

Modeling Musical Structures as EventGenerators

Stephen Travis Pope
ParcPlace Systems, Inc.
1550 Plymouth St.
Mountain View, California, 94043 USA
EMail: stp@ParcPlace.{com, uucp}

ABSTRACT

There is a broad range of music description languages. The common terms for describing musical structures define a vocabulary that every musician learns as part of his or her training. The terms we take for granted in describing music can be used for building generative software description languages.

This paper describes recent work modeling higher-level musical structures in terms of objects that understand specialized sub-languages for creation of—and interaction with—musical structures. The goal is to provide tools for composers to describe compositions by incrementally refining the behaviors of a hierarchical collection of structure models.

INTRODUCTION

Verbal and written descriptions of musical experience and musical structures provide a rich vocabulary for representing this information in a software system. A software music description language using the conceptual and operational languages of common musical description has been developed and used in several contexts based on modeling the vocabulary of verbal descriptions of musical structures.

The EventGenerator class hierarchy described here, called EGen, is an easily-extensible framework with which composers can model many types of musical structures and build real-time interactive user interfaces for manipulating them. This paper presents a discussion of the EGen background and theory, and gives several music description examples using EGen. A longer version of this manuscript, with code examples and figures, is available from the author.

The EGen System is being developed in the *HyperScore ToolKit* a portable computer music software library built in the Smalltalk-80 (™ of ParcPlace Systems) programming environment. The effect of the environment and its design methodology on the structure and implementation of EGen will be described. A companion paper (Pope 1989) describes the process of ‘composition by refinement,’ a technique of using EGen in development of musical works.

BACKGROUND

To set the stage for the discussion, we present several examples of verbal descriptions of middle-level musical structures.

“a Major triad on C in the first inversion”

“a cluster of the notes from c4 to f4”

“a mordent around a5”

“the retrograde inversion of the 12-tone row with this rhythm”

“an 8-second stream that moves from selection within the C-Major to the A-Major triad”

“an ostinato on the oboe theme from measures 24 and 5”
“a 12-bar blues bass line in E”
“a Markhov chain running on the given 12-tone row”

The order and grouping of these examples will become apparent. It is important to recognize that they fall into three general categories. Some of them describe relationships among events in composite event lists (e.g., chords described in terms of a root and an inversion); while others describe melismatic embellishments of—or processes on—a note or collection of notes (e.g., mordents); and others are descriptions of continuous event list-generating processes in terms of their parameters (e.g., ostinati or Markhov transition table processes).

OBJECT-ORIENTED ANALYSIS AND DESIGN

Object-oriented software uses message-passing interfaces derived by analysis of the communication paths in the domain of discourse. The development of an object-oriented class hierarchy that reflects the model relationships of the domain can be aided by collecting a set of description examples stated in terms of message-passing protocols. A powerful set of classes can be developed by discovering the sub-structure and relationships between models in terms of recognizing local sub-languages, and of interpreting them in terms of abstraction and sharing of behavior. Examples we might start with to model our music description domain are messages to create and play a chord and an ostinato, as shown below.

```
"create a chord (c e g)"
Chord majorTriadWithRoot: 'c3' inInversion: 0
"create and play another chord (c f a c)"
(Chord majorTetradWithRoot: 'f4' inInversion: 2) play

"create an ostinato on the given event list"
Ostinato newNamed: #riff onList: someEventList
"start it playing (repeat the list)"
(Ostinato named: #riff) play
```

The main design issue that influences the extension of the EventGenerator class hierarchy is the trade-off between the desired grain size of the EventGenerator classes and the number of classes needed to describe a real composition. Taking the example of chords, one could envision building a large hierarchy of chord classes, each capable of producing one type of chord in all of its inversions. For any large-scale tonal composition, the number of chord classes and variation among their protocol (mostly their instance creation messages) would grow quite large (e.g., *CharlieParker1957Chord minor11thOn:*). On the other side of the scale, one could build the same system as one large chord class with a large number of instance creation messages for different chords (e.g., *Chord CharlieParker1957Minor11thOn:* or *Chord fromPitchSet: #(0 4 7 11 14 17)*).

This tension is well-known in object-oriented software design and implementation. One of the extreme cases is a design that uses abstraction heavily and develops into a wide and/or deep class hierarchy with very many classes that have only a few (perhaps one) methods each. This case is called class explosion. The other end of the scale is a design that leads to a very large and indivisible class (all messages in class Chord), which is known as a monolithic class problem.

The solutions for both of these cases is the judicious use of abstraction and parameterization of classes. One answer might be the use of a smaller number of Chord classes, each of which has parameters for setting its type or its pitch set. Another workaround for this problem would

be to have a single Chord class that uses message protocols for categorizing its instance creation messages (e.g., creating minor chords, creating whole-tone chords, etc.).

There are many other design questions that arise when modeling musical structures within the EventGenerator inheritance hierarchy, for example one could ask the question whether a roll is a chord with delays, or an ostinato with one event in its event list?

EVENTGENERATOR ABSTRACTIONS

Most of the description examples shown above can be implemented in a simple set of EventGenerator classes. The challenge is to make an easily-extensible framework for composers whose compositional process will consist of enriching the EventGenerator hierarchy for a specific composition. EventGenerators can generally either return an EventList, or can behave like processes, and be told to play or to stop playing. We view this dichotomy—between views of EventGenerators as event lists versus EventGenerators as processes—as a part of the domain, and differentiate on the basis of the musical abstractions. It might, for example, be appropriate to view an ostinato as a process that can be started and stopped, or to ask it to play thrice and return an event list.

SHARED BEHAVIOR OF EVENTGENERATORS

The list-like EventGenerators have messages such as *eventList* for returning an event list (which can be edited, played, etc.); the process-like EventGenerators are more likely to have *start*, *stop*, and *playTimes*: messages. The abstract classes we choose for modeling the most abstract categorization of EGen are *Cluster*—a description of the pitch set of a group of simultaneous events; *Cloud*—a pitch-time surface for stochastic or procedural selection; and *Ostinato*—a repetition of an event list as a process. We will describe each of them below, along with their concrete subclasses. The long version of this paper presents creation message and output examples for each of these.

Clusters

The Cluster classes describe a collection of pitches with no rhythm (i.e., they all occur simultaneously). The instance creation messages of these classes allow users to describe clusters in terms of their pitch set, or their structure (e.g., root, type and inversion). Concrete types of Clusters include chords and arpeggii. The class hierarchy of Clusters is:

- Cluster*—create with an arbitrary pitch set given a pitch or an event
primary behavior: return an event list
- Chord*—create with a root and an inversion
 - MajorChord*—knows major
 - MinorChord*—knows minor
 - PentatonicChord*—constrained chords
 - ... any number of chord classes (one method each) ...
- Arpeggio*—give it a cluster or chord and a delay time
- Roll*—give it a note and it will repeat it
 - Trill*—give it an event list
 - Mordent*—melismatic embellishment of a note
 - ... any number of embellishment description classes ...

Clusters also have instance creation messages for managing pitch and/or time lists separately; a Cluster may have either of these without the other and represent either a pure pitch set or rhythm. There are standard EventList messages for mapping pitch and time among EventList types, so that Clusters with pitch only or time only can be quite useful.

Clouds

The notion of Clouds comes from aleatoric music where collections of notes can be described by their contour. Aleatoric generators that select notes from a given range or set of pitches can be modeled as types of Clouds. The class hierarchy of Clouds looks like:

Cloud—abstract class

PodCloud—give it pitch, amplitude and voice ranges for selection (à la Truax)

DynamicPodCloud—give it starting and ending ranges

SelectionCloud—give it pitch, amplitude and voice sets for selection

DynamicSelectionCloud—give it starting and ending pitch/rhythm sets

Ostinati

Process-oriented generators usually take the form of Ostinati, repeating versions or variations of a given set of material or process input parameters. The simple Ostinato repeats his event list as long as he runs, while the other two types (Markhov- and SelectionOstinati) make variations based on either transition tables or controlled random selection. More complex processes can be used for writing algorithmic composition classes as EventGenerators. Mark Lentczner's bell peal classes fit nicely into the hierarchy here. The class hierarchy of Ostinati looks like:

Ostinato—general repeater

MarkhovOstinato—uses transition tables for repetition

SelectionOstinato—selects notes from the given event list

BellPeal—EGens that ring the changes (courtesy of Mark Lentczner)

The framework provides for easy extension of these base classes via object-oriented refinement, and some of the code and tape examples will show further EventGenerators.

EVENTGENERATOR EDITORS AND INTERACTION

Because most EGen can return EventLists, we can use HyperScore's standard EventListEditors to edit their output. The real fun, however, starts when we begin constructing special dialogs and interactors that allow us to interactively change the parameters of EGen. We have constructed several so-called playgrounds for experimenting in this area and have achieved very interesting results. The examples that will be presented include tools for interaction with:

Clusters—edit the pitch or start-time/duration lists separately;

Clouds—edit the contour parameters as 2-dimensional contours; or

Ostinati—edit the EventLists, start/stop, change selection criteria or table weightings.

CONCLUSIONS

We have tried to model objects for higher-level descriptions of musical structures on the vocabulary of common-practice music description. The result has been shown to be a very compact and—relatively speaking—natural description language for simple musical structures. The system is open and easily extensible within the context of a composer's personal style or of a given composition. The design of the EGen classes has shown several interesting aspects of musical structure, especially in the abstractions that allow for more reuse within the implementation code and more commonalities among the instance creation messages. Further development of this package will certainly demonstrate other interesting, perhaps deeper, abstraction levels for event generator modeling.

REFERENCES

Pope, Stephen T. 1987. "A Smalltalk-80 Music ToolKit." *Proceedings of the 1987 International Computer Music Conference*. San Francisco: Computer Music Association. Revised as *HyperScore ToolKit Version 5.2 Description and Examples*. Available from HyperScore User's Group, San Francisco.

Pope, Stephen T. 1989. "Composition by Refinement." *Proceedings of the 8th Colloquio di Informatica Musicale*. Associazione di Informatica Musicale Italiana, Venice Italy