The SmOKe music representation, description language, and interchange format

Stephen Travis Pope
The Nomad Group, *Computer Music Journal*, CCRMA
P. O. Box 60632, 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, to be 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 the 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 their 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 test 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, concrete 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.t" in the directory "anonymous@ccrma-ftp.Stanford.edu:/pub/st80"), 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 *sans-serif italic* 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 abstractions);

- abstractions for the description of "middle-level" musical structures (e.g., chords, clusters, or trills);
- annotation of events supporting the creation of heterarchies (lattices) and hypermedia networks;
- annotation including common-practise notation possible (application issue);
- description of sampled sound synthesis and processing models such as sound file mixing or DSP;
- possibility of building convertors for many common formats, such as MIDI data, Adagio, note lists, HyTime, DSP code, instrument definitions, mixing scripts; and
- possibility of parsing live performance into some rendition in 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; they can have named properties whose values are arbitrary. These properties may be music-specific objects (such as pitches or spatial positions), and models 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 their component events. This 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 interpretation object or 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 can be a rich hierarchy of them. A scheduler expands and/or maps event lists and sends their events to their voices. Stored data functions 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 "patches," or signal processing scripts involving a vocabulary of sound manipulation 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 text file can be freely edited as a data structure, but one can compile it with the Smalltalk-80 compiler to "instantiate" the objects (rather than needing a special formatted reading function). The post-fix expression format taken from Smalltalk-80 (*receiverObject keyword: optionalArgument*) is easily parseable in C++, Lisp, Forth, 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-place precision) floating-point numbers,
- arbitrary-length ASCII/ISO strings,
- unique symbols (i.e., strings managed with a hash table),
- 2- 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 Smalltalk), 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 metalanguage 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 even associations in some cases [e.g., std. C]).

4.3 Naming and Persistency

The names of abstract classes are known and are treated as special globals. The names of abstract classes are used wherever possible, and instances of concrete subclasses are returned, as in [*Pitch value: 'c3'*] or [*'c3' pitch*] both returning a Symbol-Pitch instance (strings are written between single-quotes in SmOKe [and Smalltalk]). All central classes are assumed to support "persistency through naming" whereby any object that is explicitly named gets stored in a global dictionary under that name until explicitly released. What the exact (temporal) scope of the persistency is, is not defined here. The (lexical) extent is assumed to be as SmOKe "document" or "module."

4.4 Score Format

A SmOKe score consists of one or more parallel or sequential event lists whose events may have interesting properties and links. Magnitudes, events, and event lists are described using class messages that create instances, or using immediate objects and the terse post-fix operators. These can be named, used in one or more event lists, and their properties can change over time. There is no pre-defined "level" or "grain-size" of events; they can be used at the level of notes or envelope components. patterns, grains, etc. The same applies to event lists, which can be used in parallel or sequentially to manipulate the sub-sounds of a complex "note," or as "motives," "tracks," or "parts." Viewed as a document, a score consists of declarations of, or messages to, events, event lists and other SmOKe structures. It can resemble a note list file or a DSP program. It is structured as executable Smalltalk-80 expressions, and can define one or more "root-level" event lists. There is no "section" or "wait" primitive; sections that are supposed to be sequential must be included in some higher-level event list to declare that sequence. A typical score will define and name a top-level event list, and then add sections and parts to it in different segments of the document.

5: SmOKe MUSIC MAGNITUDES

Abstract descriptive models for the basic music-specific magnitude types such as pitch, loudness or duration are the foundation of SmOKe. These are similar to Smalltalk-80 magnitude objects in that they represent partially- or fully-ordered scalar or vector quantities with (e.g.) numerical or symbolic values. Some of their behavior depends on what they stand for, and some of it on how they're stored. These two aspects are the objects' "species" and their "class." The pitch-species objects *440.0 Hz* and *'c#3'*, for example, share some behavior, and can be mixed in arithmetic with (ideally) no loss of "precision." The expression (*261.26 Hz + MIDI key number 8*) should be handled differently than (*1/4 beat + 80 msec*). The class of a music magnitude depends on the "type" of their values (e.g., floating-point numbers or strings), while their species denote what they represent. Only the species is visible to the user.

Music magnitudes can be described using prefix class names or post-fix type operators, e.g., [*Pitch value: 440.0*] or [*440.0 Hz*] or [*'a3' pitch*], [*Amplitude value: 0.7071*] or [*-3 dB*]. The representation and interchange formats should support all manner of weird mixed-mode music magnitude expressions (e.g., [(*'c4' pitch*) + (*78 cents*) + (*12 Hz*)]), with "reasonable" assumptions as to the semantics of the operation (coerce to Hz or cents?). Applications for this representation should support interactive editors for music magnitude objects that support the manipulation of the basic hierarchy described below, as well as its extension via "light-weight" programming.

The Figure on the next page shows the class hierarchy of the model classes—those used for the species (i.e., representation)—on the left side, and the partial hierarchy of the concrete (implementation) classes on the right. The class inheritance hierarchy is denoted by the order and indentation of the list. The lines indicate the species relationships of several of the common music magnitudes. The examples below demonstrate the verbose (class message) and terse (value + post-operator) forms of music magnitude description. Note that comments are delineated by double quotes (or curly braces) in SmOKe.

Music Magnitude Examples

```
(Duration value: 1/16) asMS                    "Same as [1/16 beat]; answers 62."
(Pitch value: 36) asHertz                      "Same as [36 pitch]; answers 261.623."
(Amplitude value: 'ff') asMidi                 "Or [#ff ampl]; answers 106."
('f#4' pitch), ('pp' dynamic)                  "Terse examples; value + post-op."
(261 Hz), (1/4 beat), (-38 dB), (56 velocity), (2381 msec), (0@0 position)
```

6: SmOKe EVENT OBJECTS

The simplest view of events is as Lisp-esque property lists, dictionaries of property names and values, the relevance and interpretation of whom is left up to others (voices and applications). Events need not be thought of as mapping one-to-one to "notes," though they should be able to faithfully represent note-level objects. There may be one-to-many or many-to-one relationships between events and "notes." Events may have arbitrary properties, some of whom will be common to most musical note-level events (such as duration, pitch or loudness), while others may be used more rarely or only for non-musical events.

Events' properties can be accessed as keyed dictionary items (i.e., events can be treated as record data structures), or as direct behaviors (i.e., events can be thought of as purely programmatic). One can set an event to be "blue" (for example), by saying [*anEvent at: #color put: #blue "dictionary-style accessing"*] or more simply [*anEvent color: #blue "behavioral accessing"*] (whereby *#string* means *unique symbol whose name is string*). Events can be linked together by having properties that are associations to other events or event lists, (as in [*anEvent #soundsLike: otherEvent*]), enabling the creation of annotated hypermedia networks of events. Event properties can also be active blocks or procedures (in cases where the system supports compilation at run-time as in Smalltalk-80 or Lisp), blurring the differentiation between events and "active agents." Events are created either by messages sent to the class Event (which may be a macro or binding to another class), or more tersely, simply by the concatenation of music magnitudes using the message "," (comma for concatenation), as shown in the examples below. Applications should enable users to interactively edit the property lists of objects, and to browse event networks via their time or their links using flexible link description and filtering editors. Standard properties such as pitch, duration, position, amplitude, and voice are manipulated according to "standard" semantics by many applications.

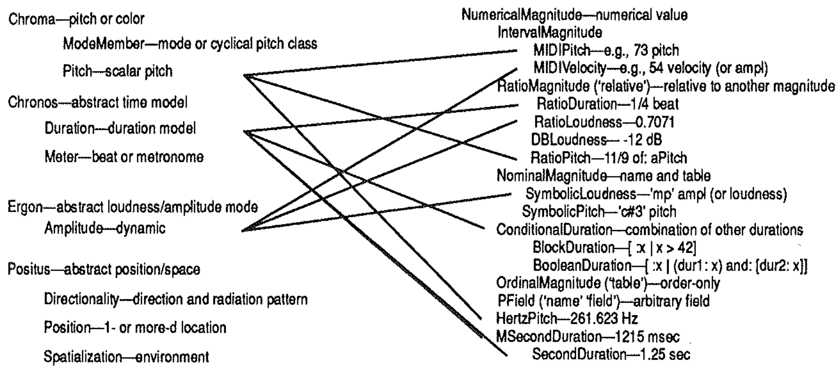

Figure 1: SmOKe Music Magnitude Model Abstractions and Implementation Classes

Event Examples

```
"Event creation examples--the verbose way (class messages)."
    [event := (Event newNamed: #flash) color: #white; place: #there]
    [(Event duration: (Duration value: 1/2) pitch: (Pitch value: #c2)
            loudness: (Loudness value: #mf)) playOn: aVoice]
"Create three events with mixed properties--the terse way"
    [(440 Hz), (1/4 beat), (-12 dB), Voice default]              "abstract props."
    [38 key, 280 ticks, 56 vel, (#voice -> 4)]                   "MIDI-style props."
    [(#c4 pitch, 0.21 sec, 0.37 ampl, (Voice named: #oboe)]      "note-list style"
"Create a named link between two events."
    [event1 isLouderThan: event2]
```

7: EVENT LISTS

Events are grouped into collections—event lists—where a list is composed of associations between start times (durations starting at the start time of the event list) and events or sub-lists (nested to any depth). Schematically, this looks like: (*EventList* = *(dur1 => event1), (dur2 => event2), ...*) where (x => y) means *association with key x and value y*. Event lists can also have their own properties, and can map these onto their events eagerly (at definition time) or lazily (at "performance" time); they have all 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: aDuration*], along with the corresponding event removal messages, can be used for manipulating event lists in the static representation or in applications. If the key of the argument to the *add:* message is a number (rather than a duration), it is assumed to be the value of a 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 contexts/closures. The behaviors for applying functions (see below) to the components of event lists can look applicative (e.g., [*anEventList apply: aFunction to: aPropertyName*]), or one can use event modifier objects to have a stateful 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 *#usedToBe* or *#viaScript11b5i4*), is defined by applications and voices.

A named event list is created (and stored) in the first example below, and two event associations are added to it, one starting at 0 (seconds by default), and the second at 1 sec. Note that the two events can have different types of properties, and the handy

instance creation messages such as [*dur: durationValue pitch: pitchValue amp: amplVaule*]. The second example is the terse format of event list declaration using the behavior of (*duration => event*) associations such that [*(aMagnitude) => (anImmediateDictionary)*] returns the association [*(duration with value aMagnitude) => (Event with given property dictionary)*]. One can use dictionary-style shorthand with event associations to create event lists, as in the very terse way of creating an anonymous (non-persistent) list with two events in the second example. The third example shows the first few notes from the c-minor fugue from *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

```
"Event Lists the verbose way"
    [(EventList newNamed: #test1) add: (0 => (Event dur: 1/4 pitch: 'c3' ampl: 'mf');
                                  add: (1 => ((Event new) dur: 6.0 ampl: 0.3772 sound: #s73bw))]
"Terse Lists--concatenation of events or (dur => event) associations."
    [(440 Hz, (1/1 beat), 44.7 dB), (1 => ((1.396 sec, 0.714 ampl) sound: #s73bw; phoneme: #xu))]
"c-minor fugue theme."
    "start time         duration  pitch          voice"
    (  (0.5 beat => ((1/4 beat), ('c3' pitch), (voice: 'harpsichord'))),
                    ((1/4 beat), ('b2' pitch)),   ((1/2 beat), ('c3' pitch)),
                    ((1/2 beat), ('g2' pitch)),   ((1/2 beat), ('a-flat2' pitch)) )
```

8: EVENT GENERATORS AND MODIFIERS

The *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 chords, ostinati, or compositional algorithms. Each event generator subclass knows how it is described—e.g., a chord with a root and an inversion, or an ostinato with an event list and repeat rate—and can perform itself once or repeatedly, acting like a function, a control structure, or a process, as appropriate. EventModifier objects hold onto a function and a property name; they can be told to apply their functions to any property of an event list lazily or eagerly. Event generators and modifiers are described elsewhere.

9: FUNCTIONS, PROBABILITY DISTRIBUTIONS AND SOUNDS

SmOKe also defines functions of one or more variables, several types of discrete or continuous probability distributions, and granular and sampled sounds. The description of these facilities is, however, outside the scope of this paper, and the reader is referred to (Smallmusic 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 instrument or algorithm. A voice object is a "property-to-parameter mapper" that knows about one or more output or input formats for SmOKe data (e.g., MIDI, note list files, or DSP commands). A *StructureAccessor* is an object that acts as a translator or *protocol convertor*. An example might be an accessor that responds to the typical messages of a tree node or member of a hierarchy (e.g., What's your name? Do you have any children/sub-nodes? Who are they? Add this child to them.) and that knows how to apply that language to navigate through a hierarchical event list (by querying the event list's hierarchy). SmOKe supports the description of voices and structure accessors in scores so that performance information or alternative interfaces can be embedded. The goal is to be able to annotate a score with possibly complex real-time control objects that manipulate its structure or interpretation. Voices and event interpretation are described in (Pope 1992).

12: SmOKe SCORE EXAMPLE

In SmOKe scores, sections with declarations of variables, naming of event lists, event definition, functions and event modifiers, and annotation, can be freely mixed. Note that one tries to avoid actually typing SmOKe at all anyway, leaving that to interactive graphical editors, algorithmic generation or manipulation programs, or read/write interfaces to other media, such as MIDI. The example below shows the components of a SmOKe score for a composition with several sections declared in different styles. Variable name declarations are placed between vertical bars.

```
"Declarations of variable names and top-level event list."
    | piece section1 section2 |   "name declarations--optional but advised."
    piece := EventList newNamed: #piece.
"Section 1--verbose, add events using add:at: message."
    section1 := EventList newNamed: #section1.
    section1 add: (...first event (may have many properties)...) at: 0.
    section1 add: (...second event...) at: 0. "starts with a chord."
        "...section 1 events, in parallel or sequentially..."
```

```
"section 2--terse, add event assoc. using ',' concatenation operator."
   section2 := ((0 beat) => (...event1...)), ((1/4 beat) => (event2)),
            "...section 2 events...",
      ((2109/4 beats) => (event3308)).
"Event list composition (may be placed anywhere)"
   piece add: section1; add: section2.          "add the sections in sequence."
   piece add: (Event duration: (4/1 beat)).     "add one measure of rest after section 2."
"Add a section from data arrays."
   piece add: (EventListwithProperties: #(duration: loudness: pitch:)
         values: (Array with:#(250 270 230 120 260 260 ... )"duration"
         values: (Array with:#('mp')                        "loudness"
         values: (Array with:#('c3' 'd' 'e' 'g' ... )).     "pitch"
"Add an event with the given samples (you want low-level? we got low-level!)"
   piece add: (Event rate: 44100 channels: 1 samples: #(0 121 184 327 441 ... )).
"Declare global (named) event modifiers, functions, etc."
   (Rubato newNamed: #tempo) function: (...tempo spline function...) property: #startTime.
      piece tempo: (Rubato named: #tempo).
"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 versions, ranging from very readable to very terse, and covering a wide range of signal, event, 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 Gomsi, were part of the team that discussed the design of SmOKe, and commented on its design documents (Smallmusic 1992).

REFERENCES

ANSI 1992 *Journal of Technical Development, ANSI Working Group X3V1.8MSD-7* (now ISOIEC DIS 10744).

R. B. Dannenberg, L. Dyer, G. E. Garnett, S. T. Pope, and C. Roads, "Position Papers for a Panel on Music Representation," *Proc. of the ICMC*, San Francisco: ICMA, 1989.

A. Goldberg and D. Robson, *Smalltalk-80: The Language*, (revised and updated from 1983 edition). Menlo Park: Addison-Wesley, 1989.

MusRep 1986, R. Dannenberg, J. Maloney, et al., "Network electronic discussion on music representation" USENET.

MusRep 1990, G. Diener, L. Dyer, G. E. Garnett, D. Oppenheim, S. T. Pope et al., "Notes of meetings on music representation at CCRMA", Fall, 1990.

Newcomb, N. A. Kipp, and V. T. Newcomb. "The HyTime Hypermedia/Time-based Document Structuring Language," *Communications of the ACM*, vol. 34, no. 11, pp. 67-83

S. T. Pope, "Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers," *Computer Music Journal* 16:3. Fall, 1992.

Smallmusic 1991. G. E. Garnett, J. Gomsi, C. Latta, D. Oppenheim, S. T. Pope et al., Smallmusic discussion group notes, Credo 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 ccrma-ftp.Stanford.edu in the directory pub/st80 (see the README file there).

Análise Musical, Educação