The Sm0Ke music representation, description language, and intercante change format

Stephen Travis Pope
The Nematode Group, Computer Music Journal, CCRMA
P. O. Box 50832, Palo Alto, California 94306 USA.
Electronic Mail: stp@CCRMA-Stanford.edu

ABSTRACT

The Smallmusic Object Kernel (Sm0Ke) 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 Sm0Ke 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 defines Sm0Ke's 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 1992b), describe the Sm0Ke language, and the M0RE environment within which it has been implemented, in more detail.

1. INTRODUCTION

The desire has been voiced (ANSI 1992; Danneskiold 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 lists hierarchical 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 intercante change formats, and description languages based on the formal definition of Sm0Ke (Smallmusic 1992). The naming conventions and the code description examples use the Smalltalk-80 programming language (Goldberg and Robson 1983), 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.i" in the directory "anonymous@ccrma-fp.stanford.edu/pub/small0ke") introduces the language's concepts and syntax to facilitate the reading of the code examples in the text. In this document, Sm0Ke examples are written between square brackets in small- and mixed-case typeface.

2. EXECUTIVE SUMMARY

The Sm0Ke 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 microseconds to tens of seconds). 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., as event list can have another event list as one of its events, etc.). They can also map the properties onto their component events. This allows any event can be "shared" by being more than one event-list at different relative start times and with different properties mapped onto it. Events and event lists are "executed" 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 breakpoints 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.
A linear 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 (spelling out) or procedural (listing of music data definitions) in nature. A text file can be freely edited as a data structure, but one can compile it with the Smalltalk-80 compiler to “instance” the objects (rather than needing a special formatted reading function). The post-fix expression format taken from Smalltalk-80 (receiver:object keyword: optional:argument) is easily parseable in C++, Lisp, Forth, and other languages.

3.2 Language Requirements

The basic representation itself is language-independent, but assumes that several immediate types (such as numbers, strings, and symbols) are representable as ASCII/ISO character strings in the host language. 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 metalinguage 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)).

3.3 Naming and Persistence

The names of abstract classes are known and are treated as special globals. The names of abstract classes are used whenever possible, and instances of concrete subclasses are returned, as in (Pitch value: 'c3' or '106 pitch') both returning a SymbolicPitch instance (strings are written between single-quotes in SmoKe [and Smalltalk]). The central classes support “persistence through naming” whereby any object that is explicitly named gets stored in a global dictionary under that name until released.

3.4 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 zero 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 “length” of events; they can be used at the level of notes or envelopes, 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 reassemble a note list file or a DSP program. It is structured as executable Smalltalk-80 expressions, and can define one or more “new” classes. There is no “section” on the procedural side, that is, everything that is 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.

4. 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-ordcred 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 'C8'], 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 (114 beat + 80 more). 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 ['C7 pitch], (Amplitude value: 0.727) or [.3 dB]). The representation and interchange formats should support all manner of weird mixed-mode music magnitude expressions (e.g., ([4d pitch] + [78 cents] + [12 Hz]), with “reasonable” assumptions as to the semantics of the operators (e.g., Hz to Hz). Applications for this representation should support interactive editors for music magnitude objects that support the manipulation of the basic hierarchy described below, as in an extension via “light-weight” programming. The examples below demonstrate the verbose (class message and terse (value + post-op-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 | asw3 | "Same as | 1/16 beat" | answers | 62. |
| Pitch value: | 36 | aslote | "Same as | 36 pitch" | answers | 261,623." |
| Amplitude value: | 'E' | (mw4i) | "Or | [E] amf | answers | 106. |

(261 Hz), (1/4 beat), (-38 dB), (56 velocity), (238 memo), (080 position)
EVENT OBJECTS

The simplest view of events is as Lisp-esque property lists, dictionaries of property names and values, the relevance and interpretation of which 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 non-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-related 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 key-value dictionary items (i.e., events can be treated as record data structures), or as direct behaviors (i.e., events can be thought of as programmatic). One can set an event to be "blue" (for example), by saying (set-event at: #now put: #true value: #"colored" style: #"colored" accessor: #"color") (whereby accessor 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 [set-event @roundTo: 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 event messages using the message "=" (comma for concatenation), as shown in the examples below. Common properties such as pitch, duration, position, amplitude, and voice are manipulated according to "standard" semantics by many applications.

Event Examples

"Event creation examples—the verbouse way (class messages)."

[Event r := (Event newNamed: #flash) color: #white place: #there)
{Event duration: #l/2 pitch: #2 loudness: #nf playOn: #Voice})

"Create three events with mixed properties—the terse way"

[(#440 Hz. 1/4 beat. -12 dB, Voice default)
{38 key. 280 ticks, 56 vel. (Voice => 4)}
{(#IV pitch. 0.2) sec. 0.37 ampl. (Voice named: #p001})

Event Lists

Events are grouped into collections—event lists—where a list is composed of associations between start times (during starting at the start time of the event list) and events or sub-lists (nested to any depth). Schematically, this looks like: (EventList = (dur => event1, (dur2 => event2), ...) where (x => y) means association with key x and value y. Events list can also have their own properties, and can map these onto their event tagged (by definition time) or lazily (as "performances" time); they have all the property and link behavior, and special behaviors for mapping with voices and event modifiers.

The messages [EventList add: anAssociation] and [EventList add: anEventEventList at: anIndex], along with the corresponding event removal messages, can be used for manipulating event lists in the static representation or in applications. Event lists also respond to Smalltalk-80 collection-style control structure messages such as [EventList copy: aSelection-Block] or [EventList 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., [EventList apply: aFunction to: allPropertiesNames]), or one can use event modifier objects to have a stylesheet representation of the mapping. Applications will use event list hierarchies for browsing and annotation as well as for scores following and performance control. The use of standard link types for such applications is version control (with such link types as RootedTo or hvaScript18514d), is defined.

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: ampValue]. The second example is the time format of event list declaration using the behavior of (duration => event) associations such that (#MAGNITUDE => ) (AnImmediate-Dictionary) returns the association [(duration with value #MAGNITUDE) => (Event with given property dictionary)]. One can use dictionary-like shorthand with event associations to create event lists, as in the very terse way of creating an anonymous (non-questions) list with two events in the second example. The third example shows the first few lines from Bach's Well-Tempered Clavier 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 verbouse way"

[EventList newNamed: #test1 add: [0 => (Event dur: l/4 pitch: #c3 amp: #mf)].
add: [1 => ((Event new dur: 4.0 ampl: 0.3772 sound: #b73bw)])

"Term Lists—concatenation of events or (dur => event) associations."

[(#440 Hz. 1/2 beat. 4.47 dB.), [l => ([0.396 sec. 0.714 ampl] sound: #b73bw phoneme: #v011])

108
c-minor fugue theme.

f (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))

7: EVENT GENERATORS AND MODIFIERS

The EventGenerator and EventModifier packages provide for music description and performance using generic or composition-specific, fixed-length objects. Event generators are used to represent the common structures of the musical vocabulary such as chords, ostinato, or compositional algorithms. Each event generator subschema knows how it is described—e.g., a chord with a root and 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 used to apply their functions to any property of an event list lastly or earlier.

8: 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 SmoKe (1992).

9: 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-parameters" 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 converter. An example accessor might respond to the typical messages of a tree node or member of a hierarchy and know 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).

10: 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 forms, 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.

11: ACKNOWLEDGEMENTS

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

REFERENCES


Smallmusic 1992. G. E. Carus, J. Gomni, C. Laus, D. Oppenheim, S. T. Pope et al., Smallmusic discussion group notes, Credo documents (from which this document was derived), and MODE User Primitive specification available from Smallmusic/DEC Berkeley, ed at email or via anonymous Internet ftp from the server ccrma-fp.stanford.edu in the directory pub/sm/80 (see the README file there).