The SmOKe music representation, description language, and interchange format

Stephen Travis Pope  
The Nomad Group, Computer Music Journal, CCRMA  
P. O. Box 60032, Palo Alto, California 94306 USA  
Electronic Mail: stp@CCRMA.Stanford.edu

ABSTRACT

The Smallmusic Object Kernel (SmOKe) is an object-oriented representation, description language and interchange format for musical parameters, events, and structures. The author believes this representation, and its proposed linear ASCII description, is well-suited as a basis for: (1) concrete description interfaces in other languages, (2) specially-designed binary storage and interchange formats, and (3) use within and between interactive multimedia, hypermedia applications in several application domains.

The textual versions of SmOKe share the terseness of note-list-oriented music input languages, the flexibility and extensibility of "real" music programming languages, and the non-sequential description and annotation features of hypermedia description formats. This description presents the requirements and motivations for the design of the representation language, defines its basic concepts and constructs, and presents examples of the music magnitudes and event structures. The intended audience for this discussion is programmers and musicians working with digital-technology-based multimedia tools who are interested in design issues related to music representations, and are familiar with the basic concepts of software engineering. Two other documents (Smallmusic 1992 and [Pope 1992]), describe the SmOKe language, and the MODE environment within which it has been implemented, in more detail.

1. INTRODUCTION

The desire has been voiced (ANSI 1992; Dannenberg et al. 1989; MusRep 1987; MusRep 1990; Smallmusic 1992), for an expressive, flexible, abstract, and portable structured music description and composition language. The goal is to develop a kernel description that can be used for structured composition, real-time performance, processing of performance data, and analysis. It should support the text input or programmatic generation and manipulation of complex musical surfaces and structures, and facilitate capture and performance in real time via diverse media. The required language have a simple, consistent syntax that provides for readable complex nested expressions with a minimum number of different constructs. The text of the language and its underlying representation will be the facility with which applications can be ported to it.

Smallmusic Object Kernel consists of primitives for describing the basic scalar magnitudes of musical objects, abstractions for musical events and event lists, and standard messages for building event list hierarchies and networks. Structures in the underlying music representation can be described in a text-based linear format, and in terms of the of in-memory data structures that might be used to hold them. It is intended that independent parties be able to implement compatible abstract data structures, extend interchange formats, and description languages based on the formal definition of SmOKe (Smallmusic 1992).

The naming conventions and the code description examples use the Smalltalk-80 programming language (Goldberg and Robson 1989), but the representation should be easily manipulable in any object-oriented programming language. For readers unfamiliar with Smalltalk-80, another document (available via InterNet ftp from the file named "reading. smalltalk" in the directory "anonymous@ccrma-fpl:Stanford.edu/pub/old") introduces the language's concepts and syntax to facilitate the reading of the code examples in the text. In this document, SmOKe examples are written between square brackets in small caps typeface.

2. REQUIREMENTS AND MOTIVATIONS

Several of the groups that have worked on developing music representations have started by drawing up lists of requirements on such a design, and separating out which items are truly determined by the underlying representation, and which are interface or application issues. The Smallmusic group developed the following list, using the results of several previous attempts (see citations above) as input. SmOKe shall provide or support:

- abstract models of the basic musical quantities (scalar magnitudes such as pitch, loudness or duration);
- sound functions, granular description, or other (non-note-oriented) description abstractions;
- flexible grain-size of "events" in terms of "notes," "grains," "elements," or "textures";
- description/manipulation levels including event, control, and sampled function;
- hierarchical event-tree (nested lists) for "parts," "tracks," or other parallel or sequential structures;
- separation of "data" from "interpretation" (what vs. how in terms of having interpretation objects such as the instru-
ment, note, voice, event, or performer/part abstraction); 
- abstractions for the description of "middle-level" musical structures (e.g., chords, clusters, or trills); 
- annotation of events supporting the creation of hierarchies (lattices) and hypermedia networks; 
- annotation including common-practice notation possible (application issue); 
- description of sampled sound synthesis and processing models such as sound file mixing or DSP; 
- possibility of building converters for many common formats, such as MIDI data, Audio, note lists, HyTime, DSP, 
  MIDI converter definitions, mixing scripts; and 
- possibility of preserving live performance in some rendition to the representation, and of interpreting it (in some rendition) 
  in real-time (application issue related to simplicity, terseness, etc.).

3. EXECUTIVE SUMMARY

The SmoKe representation can be summarized as follows. Music (i.e., a musical surface or structure), can be represented as a 
series of "events" (which generally last from tens of msec to tens of sec). Events are simply property lists or dictionaries. Events 
have named properties whose values are arbitrary. Those properties may be music-specific objects (such as pitches or 
properties), and positions of many common musical magnitudes are provided. Events are grouped into event lists as records 
consisting of relative start times and events. Event lists are events themselves and can therefore be nested into trees (i.e., an 
event list can have another event list as one of its events, etc.). They can also map their properties onto parameters of specific events. The 
means that an event can be "shared" by being in more than one event list at different relative start times and with different 
properties mapped onto it. Events and event lists are "performed" by the action of a scheduler passing them to an object 
representation, voice. Voice objects and applications determine the interpretation of events properties, and may use "standard" property 
names such as pitch, loudness, voice, duration, or position. Voices map event properties onto parameters of I/O devices; there 
are rich hierarchies of them. A scheduler expands and/or maps event lists and sends their events to their voices. Stored data 
databases can be defined and manipulated as breakpoint or summation parameters, "raw" data elements, or functions of the 
above. Sampled sounds are also describable, by means of synthesis "pitches," or signal processing scripts invoking a vocabulary 
of sound generation messages.

4. THE SmoKe LANGUAGE

4.1 Linear Description Language

The SmoKe music representation can be linearized easily in the form of immediate object descriptions and message 
expressions. These descriptions can be thought of as being declarative (in the sense of static data definitions), or procedural (in 
the sense of messages sent to class "factory" objects). A test file can be freely edited as a data structure, but one can compile it with 
the SmoKeTalk-80 compiler to "instantiate" the objects (rather than needing a special formatted reading function). The post-fix 
expression format taken from SmoKeTalk-80 (receiverObject keyword: optional/argument) is easily parseable in C++, LISP, 
Fortran, and other languages.

4.2 Language Requirements

The basic representation itself is language-independent, but assumes that the following immediate types are representable as 
ASCII/ISO character strings in the host language: 
- arbitrary precision integers (at least very large), 
- integer fractions (i.e., stored as numerator/denominator, rather than the resulting whole or real number), 
- 32- and 64-bit (-7, 12-precision) floating-point numbers, 
- arbitrary-length ASCII/ISO strings, 
- unique symbols (i.e., strings managed with a hash table), 
- 1- and 3-dimensional points (or n-dimensional complex numbers) (axial or polar representation), and 
- functions of one or more variables described as breakpoints for linear, exponential or spline interpolation, Fourier 
  sums, series, sample spaces, and probability distributions.

The support of block context objects (in SmoKeTalk, or closures in LISP), is defined as being optional, though it is considered 
important for complex scores, which will often need to be stored with interesting behavioral information. (It is beyond the 
scope of the present design to propose a metalogic for the interchange of algorithms.) Dictionaries or property association 
lists must also either be available in the host language or be implemented in a support library (as must unique symbols and 
evanescent associations in some cases [e.g., std. C]).

4.3 Naming and Persistence

The names of abstract classes are known and are treated as special symbols. The names of abstract classes are used wherever 
possible, and instances of concrete subclasses are returned as, in [FltPatch value: 2f] or [TscfPitch] both returning a Symbol-

permedia networks of events. Event properties can also be active blocks or procedures (in cases where the system supports activation at run-time as in Smalltalk-80 or Lisp), blurring the differentiation between events and "active agents." Events can be created either by messages sent to the class Event (which may be a macro or binding to another class), or more freely, by the concatenation of music magnitudes using the message "::=" (concatenative concatenation), as in the Smalltalk-80 message syntax. Applications should enable users to interactively edit the property lists of objects, and to browse network events via the network's or their links using flexible link description and filtering editors. Standard properties such as pitch, duration, position, control, and voice are manipulated according to "standard" semantics by many applications.

Figure 1: SmoKe MUsicMagnitude Model Abstractions and Implementation Classes

Event Examples

"Event creation examples--the verbose way (class messages)."

Event: (Event duration: (value: 1/2) pitch: (value: #02) loudness: (value: #4f)) playOn: voice

"create three events with mixed properties--the terser way"

([440 Hz], [1/4 beat]), [-12 dB], [Voice: vocal]

[38 kHz, 280 Licks, 56 val, [Voice: #4f]

[40 pitch, 0.21 sec, 0.37 ampl, [Voice: named: #89]]

"abstract props."

[MIDI-style props.]

"note-list style"

7. EVENT LISTS

Events are grouped into collections--a list is composed of associations between start times (durations start at the start time of the event list) and events or sub-lists (nested to any depth). Schematically, this looks like: (EventList (duration: event 1) (duration: event 2) ...) where (x => y) means association with key x and value y. Event lists can share their own properties, and can map these onto their events easily (as a class or lattice) or lazily ("performance")--that is, all have the property and link behavior, and special behaviors for mapping with voices and event modifiers. Event lists can be named, and when they are, they become persistent (until explicitly erased within a document or session).

The messages [anEventList add: anAssociation] and [anEventList add: anEventOrEventList at: anDuration], along with the corresponding event removal messages, can be used for manipulating event lists in the static representation or in performance. If the key of the argument to the add message is a number (ranging from one to the number of messages), it is assumed to be the value of duration in seconds or milliseconds, as "appropriate." Event lists also respond to Smalltalk-80 collection-style control structure messages such as [anEventList collect: aSelectionBlock] or [anEventList select: aSelectionBlock], though this requires the representation of context/locators. The behaviors for applying functions (see below) to the components of an event list look applicative (e.g., [anEventList apply: aFunctions to: anProperty]), or one can use event modifier objects to provide a statement representation of the mapping. Applications will use event list hierarchies for browsing and annotation as well as for score following and performance control. The use of standard link types for such applications as version control (with such link types as FixedFolder or FixedScript16384), is defined by applications and voices.

A named event list is created (named) in the first example below, and two event associations are added to it, one lasting 0 seconds by default, and the second at 1 sec. Note that the two events can have different types of properties, and the list creation messages such as [dur: durationValue pitch: pitchValue amp: ampValue]. The second example is the terse form of event list declaration using the behavior of (duration => event) associations such that (andMagnitude) => (event with given property dictionary). One can use dictionary-style shorthand with event associations to create event lists, as in the terse way of creating an anonymous (top-tier) event list with two events in the second example. The third example shows the first few notes from the c-minor of The Well-Tempered Clavichord in which the first note begins after a rest (that could also be represented explicitly as an event with a duration and no other properties). Note that there is one extra level of parentheses for readability.

Event List Examples

\[\text{Lists--the verbose way}\]

\[\text{(EventList newNamed: #test1 add: (0 => (Event dur: 1/4 pitch: #c3 ampl: #m7))) add: (1 => (Event new dur: 6.000 ampl: 0.3772 sound: #s3#2w))}\]

\[\text{Lists--concatenation of events or (dur => event) associations.}\]

\[\begin{aligned}
440 Hz, (1/2 beat), 44.7 dB, (0 \rightarrow (1.396 sec, 0.714 ampl) sound: #s3#2w; phoneme: \text{#m7}) \quad \text{(playing theme.)}
\end{aligned}\]

\[\begin{aligned}
The\text{\, next time}
duration\text{ pitch} & \quad \text{voice} \\
0.5 \text{ beat} & \quad (1/4 \text{ beat}), (c'3 \text{ pitch}), (\text{voice: harpsichord}),
&(1/4 \text{ beat}), (b'2 \text{ pitch}), (1/2 \text{ beat}), (e'3 \text{ pitch}),
&(1/2 \text{ beat}), (g'2 \text{ pitch}), (1/2 \text{ beat}), (\text{a-flat2 \text{ pitch}})
\end{aligned}\]

8. EVENT GENERATORS AND MODIFIERS

EventGenerator and EventModifier packages provide for music description and performance using generic or composition-specific middle-level objects. Event generators are used to represent the common structures of the musical vocabulary such as melodic, rhythmic, or compositional algorithms. Each event generator subclass knows how it is described—e.g., a chord with a specified inversion, an ornament with an event list and repeat rate—and can perform itself once or repeatedly, acting as either a control structure, or a process, as appropriate. EventModifier objects hold on to a function and a property name; they are used to apply their functions to any property of an event list lazily or eagerly. Event generators and modifiers are defined elsewhere.

9. FUNCTIONS, PROBABILITY DISTRIBUTIONS AND SOUNDS

Function objects not only define functions of one or more variables, several types of discrete or continuous probability distributions, and sample and sampled sounds. The description of these facilities is, however, outside the scope of this paper, and the reader is referred to [Goldman 1992].

10. VOICES AND STRUCTURE ACCESSORS

The "performance" of events takes place via Voice objects. Event properties are assumed to be independent of the parameters of any synthesis algorithm or structure. A voice object is a "property-to-structure accessor" that knows about one or more input or output formats for SmoKe data (e.g., MIDI, note list files, or DSP commands). A StructureAccessor is an object that can act as a translator or protocol converter. An example might be an accessor that responds to the typical messages of a tree node structure of a hierarchy (e.g., what's your name, who are you children and sub-nodes, etc?). Who are you? What are you? What are you children? etc. The accessor that knows how to apply that language to navigate through a hierarchical event list (by querying the event list's hierarch) to SmoKe supports the description of voices and structure accessors in scores so that performance information or alternative accessors can be embedded. The goal is to be able to associate a score with possibly complex real-time control objects that maintain structure or interpretation. Voices and event interpretation are described in [Pope 1992].

12. SMOKE SCORE EXAMPLE

BomSoCou, sections with declarations of variables, naming of event lists, event definition, functions and event modifiers, can be freely mixed. Note that one tries to avoid actually typing SmoKe at all anyway, leaving that to interactive assistants, algorithmic generation or manipulation programs, or read/write interfaces to other media, such as MIDI. The SmoKe behavior is not shown in the components of a SmoKe score as a composition with several sections defined in different styles. With these declarations are placed between vertical bars.

\[\begin{aligned}
\text{variables of variable names and top-level event list.}
\end{aligned}\]

\[\begin{aligned}
\text{sections 1 section 2}
\end{aligned}\]

\[\begin{aligned}
\text{section 1 events, in parallel or sequentially...}
\end{aligned}\]
"section2 := (0 beat) => {...event1...} (1/4 beat) => (event2),
...section 2 events...", 
(2109/4 beat) => (event3308)),
"event list composition may be placed anywhere"
"add the sections in sequence",
"place add: section1; add: section2."
"add one measure of rest after section 2."
"position from data arrays.
"values: (Eventlist(18)Properties: (duration; loudness; pitch)
values: (Array with(200 270 300 150 200 250 ... ));"duration"
values: (Array with(80 90 100 120 130 150 ... ));"loudness"
values: (Array with(90 100 110 120 130 140 ... ));"pitch"
"Add an event with the given samples (you want low-level? we got low-level!)
"add event: (Event rate: 44100 channels: 1 samples; 0 0 0 0 37 317 441 ... ).
"Declare global (named) event modifiers, functions, etc.
(Rubato newnamed: # Tempo function: (...tempo spline function...)) property: # start time,
"optionally declare voices, accessors, other modifiers, etc.

13: CONCLUSIONS

The Smallmusic Object Kernel (SmOKe) is a representation, description language and interchange format for musical data that eases the creation of concrete description interfaces, the definition of storage and interchange formats, and is suitable for use in multimedia, hypermedia applications. The SmOKe description format has several features, ranging from very readable type terms, and covering a wide range of signals, events, and structure types from sampled sounds to compositional algorithms. SmOKe can be viewed as a procedural or a declarative description; it has been designed and implemented using an object-oriented methodology and is being tested in several applications. More explicit documents describing SmOKe and the Smalltalk-80 implementation of the system in the MODE system are freely available via Internet file transfer.

14: ACKNOWLEDGEMENTS

SmOKe, and the MODE of which it is a part, is the work of many people. Craig Latta and Daniel Oppenheim came up with the names Smallmusic and SmOKe. These two, and Guy Garnett and Jeff Gomzi, were part of the team that discussed the plans of SmOKe, and commented on its design documents (Smallmusic 1992).

REFERENCES

Smallmusic 1991, E. G. Garnett, J. Gomzi, C. Latta, D. Oppenheim, S. T. Pope et al., Smallmusic discussion group notes, Creo 1-6 documents (from which this document was derived), and MODE User Primitive specification available from Smallmusic@XCF.Berkeley.edu as email or via anonymous InterNet ftp from the server crma.ftp.Stanford.edu in the directory pub/440 (see the README file there).