ABSTRACT BEHAVIORS FOR STRUCTURED MUSIC PROGRAMMING

Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science

ABSTRACT

Music Behaviors are introduced as a way to conceptually organize computation for music generation. In this abstraction, music is organized hierarchically by combining substructures either in sequence or parallel. While such structures are not new to either computer music or computer science, an efficient and simple real-time implementation that does not require multiple threads or translation to data structures is offered, making this abstraction more appropriate in a variety of languages and systems where efficiency is a concern or where existing support is lacking.

1. INTRODUCTION

Computer music presents interesting programming challenges. Unlike traditional programming tasks, where the goal is to compute an answer or a response quickly and efficiently, music demands output with specific and often sophisticated timing, including potentially many parallel and synchronized streams of information.

These particular requirements have led to innovation in programming languages and systems for music. Just a few examples are MUSIC *N* [9], FORMES [12], FORMULA [1], Nyquist [4], Siren [10], MAX [11], SuperCollider [8], and JMSL [7]. Aside from enabling interesting music applications, music languages undoubtedly affect the music they are intended to express. Music programs are often used as sketches and experiments to explore ideas, with no specific end result in mind. It is important, therefore, to offer languages that can express musical ideas directly and conveniently.

In this short paper, I discuss one style or design pattern that has been found useful for algorithmic music generation. The approach, called *Music Behaviors,* has been found to greatly simplify some practical programming problems, making it easy to express and modify music computation. Music Behaviors enable the programmer to describe musical "events" (really, any behavior that extends over time, such as a note, a sound object, silence, or a sound clip) and to combine events through sequential and parallel composition.

One danger in presenting work of this kind is that the reader's first response is likely to be "but I could solve problem *X* in system *Y* by doing *Z*." Almost any programming problem can be solved many ways, so the question is not so much "is it *possible* to express *X* in *Y*," but rather "how *directly* and *naturally* can I express *X* in system *Y*?" Evaluations of this kind are subjective, but nevertheless useful.

Music Behaviors are very similar to Composables in JMSL [7]. This paper extends that work by (1) providing an implementation without threads, (2) suggesting new termporal constructs, (3) presenting Music Behaviors as a generally applicable approach not tied to a specific language or system, and (4) showing that the syntactical hierarchy can mirror the musical hierarchy, improving the notation. The present work also introduces some terminology and concepts that may help to clarify the design space of language support for expressing and organizing temporal behavior.

2. DEFINING THE PROBLEM

The notion of sequential and parallel composition has been around many years. Aside from generic programming constructs like "split" and "join" for parallelism, sequential execution is seen in almost all programming languages. There are music-specific languages such as LOCO [6] and Canon [3] that directly support sequential and parallel composition. There are some subtle details and distinctions to be made, however. First, there is the question of whether durations are *internal* or *external* to the sequenced objects. Second, there is the question of whether the sequenced objects are computed *on-line* or *off-line*. These distinctions are important factors in determining exactly what can be expressed.

2.1. On-Line vs. Off-Line Representations

In many systems, music is represented as a data structure. For example, Buxton's SSSP [2] introduced a hierarchical data structure that supported notions of sequential and parallel composition. With data, one can look ahead to find and schedule future events. For example, to play a structured composition with sequential and parallel sub-structures, one can simply traverse the data, enumerating all events and their times, and build a "flat" schedule of events to perform sequentially. Since the data is available ahead of the music performance, we call this an *off-line* representation.

Computational, or program representations produce behaviors as a side-effect of program evaluation. In general, one cannot enumerate all the events and their times because that would require the program to be executed in advance of playback. Systems like FORMULA [1] that compute music events and controls in time order (and usually in real time) are *on-line* representations.

2.2. Internal vs. External Duration

The distinction between internal and external durations is best illustrated with an example. Suppose you have a sequence of events [A, B, C] to perform. The simplest way to do this is just to write a sequential program:

```
A; B; C;
```

but in single-threaded systems, this will prevent the program from doing anything else until the sequence completes. The standard solution is to *schedule* behaviors like [A, B, C] as a set of distinct events that execute quickly and return control to the scheduler. In this way, the events of many concurrent tasks can be interleaved. So, we need to schedule [A, B, C] as a sequence. If we know the durations of these events, we can compute when they finish and write something like:

```
schedule A at now
schedule B at now+dur(A)
schedule C at now+dur(A)+dur(B)
```

This is the *external duration* case. Either we have some way to determine the duration by querying an object or reading a data structure, or we can specify a desired duration to each member of the sequence

The other case is *internal duration*. Suppose we do not know the durations of A, B, and C. This not a problem for the sequential execution case: by definition, sequential execution will evaluate B when A finishes and evaluate C when B finishes. But how do we implement this with a scheduler? One way to do this is to modify A and B to invoke the next element of the sequence based on an internally derived duration. Using this approach, the implementation of A becomes:

```
def A():
   var dur = compute_duration_of_A
   actions_that_implement_A
   schedule B at now+dur
```

But notice that this solution requires an internal modification to all but the last elements of the sequence. This makes the program hard to read and modify, and A cannot be performed without starting the sequence ABC.

2.3. The Problem Statement

We want to support real-time music programming with a "natural" programming style that makes programs easy to specify, read, and modify. The specific problem is to support hierarchical sequential and parallel structures represented by *computation* and with *internal duration* determination.

For example, it would be nice if we could write programs that look something like this:

```
par( seq( A, B, A ),
     seqrep(10, C) )
```

which says to play A, B, A in sequence, starting at the same time ("par"-allel) as 10 repetitions of C. The details of A, B, and C are specified elsewhere (this is a feature), and might contain additional structure.

One solution to this problem is to use threads for parallel composition. Using *parbegin* and *parend* to denote the creation of threads to evaluate statements in parallel, one might write the following:

```
parbegin
   begin A(); B(); A(); end
   for i= 0 to 10 begin C() end
parend
```

This is a perfectly good solution except that it requires the creation of threads. Threads typically require more space and computational overhead than procedure calls or even scheduled events, and threads can make programs more difficult to understand because of unanticipated interactions between threads. FORMULA [1] and ChucK [13] are examples that use threads to express parallelism and implement clocks and scheduling carefully to support precisely coordinated timing.

Another solution is based on either polling or some kind of notification mechanism. The general idea is that sequential behaviors include a "supervisor" that monitors the progress of the sequence and starts the next item when the precursor completes. The problem here is that the programmer must add some explicit signaling so that the supervisor can know when each item of the sequence completes. To avoid blocking other computation while waiting for items to finish, the supervisor reschedules itself periodically and polls for progress. This generates extra work and also quantizes timing to the time steps at which poling takes place.

3. BEHAVIOR IMPLEMENTATION

The solution I propose eliminates thread creation and polling, and provides ready-made sequential and parallel behaviors in the form of classes. Instances of these classes (objects) provide a place to keep track of state and progress so that multiple instances of behaviors do not interfere with one another. The solution uses explicit notifications to eliminate polling. Finally, behaviors can be accurately timed, performing exactly as intended, limited only by floating point resolution. Behaviors are inherently composable, simplifying the expression of complex parallel and sequential timed execution. Abstractly, a Behavior is a set of computations with a starting time and an *internally* determined duration. The duration need not be known until the moment that the behavior ends.

A Python- or Serpent-like [5] pseudo language is used for examples that follow. Note that: Nested program structure is indicated by indentation, procedures defined (using "def") within a class definition become methods of the class, an instance of the class Myclass is created by the expression Myclass(), and the new instance is initialized automatically by a call to the init() method.

In this approach, the heart of every program is a single thread that runs periodically (e.g. every 1 ms) to check if a previously scheduled event (a call to a method) should be run or if input needs to be handled. All events are assumed to run quickly and return to the scheduler. Interleaving these short computations within one thread simulates parallelism.

As can be seen below, the Behavior class provides a very simple interface: a Behavior is started by calling *run* (normally the parent does this). When the Behavior

ends, it should call its own *done* method. When a child finishes, *next* is called. While running, a behavior can call *tlocal* to read elapsed real time. Subclasses of Behavior typically override *run* and *next*:

```
class Behavior:
   // this is a comment; here are instance vars:
   var parent // another Behavior or nil
   var tstart // start time
   // call run method to start this behavior
   def run()
      tstart = get_time()
      next() // subclasses override next
   // call done method at end of behavior
   def done()
      if parent: parent.next()
   // next method is called when child is done
   def next()
      return nil // default is do nothing, return nil
   // get the elapsed real time
   def tlocal()
      return get_time() - tstart
```

The *get_time* function returns the current logical time from the scheduler, *i.e.* the time at which, *ideally,* this computation should take place. This is normally just the real time at which a computation is *scheduled* to run, ignoring any latency or computation time. By working with idealized time, computation time and system latency do not accumulate to create long-term scheduling errors, and multiple behaviors can easily synchronize.

Here is an implementation sketch of Note, which selects a pitch and duration and plays the note. Since the details of sound generation by MIDI or direct synthesis are unimportant here, they are omitted:

```
// Note is a subclass of Behavior:
class Note (Behavior):
   def run()
      var pitch = compute_a_suitable_pitch
      var dur = compute_a_suitable_duration
      of_course_other_parameters_are_possible
      synthesize_note(pitch, dur, other_params)
      // cause uses scheduler to send a message
      // ('done') to this object after dur secs:
      cause(dur, 'done')
```

Now, we can write a complex behavior such as our earlier example. In this language, square brackets are array constructors, e.g. "[1, 2, 3]" denotes an array.

```
score = Par([ Seq([ Note(), Note(), Note() ]),
            Seqrep(10, Note()) ])
score.run()
```

In practice, it would be odd to make so many copies of Note. Instead, either Note would take initialization parameters so that each copy could be different, or different behaviors, e.g. A, B, and C, would be used in place of Note.

The following is an implementation of Seq, which runs each child in sequence. A counter, *i*, is used to remember which element of the *children* array to run next. When *next* is called as a consequence of the child calling *done*, *i* is incremented and the next child is started. When the last child completes, the Seq is done. Seq inherits from Behavior, but most of the methods are overridden:

```
class Seq (Behavior):
   var i, children // instance variables
   def init(c)
      children = c // copy to instance var
   def run() // run starts this behavior
      i = -1 // this will be updated by next
      super.run() // invoke superclass's run method
   def next()
      i = i + 1
      if i < len(children)
         // set child's parent to this Seq object:
         children[i].parent = this
         children[i].run() // start the ith child
      else // no more children, so sequence is done
         done() // inform parent we're done
```

The following is an implementation of Seqrep, which is similar to Seq, but repeats a child behavior *n* times:

```
class Seqrep (Behavior):
   var i, n, child
   def init(reps, c) // repeat child behavior c
      n = reps
      child = c
      child.parent = this // this object is c's parent
   def run()
      i = -1 // this will be updated by next
      super.run() // invoke superclass's run method
   def next()
      i = i + 1
      if i < n
         child.run() // start child again
      else
         done() // inform parent we're done
```

Similarly, the Par class runs children in parallel, finishing when the last child is done:

```
class Par (Behavior):
   var children, count
   def init(c)
      children = c
   def run()
      tstart = get_time()
      count = 0
      for c in children // start them all at once
            c.parent = this
            c.run()
   def next() // called as each child finishes
      count = count + 1
      if count = len(children)
         done()
```

To implement Parrep, which invokes *n* copies of a child, we need some way to get copies of objects rather than reusing the same object (as in Seqrep). In a dynamically typed language like this, we can allow the child to be specified several ways: If the child is a function, the function is called to create an object that is the child to run. If the child names a subclass of Behavior, instances of the subclass are created and run.

4. DISCUSSION AND EXAMPLE

The Behavior class and associated subclasses offer a simple but powerful way to compose temporal behaviors into complex musical structures, especially when the music generation is described computationally. This approach has several advantages:

- It is simple and direct to write nested structures of sequential and parallel activities.

- The resulting programs are efficient: They do not require the creation of threads, the allocation of many objects, or polling.
- By representing the music computationally, sounds and even musical structure can respond to real-time events, in contrast to event lists.
- New control constructs can be created easily, such as "repeat A until time T," "run A after delay D1 and followed by delay D2," or "repeat A until predicate P is true."

Below, we offer a slightly more realistic example to illustrate this approach. Here, melodic lines are generated by making sequences of phrases, where each phrase is a sequence of short notes followed by a long one. The note is described below.

```
class Nt (Behavior):
  var p // pitch as MIDI key number (60 = C4)
  def run()
    // select random interval in range -9 to -4
    p = p + irandom(6) – 9
    if p < 36: p = p + 12
    var dur = 0.2 // seconds
    if parent.i == parent.n – 1 // last in phrase
      dur = 5 * dur
    play_note(dur, pitch)
    cause(dur, 'done')
```

The Phrase class plays a sequence of Nt notes:

```
class Phrase (Seqrep):
  def init()
    super.init(0, Nt()) // rep count computed later
  def run()
    // select random starting pitch from 36 to 59
    child.p = 36 + irandom(24)
    // select number of repetitions
    n = 1 + irandom(4) // from 1 to 4
    super.run()
```

And finally, we generate a sequence of Phrases using Seqrep:

```
melody = Seqrep(10, Phrase())
melody.run()
```

In most of these examples, objects are reused. For example, the Phrase class is run 10 times, and each Phrase creates one instance of Nt, which is run a random number of times. In some cases, this is a "feature," e.g. Nt computes each pitch by adding an interval to the previous pitch saved in an instance variable. However, in other cases, code will be simpler and more readable if new instances are created for each use. The details, efficiency, and desirability of creating instances are language-dependent, but should certainly be considered.

5. CONCLUSION

Music Behaviors are a conceptually simple but powerful way to organize and express temporal music structure and schemes for music generation. While any given program or music specification can be implemented in many ways using many different abstractions, the conceptual organization of programs is an important part of the compositional process. Music Behaviors abstract activities that take place over intervals of time. These behaviors can be composed sequentially and in parallel using lists or iteration. The implementation is such that nested expressions directly convey the nested structure of behaviors, making them easy to write and modify. While nested sequential and parallel structures are hardly new, the Music Behaviors approach facilitates their use by offering an efficient, simple implementation that does not require thread creation, polling, or translation to data structures.

REFERENCES

[1] Anderson, D. and Kuivila, R. "A System for Computer Music Performance." *ACM Transactions on Computer Systems,* 8(1) (Feb. 1990), pp. 56-82.

[2] Buxton, W., Sniderman, R., Reeves, W., Patel, S. & Baecker, R. "The Evolution of the SSSP Score Editing Tools." In Roads & Strawn, *Foundations of Computer Music*. MIT Press, Cambridge, (1985), pp. 376-402.

[3] Dannenberg, R. B. "The Canon Score Language." *Computer Music Journal* 13(1), (Spr 1989), pp. 47-56.

[4] Dannenberg, R. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis," *Computer Music Journal*, 21(3) (Fall 1997), pp. 50-60.

[5] Dannenberg. "A Language for Interactive Audio Applications," in *Proceedings of the 2002 Int. Computer Music Conf.* San Francisco: International Computer Music Association, (2002), pp. 509-15.

[6] Desain, P., & Honing, H. "LOCO: a composition microworld in Logo." *Computer Music Journal*, *12*(3), (1988), pp. 30-42.

[7] Didkovsky, N. and Burk, P. "Java Music Specification Language, An Introduction and Overview," in *Proceedings from the 2001 International Computer Music Conference.* San Francisco: International Computer Music Association, (2001), pp. 123-126.

[8] McCartney, J. "Rethinking the Computer Music Programming Language: SuperCollider." *Computer Music Journal*, 26(4), (2002), pp. 61-68.

[9] Mathews, M., Miller, J., Moore, F. R., Pierce, J., and Risset, J. C. *The Technology of Computer Music.* Cambridge, Mass.: MIT Press, 1969.

[10] Pope, S. T. and Ramakrishnan, C. "Recent Developments in Siren: Modeling, Control, and Interaction for Large-scale Distributed Music Software," in *Proc. of the 2003 ICMC.* San Francisco: International Computer Music Assoc., (2003), pp. 5-9.

[11] Puckette, M. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal*, 15(3), (1991), pp. 68-77.

[12] Rodet, X., Cointe, P., "Formes: Compostion and Scheduling of Processes," *Computer Music Journal* 8 (3), (Fall 1984), pp. 32-50.

[13] Wang, G. and Cook, P. "ChucK: A Concurrent, On-the-fly Audio Programming Language." In *Proceedings of the International Computer Music Conference*, San Francisco: International Computer Music Association, (2003), pp. 217-225.