation and signal copying to form intermediate results.

We considered but did not implement another approach for efficient execution. In the alternate approach, the composer explicitly allocates a buffer to hold the final sum of the MakeNote signals, and the system allows the buffer to be modified by an add-signal operation. This approach violates the principle that signals are immutable, and it places more storage management burden on the composer. Lazy evaluation, on the other hand, allows Fugue to exhibit clean and simple semantics without loss of efficiency.

Future directions

We need to extend Fugue with more sound functions as in Moore's cmusic (distributed by the University of California at San Diego), Vercoe's Csound (distributed by the MIT Media Lab), Next's Sound Kit,[7] and Lansky's Cmix (distributed by the Princeton University Music Department). These systems are popular partly because of the library of synthesis techniques they provide.

We may investigate the use of Fugue in parallel computation. Because of its functional style, Fugue programs contain explicit parallelism in the sim ("simultaneous") construct. Even when there are data dependencies such as in the seq construct, lazy evaluation often defers signal computations so that the data dependencies can be resolved immediately. Then the signal processing can proceed in parallel. If we implement sounds in Fugue as streams, we can achieve even more parallelism by lazily evaluating streams. This could dramatically reduce the memory requirements for intermediate results in Fugue expressions. Even with large virtual memories and automatic garbage collection, storage is a serious problem in the current implementation.

An exciting potential of the lazy evaluation of streams is real-time execution.[8] This would require real-time garbage collection as well. We have not yet explored many opportunities for the compilation and optimization of Fugue behaviors.

Many of the ideas we use in Fugue seem appropriate for computer graphics and computer animation. The idea of behavioral abstraction seems to fit nicely with graphical transformations (for example, "make this truck longer") and also with action in animations ("run faster"). In computer animation, Fugue's notions of explicit timing and constructs for parallel and sequential behavior might be useful. For images, new constructs might be added to represent spatial as well as temporal relationships.

We could extend Fugue's semantics in several ways. Currently, only someone with a fair understanding of how contexts are implemented can extend the context in Fugue. We should make this simpler. Fugue should also support the use of MIDI (musical instrument digital interface) files as scores so that composers can use existing music editors as data sources. Another improvement would be to allow time-varying transformations,[9] using signals in place of real numbers to achieve musical effects such as accelerando (gradual increase in overall tempo) and crescendo (gradual increase in overall loudness). Also, Fugue must support multidimensional signals. We plan to make these changes in a future version. ■

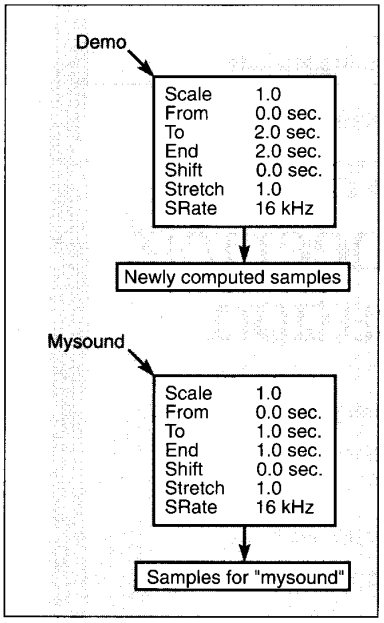

Figure 6. Memory structure of Demo after the third operation in Figure 3. The system computes samples according to the transformations in Figure 5 and caches the resulting samples, replacing the previous data structure with a new one.

Acknowledgments

We thank Carnegie Mellon University and Yamaha Music Technologies for their support of this work. Dean Rubine provided Lisp consultation for the production of *Spomin*, and we used his Strips library to produce the graphical score. George Polly contributed some of Fugue's signal processing functions.

References

1. M.V. Mathews, *The Technology of Computer Music*, MIT Press, Boston, 1969.
2. F.R. Moore, *Elements of Computer Music*, Prentice Hall, Englewood Cliffs, N.J., 1990.
3. B. Vercoe, "Csound: A Manual for the Audio Processing System and Supporting Programs," MIT Media Lab, MIT, Cambridge, Mass., 1986.
4. A.J. Field and P.G. Harrison, *Functional Programming*, Addison-Wesley, Reading, Mass., 1988.
5. R.B. Dannenberg, "Expressing Temporal Behavior Declaratively," *Carnegie Mellon Computer Science 25th Anniversary Proc.*, Addison-Wesley, Reading, Mass., 1991.
6. R.B. Dannenberg and C.L. Fraley, "Fugue: Composition and Sound Synthesis with Lazy Evaluation and Behavioral Abstraction," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1989, pp. 76-79.
7. D. Jaffe and L. Boynton, "An Overview of the Sound and Music Kit for the Next Computer," *Computer Music J.*, Vol. 13, No. 2, Summer 1989, pp. 48-55.
8. R.B. Dannenberg, "A Runtime System for Arctic," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1990, pp. 185-187.
9. R.B. Dannenberg, "The Canon Score Language," *Computer Music J.*, Vol. 13, No. 1, Spring 1989, pp. 47-56.

Roger B. Dannenberg is a senior research computer scientist at Carnegie Mellon University. His research interests include programming-language design and implementation, and the application of computer science techniques to the generation, control, and composition of computer music. He

is codirector of the Piano Tutor project, whose goal is applying music understanding and expert system technology to music education. He frequently performs jazz and experimental music on trumpet or electronically.

Dannenberg received a BSEE from Rice University in 1977, an MS in computer engineering from Case Western Reserve University in 1979, and a PhD in computer science from Carnegie Mellon in 1982. He is a member of Phi Beta Kappa, Sigma Xi, Tau Beta Pi, Phi Mu Alpha, ACM, and SIGCHI, and research coordinator for the Computer Music Association.

Christopher Lee Fraley is a computer design engineer at Microsoft Corporation, where he is working on graphical user interface builders. His interests include computer applications in music and poetry. He received his BA in computer engineering from Carnegie Mellon University in 1989.

Peter Velikonja is an assistant professor of oboe and theory at West Virginia University and is active as a performer and composer. He has performed with several major orchestras, including the Chicago Symphony and the Metropolitan Opera Orchestra. At the School of Computer Science at Carnegie Mellon University, he writes music using digital synthesis and develops other music software.

Velikonja received his training at Northwestern University; the Folkwang Musikhochschule in Essen, Germany, on a Fulbright grant; and the Mannes College in New York City.

Dannenberg can be reached at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. His e-mail address is dannenberg@cs.cmu.edu.