	added to explanations, fleshed out motivations, 1/3/92
0.4b	added section numbers, notes, general cleaning, 1/10/92
0.5	many changes after CL’s first and second comments:
	new text about durations and event lists;
	new Ergon/Positus abstraction hierarchy;
	added “formal” (yuk yuk) specs. and d. struct. defs.;
	many new notes/comments, language req. descr. reworked;
	added new section 5a, shifting section 5 paragraph
	numbers by 1 subsection (or by “0.a”). 1/16/92
0.6	reordered, separating out discussion into Appendix 2/4/92

==

B: Example Score: First Three Measures of “Eine Kleine Nachtmusik”

(any other suggestions?)

Version 1: Verbose and Linear

(to come)

Version 2: Terse and Structured (and annotated)

(to come)

<<END>>

messages), rather than by their state. Because the SmOKe music representation is embedded in an object-oriented language, score files can be seen as static data declarations, or as active programs that use the behaviors of the classes in the system that reads or writes the files.

8: References

A complete bibliography for this project can be found in (Pope 1986, 1991a, 1991b) and includes the influential music representation work of (in no particular order): Max Mathews, Bill Buxton et al., Leland Smith, C. Fry, Mark Lentczner, Roger Dannenberg, Glen Diener, Lounette Dyer, Mira Balaban, Bernard Mont-Reynaud, D. Gareth Loy, Curtis Abbott, Guy Garnett, Curtis Roads and others. Only the reference list for this document is included below; I've tried to keep it small and relevant.

(Let's get a group bibliography together for this project, who has contributions?) (I have a large 1986 bibliography (from [Pope 1986]) of AI, UIMS, and CM software, anybody want to bring it up to date?)

Gibson, E. 1990. "Objects, Born and Bred." Byte Magazine. May?, 1990.

Goldberg, A., D. J. Leibs, and S. T. Pope 1988. "Sleeper/Ajax HyperMedia Development Environment Project Documentation." Internal document. Mountain View: ParcPlace Systems Inc.

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

Pope, S. T. 1982. "An Introduction to msh: The Music Shell." Proceedings of the 1982 ICMC. San Francisco: ICMA.

Pope, S. T. 1986. "The Representation of Musical Structure and Knowledge." Proceedings of the 1986 ICMC. San Francisco: ICMA.

Pope, S. T. 1988. "The RecordEditor DataBase Management System." ParcPlace public domain software and documentation available from archives of comp.lang.smalltalk or from parcbench@ParcPlace.com.

Pope, S. T. 1989a. "Machine Tongues XI: Object-Oriented Software Design." Computer Music Journal 13:2, and in S. T. Pope, ed. "The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology." Cambridge: MIT Press.

Pope, S. T. 1989b. "Modeling Middle-Level Musical Structures Using EventGenerators." Proceedings of the 1989 ICMC. San Francisco: ICMA.

Pope, S. T. 1991a. "A Description of the MODE: The Musical Object Development Environment" in "The Well-Tempered Object."

Pope, S. T. 1991b. "Interim DynaPiano" paper in "Proceedings of the IX Italian CIM Colloquium," Genova, 11.1991, Venice: AIMI, also available via ftp from CCRMA.Stanford.edu in the compressed PostScript file IDP.ps.Z or compressed FrameMaker document IDP.frame.Z.

Pope, S. T. 1992. "Object-Oriented Design Issues in the MODE Musical Object Development Environment." submitted to JOOP, 1992, also available via ftp from CCRMA.Stanford.edu in the file MODE.oop.t.

Pope, S. T., N. E. Harter, and K. Pier 1989. "A Navigator for UNIX." ACM/SIGCHI 1989 Video, available from ACM/SIGGRAPH.

Smallmusic 1991. Smallmusic discussion group notes, Credo 1 and 2 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.Stanford.edu in the directory pub/st80 (see the Readme file there).

==

Appendices

A: Version History

0.0	Smallmusic Credo 1 and 2 Documents, stp, 11/91
0.3	merged into OOMR.t Document, 12/31/91
	distribution to Smallmusic and MusicResearchDigest
0.4	stripped out st80 intro, class hierarchies, etc.

[6f] “OOMR and Standardization”

The current format of this document, and the language of the specification, preclude the treatment of this as a “ Standards”(upper-case ‘s’) proposal of any kind. A more formal list of technical requirements and derivation of an abstract representation based on the principles mentioned above is a task still to be done. What relationship this effort has with the ANSI MIPS SMDL standard is undefined, though the ability to import/export SMDL to/from SmOKe applications will be relevant should SMDL emerge.

==

7: Glossary

The abstract class names and concepts used in the SmOKe music description are defined informally below before they are more fully described in the following sections.

MusicMagnitude: an object with a value (i.e., a numerical or symbolic object), and some notion of what it represents (e.g., a pitch or loudness), with special behaviors for operating with other objects that model the same thing. An example MM would be (440.0 Hz), which will have some floating-point-number-like and some pitch-like behaviors.

Association: a binding between a key and a value. The key is often a symbol, which means it can be rapidly searched for and compared to. Associations are written as (key -> value) in Smalltalk-80.

Event: a very generic list of values for a list of properties at a given time, like a LISP prop-list or a Smalltalk dictionary (a set of associations between symbolic keys and values). Events may have special time- or music-related properties (e.g., duration or pitch), or may have properties that are symbolic links to other events (e.g., [#soundsLike someOtherEvent]). Events may be used at the level of “ notes,”or they may be very much “ lower”level and represent parts of a sonic event such as envelope components, grains, or DSP effects. The interpretation of an event’s properties is arbitrary and it is left to applications to map “ abstract”event properties onto “ concrete”parameters for I/O or presentation. An event could, for example, have several types of pitch properties, or could have its pitch coerced into various formats by mapping objects. Applications are free to attach new properties to events, but must be careful (through the common software engineering techniques), to avoid collisions in their choice of property names.

EventList: a time-ordered collection of events with its own property list that it can map onto its components (now or later). Event lists can contain sub-event-lists as elements (i.e., they can be nested to any depth), and can be expanded or flattened on command. An event list can be thought of an just an event that holds onto a sorted list of associations between start times and other events.

Link: a property of an event (or event list) that is an association with a symbol for a key and an event or event list for a value, e.g., the event list melody1 having the property [#isAnswerTo: melody0]. Links may be used by applications to provide multi-dimensional, non-sequential access to event lists in a hypermedia interaction style.

Voice: a mapper or performer that takes events and event lists and maps their structure and magnitudes and/or schedules them to some I/O port in or slightly before real-time.

EventGenerator: an object that can generate an event list based on an algorithmic or numerical description, as in a chord (which generates notes when given a root and inversion). It is undefined whether an event generator stores its events (as in the notes of the chord), or its parameters and algorithm (as in the chord creation data and harmony rules).

Function: a map of one or more free variables that can be described in a number of ways and stored for application as a look-up table, map or sample buffer.

EventModifier: an object that can map a function of one or more variables onto one or more properties of an event or event list, as in a crescendo that can map a given spline function onto an event list’s components’ loudness properties.

Sound: a sound modeled as a sample or grain stream, or a function or process of a collection of sample or grain streams.

Behavior: the action taken by an object in response to receiving a message. In object-oriented programming, objects are described primarily by their behaviors (the methods they have for responding to

[6.5l] Structure Accessors

Interfaces to applications that want to store their own information about an event or event list (e.g., the layout of a score), do so in structure accessors--interface objects that can have properties related to a " subject"event or event list, and that respond to a specified message-passing interface (their " perspective"). Perspectives pertain to different domains, such as graphical presentation or analytical annotation. By making smart structure accessors for event lists, one can build applications that add information to the lists in a generic way (see the Navigator and Sleeper literature).

Although structure accessors are application-specific, they are relevant to the representation because applications are free to attach any number of them to musical structures over their lifetimes, and they can contain any of the basic types used here, in addition to arbitrary implementation-language-specific text descriptions. Examples such as graphical accessors for heavily marked-up scores that include complex X or PostScript data structures, or event list structures where the volume and complexity of the hypermedia link data far exceeds that of the actual musical properties, are expected to be typical.

[6] Discussion

[6a] (moved)

[6b] "SmOKe and Navigator Representations"

In its purest form, this architecture of magnitudes, events, event lists, voices/interpretations, and accessors is very powerful and music-inspecific or application-generic. One can use events as " nodes"in network- or document-like systems, or as " records"in a database sense. It is very straightforward to coerce common data structures (e.g., display lists) into event list-like hierarchies. Very similar architectures have in fact been used for such related areas as structured graphical user interfaces (e.g., Navigator [Pope, Harter, and Pier 1989]), on-line database management (e.g., RecordEditor [Pope 1988]), hypermedia active document interaction and application frameworks (e.g., Sleeper/Ajax [Goldberg, Leibs, and Pope, 1989]), or Petri net editing and animation tools (e.g., DoubleTalk [Pope 1986]).

[6c] (moved)

[6d] "OO Methodology"

The object-oriented nature of this design is evident in all of the basic models introduced above. The representation/implementation duality of music magnitudes is modeled as intrinsic and obvious, and implementations in fully object-oriented languages are possible in a very few kBytes of source code (!!). The nature of events and event lists (see section 6b above), and the construction of event list heterarchies with distributed properties, temporal conditions, and behaviors, is also derived from iterative OO analysis of the structures used in the types of documents for which these systems are destined. The event generators are an excellent example of a hierarchy with a very high degree of behavioral sharing and easy extensibility. The abstraction of voices as drivers and the flexible description of performance situations it makes possible, is based on a behavioral abstraction interface of " performance clients,"related to the client-server model. Lastly, using structure accessors to allow applications to create and store their own information along with " user"data is a technique that should allow easy extension of the notation as more interfaces (that cannot be addressed in the form of new voice drivers) become standardized.

[6e] "Implementations of the SmOKe Representation"

Any significant changes or additions to this document should be reflected in one or more Smalltalk-80 implementations. In general, several implementations of the representation (including working demonstration versions in Smalltalk, LISP, C, C++, and Forth) should be assumed. The MODE V1.0 implementation is expected to be re-sync'ed to this description sometime in Spring 1992, and that implementation will (as always) be on CCRMA.Stanford.edu.

The messages for creating events using immediate dictionary formats should hide the implementation of event associations and/or event dictionaries. Depending on the architecture of the implementation, a special event dictionary class (in addition to the event association class), may be necessary.

[6.5f2] EventList Notes

The two ramifications of this architecture are (1) that events can be shared within event lists, and can have different properties mapped onto them at different times; and (2, related to 1), that having lazy and/or dynamic properties distributed within event lists enables one to blur the state/behavior or declarative/ procedural distinctions in thinking about event lists quite a bit.

The semantics and grain-sizes of events and event lists are not limited; events can be " notes,"or they can be much finer- or coarser-grained, depending on the requirements of the user. Event lists can be used sequentially or in parallel to model many types of different voice/track/part organizations. The distribution of information between events and voices is also arbitrary, and the instrument/note or object/ event boundary is not predetermined.

The " standard"link types and their semantics will depend on the applications and annotation styles that a particular user prefers. A hypermedia version handler for sketches and scores would, for example, add links named #usedToBe: among different versions of an event or event list during development, and would use properties for version numbers and change comments. A Petri net composition tool would use typed links whose symbolic keys are the names of predicates or transitions in a network, and would add properties for the capacities or predicates or the conditions and actions of transitions. More common applications, such as score editors, can use the primary event list hierarchy or their own link heterarchies for representing their notions of " tracks,"" parts,"or other abstractions.

Note that the semantics of sorting durations has been ignored, other than to say that if all durations have numerical magnitudes, it can probably be performed. Using relative or conditional durations can change the order of events in static event lists, and the representation *and* the interchange format cannot assume that an event lists events can be delivered in order. In many cases (even without the use of conditional), event lists will be difficult or impossible to measure or sort. The desired behavior of applications in these cases should be described.

It is interesting that the terse, immediate style of declaration makes most class names unnecessary. There are straightforward ways of handling persistency and global naming to reduce this even further (an idea I like quite a bit).

[6.5i2] Voice Notes

There are several points of discussion related to the design of voices. The first is related to naming: voice vs. performer or interpreter or render. The second is the voice/scheduler vs. process-per-active-event architecture debate.

The detailed design issue of voices on ports on devices is also unresolved. The old HSTK design (see the example above), was flexible, but inelegant and difficult to extend. A collapsed design has yet to be tested, but offers increased simplicity for smaller systems.

[6.5k2] Sound Notes

The exact model used for describing sound synthesis and processing has yet to be derived. An mshell-like (Pope, 1982) DSP interpreter (" a programmable desktop calculator for dsp"), would be of great use. Other formats would include a music5-like unit generator model is also possible, and has been started in an OO style. One should, of course, also allow direct specification of samples, so that Dick Moore can use SmOKe.

The issues of whether or not there are special sound file objects (as separate from sampled sound objects), is thought to be a detail, and should be invisible to the interchange format.

The naming would be easier if I didn't want to keep the name Sound abstract (to allow granular, PILE-like, or other [non-sample-stream-like] descriptions). Can anyone come up with another Greek word for it, so that Sound can mean SampledSound? (Is this a good idea?)

address block contexts, since they should at least be portable (and transmittable in real-time!) between same-language applications.

Whether the use of global variables is preferable to a macro preprocessing facility is unclear to me. (Why not define CPP as standard, and provide #define and #include?)

The mixing of data in properties of events (leaves) or event lists (branches) is arbitrary, making scores be structured as DAGs or like simple trees where all the " data"is in the leaves.

[6.5b2] Music Magnitude Notes

Note that absolute time is not mentioned above as I believe it to be a very bad idea in music representations. I assume below only the existence of a model of relative duration, which is a standard abstraction and a set of concrete music magnitude implementation classes.

The notion of " conditional durations"is desirable for complex scores; these are expressed in terms of a block closure that is evaluated repeatedly (with no assumption as to exactly how often) until it returns " true."By default, this block takes two arguments: the event it is a property of, and some external " timer" of the user's [or scheduler's] choice) (e.g., [:ev :t | t value > 40]).

Note also that no model is proposed for timbre, this being left up to voice objects to interpret the properties of events for use in various I/O media. Voices are assumed by be symbolically named (stored persistently by the Voice class) so that users can generate scores that are portable among hardware systems through their use of common voice names (the set of which is not specified here). It is, of course, possible for events to be given properties for such synthesis-model-specific parameters as modulation indices or envelopes, or relative formant amplitudes when scoring for known instrument (such as software sound synthesis/processing).

It is still unclear to me whether I'll get away with the merging of the pitch and interval models; it seems to work in this framework for what I can think of, and even complex relative models are possible, but I'm sure somebody who knows more than me will stand up and tell me it can't be done (Loon?, Roger?, Bernard?).

The naming of pitches and mode members is also unsettled; the assumption is that #c4 (symbol with string value 'c4') should mean " the pitch c in the 4th octave."(American numbering, c1 in Europe). The two suggestions are to use #c to mean #c0, or " mode/pitch-class C."(In the first case, the mode C would be described as #C.) This should not be frozen until a mode/pitch class model is proposed.

The use of the hash-mark (" #'), for symbols in Smalltalk-80 creates obvious unfortunate interference with the musical sharp-sign. We could re-define SmOKe's immediate symbol identifier (e.g., to be _ or @), or require that pitch names with accidentals be spelled-out, (e.g., #gSharp3, #aFlat6). I'd prefer the latter, but it will be unpopular with those who still write music with pitched notes...

[6.5c2] AEvent Notes

Note the terseness of the second group of examples above. This format is possible due to the fact that music magnitudes (and voices) transmogrify into property associations when sent the concatenation message " ,"--i.e., evaluating the expression [1 beat, ...] creates the association (#duration -> 1 beat) and sends *it* " ,". This means there is often no need to assign properties--e.g., adding a property that is a Pitch-species object assigns it to the #pitch property. The terseness is increased because of Smalltalk's " immediate dictionary format,"meaning that [association, association] creates a dictionary containing the two associations, and [dictionary, association] adds the argument to the receiver dictionary as a new association. [ADM]Events (acting like dictionaries) also provide this behavior.

See the note below on the " grain size"of events for a large disclaimer.

It is an implementation detail whether or not all events have a duration slot or other default slots; the introduction of DEvents and MEvents below can be ignored at the abstract level.

The messages for setting and querying an event's properties should hide the implementation of the property dictionary. The behavioral accessing style should be maintained wherever possible, and translating front-ends (that turn everything into dictionary-style accessing messages), should be constructed for implementation readers that cannot cope with it.

performance processes by providing tools for dealing with abstract descriptions of musical structures and allow flexible refinement and specification of large weakly-structured data sets.

Another aspect of the SmOKe representation is the desire for a music description language designed using a strict object-oriented analysis and design methodology (object-behavior analysis, [Gibson 1990]), for the modeling of the domain of musical knowledge. This means discovering, and building behavioral models of, the basic abstractions and generalizations that form the foundations of music theory and composition. This is described in more detail in (Pope 1989a, 1992).

[4] Language Syntax and Features

Using Smalltalk-80, rather than a pure specification language such as BNF or ANNA, is thought to be an excusable detail, as it can be reduced to the basic types and constructs that are enumerated below, which are easily defined in a more formal language. A formal description of the semantics of the SmOKe objects' behaviors will evolve along with this document.

The only syntactical additions to Smalltalk-80 is the addition of a second comment string (perhaps the C-style /* ... */), so that sections containing " ..."comments can be commented-out. It might also be useful to include a one-line comment string (i.e., comment-out everything up to <CR>), such as Lisp's " ;"or ADA's " --"(given the choice, I'd choose " --").

[6.4b] Language Requirements

Implementation languages must provide the ability to compose structured data types consisting of the immediate types, arrays or lists, and pointers to other composite types.

As variable name declarations are optional in SmOKe, readers must be able to auto-declare variable names (or cope with undeclared ones).

The representation also has an easily-described in-memory structure in the Smalltalk-80 object space; data structures can be linearized in very compact binary format using the BOSS (or other) Smalltalk-80 object storage, persistence, and sharing tool.

The only special considerations in the linearization of SmOKe in Smalltalk-80 are the use of the compiler's file-in " chunk"format, and the maximum length of message cascades supported by the compiler, though these are both details beyond the scope of the present document, and should be handled by automatic " chunkifiers"in implemented representation scanners.

[6.4d] Score Format

SmOKe scores, when stored in ASCII/ISO data files, may use arbitrary indentation formats; the use of the Smalltalk-80 standard formatter (or something similar), is optional and recommended, however, see (Goldberg and Robson 1989). The use of comments (enclosed in double-quotes " ...") is advised. They are ignored by readers and may be discarded by interchange format transmitters as well. Applications can assume score files to be named with the suffix [" .sco"" .score"" .sm"" .SmOKe"" .el"] (pick one).

[6.4d2] Notes

The use of global names for SmOKe structures " finesses"the issue of scope and extent somewhat (or rather blurs it). It is unclear whether to allow documents to refer to existing event lists (e.g., to start with messages like (EventList named: #sketch2) add: (newEvent) without declaring (EventList newNamed: #sketch2) (which is defined as being the same as named: when the list already exists). the option would be to require declaration/definition of all lists used in any document, possibly via #include files.

Specifying a compact binary interchange format for SmOKe documents involves several steps (i think...). The first is to create a symbol map for the list of global symbols (class and message names); it would appear that any hashing mechanism will do for this. Step two, specifying the handling of immediates, is no easy task in an RPC-like environment, but we could import some format (SunRPC, BOSS, ISIS, etc.). The movement of symbols and OOP-like references is another difficult issue, and is not supported by many (structured-language-based) RPC systems. The next step is the design of the " out-of-band"information necessary to transmit SmOKe--the structure-related " punctuation marks."Lastly, we would need to

```
[aSound := (SampledSound named: #word1)]
[aSound2 := aSound from: 1024 to: (aSound size)]
[aSound2 rmsInto: (Array new: 512 withAll: 0.0)]
aSound3 := aSound2 + aSound."math is possible"
aSound3 := aSound3 / 2. "even mixed mode"

 (other sound behaviors such as from:to:, scaledBy:,
interpolatedFrom:to:into:, osc:freq:amp:, mixFrom:to:into:)

aSound4 := SampledSound newNamed: #stutter.
aSound4 duration: 4.0. "get empty samples."
aSound4 := aSound4 + aSound2.
aSound4 add: aSound at: 1.0."mix in the other word"
```

[5k3] Formal Specification of Sounds

Keywords:

Sound, SampledSound, StereoSampledSound, QuadSampledSound, Sampled16BitLinearSound, Sampled64BitFloatSound, etc.

Primitive Types Used:

All, Functions

Creation/Evaluation Operators:

Event behaviors
new, newNamed:, named:, newNamed:fromFile:, etc.
on:, on:from:to:, etc.
sampleAt:, sampleAt:put:, do:, collect:, inject:into:, etc.
+, -, *, /, add:at, etc.
rmsFrom:to:into:, maxFrom:to:, scaledFrom:to:by:into:,
fftFrom:to:window:into:, etc.

Data Structures:

SampledSounds are DEvents with pointers to sample data arrays.

==

6: Notes and Discussion

General notes and points of discussion have been moved here to minimize the length of the general SmOKe description. Additions are invited.

[6.1] Introduction

The Smallmusic Kernel representation is similar (though not currently identical) to that of the Musical Object Development Environment (MODE), a Smalltalk-80-based music software framework and tool kit (Pope 1991a). In this document, we will speak of the current proposal as SmOKe, ignoring the current MODE implementation (which will, of course, be brought into alignment with the version documented here at some future point).

The representation should be easily manipulable in any language that supports four basic features: (a) unique symbols, (b) untyped variables, (c) full polymorphism of message/function names, and (d) object identity. The immediate candidates are Smalltalk or LISP, although C++ and others are possible when compromises are made on points [b] and [c].

According to Smalltalk usage, global variable names (such as class names) are capitalized (e.g., Pitch), while local variable names, message names, and symbolic identifiers (in general) are not.

[6.2c] Related System Design Issues

The motivation for the use of this language in intelligent composer's assistants or Interim DynaPianos is documented in (Pope 1986, 1991b). The focus in these systems is supporting the composition and

Primitive Types Used:
All, Streams
Creation/Performance Operators:
new, newNamed:, named:
on: (a stream or device)
play:, close, release
Data Structures:
Voices consist of their class pointer and one to an I/O stream.

[5j] Functions and ProbabilityDistributions

The SmOKe representation also assumes the existence of a flexible notation for functions of one or more variables. The Magnitude hierarchy is extended by a simple class hierarchy for functions using linear, exponential, or spline interpolation between breakpoints, summation of sine components, sample spaces, or other description formats for creating and applying sampled or calculated tables. The MODE function hierarchy is described in (Pope 1991a).

A set of objects similar to the Smalltalk-80 system's probability distribution classes (e.g., Geometric, Binomial, SampleSpace, or Gaussian), are also desirable and should be specified based on (Goldberg and Robson 1989).

[5j2] Formal Specification of Functions and Probability Distributions

Keywords:
LineSegment, ExponentialSegment, SplineSegment, SumOfSines, etc.
SampleSpace, Gaussian, Uniform, etc.
Primitive Types Used:
All, Points
Creation/Evaluation Operators:
breakpoints:, weights:, coefficients:
valueAt:, valuesInto:
next (for P. Dists)
Data Structures:
Functions have class pointers and pointers to arrays of either breakpoints or data values (which may be interpreted as an ordered array or unordered sample space).

[5k] Sounds

Sampled sounds can be described using messages for reading and writing sample streams from files or live sources. Process of streams can be described using the basic sample manipulation messages supported by the sound object. Sounds are modelled as DEvents, and will often have the standard property sampleRate.

The representation supports different class names for different sample formats and numbers of channels, rather than having sound objects with flag fields for these properties. It is up to implementations to decide how to represent the various sampled sounds formats internally. The desired encoding formats are linear or Mu-law, with precisions ranging from 8- to 64-bit integer and floating-point numbers and 1, 2 or 4 channels. How much of the entire matrix of possible formats should be named and specified remains to be determined.

The abstract classes Sound, StereoSound, and QuadSound create instances of a system-dependent default class (preferably at least 16-bit linear 44.1 kHz), and should be used when no special format is required. Supporting default floating-point class names might also be worthwhile.

```
    "Sampled Sound Examples"
[((SampledSound newNamed: #word1) readFromFile: 'word1.snd')]
```

Creation/Application Operators:
 function:property:, apply:to:
 Class-specific creation messages
Data Structures:
 EventModifiers are simply records with a class pointer, a pointer to a function of 1 or more variables, and a symbolic property name.

[5i] Voices

Event and event list performance is handled by " voice"or " performer"objects that can be seen as " device drivers"in that they handle the mapping of abstract event properties onto concrete parameters for some specific performance medium. A scheduler can ask an event to play itself on a voice, which means that the voice creates an output packet for some output device or file. Schedulers can be told to wait and execute their lists in real time, or to perform as fast as possible, i. e., when dumping to a note list file or external scheduler. Some voices can also read input, either in real-time or from a stored file, to create and return event lists. A set of standard voices is provided for describing concrete I/O formats in a number of interchange languages (e.g., cmusic note lists or Adagio language scores), or real-time formats (e.g., MIDI synthesizer commands or samples sent to/from a DAC/ADC).

One creates a voice object and sends it an event or event list to " play,"whereby the semantics and side-effects of the operation depends entirely on the voice. Voice objects generally hold onto a smart stream onto a file or I/O driver, and do their best to map the properties of events for their media, losing as much precision and control as necessary given their medium (as determined by the [play: anEvent at: aDuration(StartTime)] behavior of the voice). Some voices operate at maximum speed (e.g., when reading or generating a notelist file), while others schedule their events in or slightly ahead of real-time (e.g., for MIDI or sampled sound output).

```
"Voice Usage Examples, Declaration and I/O"
              "Set up a new device driver for a DX7."
   aDX7 := MidiDevice on: (MidiPort newOn: 1).
              "Open a voice on it."
   anOboe := MidiVoice on: aDX7 channel: 6.
              "Define a new chord (EventList)."
   aChord := Chord majorTriadOn: 'C4' duration: 2.
              "Play it on the given voice."
   aChord playOn: anOboe.
              "Set its voice and play it."
   aChord voice: anOboe; play
              "Set up a new device driver for an FB01."
   anFB01 := MidiDevice onPort: (MidiPort newOn: 1).
              "Add another device to the voice."
   anOboe addVoiceOnDevice: anFB01 channel: 3.
   aChord play."Play the Chord again."

              "Create a named formatted file voice."
   CmusicVoice newNamed: #k11
         onStream: ('k11.2a.sc' asFilename writeStream)
   aCmusicFile := Voice named: #k11.
              "Play the chord on it."
   aChord playOn: aCmusicFile.
              "Close the File."
   aCmusicFile close.
```

[5i3] Formal Specification of Voices

Keywords:
 Voice, MidiVoice, CMusicVoice, AdagioVoice, etc.

```
      pitch: #( #(40 43 45) #(53 57 60))
      ampl: #(80 80 120)"static amplitude set"
      voice: #(1 3 5 7)"and voice set"
      density: 20)]
```

[5g2] Formal Specification of Event Generators

Keywords:
  Cluster, Cloud, Ostinato
  MajorChord, MinorChord, Trill, Roll, Mordent, etc.
  Cloud, SelectionCloud, DynamicCloud, etc
  TransitionTable, BellPeal, etc.
Primitive Types Used:
  All, Music Magnitudes, Events, Event Lists
Creation/Execution/Expansion Operators:
  Event List behaviors
  Class-specific creation messages, e.g., root:inversion:,
    pitch:ampl:voice:density:, etc.
  events
Data Structures:
  EventGenerators are EventLists whose identity (class membership) determines the relevance of their properties.
  The ability to represent contexts/closures is required for the more involved ostinato generators.

[5h] Event Modifiers

Objects that map their own properties or functions onto single events or whole event list hierarchies are called event modifiers (EMods). The separation between definition (what) and interpretation (how) can be maintained in an event's properties, or effected via event modifier or voice objects. These can be used for description of metalevel processes such as crescendi or rubati, or for real-time control of event list performance, and are also described elsewhere.

```
"Function usage example–make a roll-type eventList and apply a crescendo/
decrescendo to it."
   | list fcn |
   list := EventList newNamed: #test3."Create a named EventList."
   (0 to: 4000 by: 50) do:"Add 20 notes per second for 4 seconds."
         [ :index |"Add the same note"
         list add: (MEvent dur: 100 pitch: 36 ampl: 100)
            at: (index asMilliseconds)].

            "Create a line segment function."
            "x@y is Smalltalk-80 point creation short-hand"
   fcn := LineSegment breakpoints: #((0 @ 0) (0.3 @ 1) (1 @ 0)).
   list apply: fcn to: #loudness.
            "Apply the function to the list's loudness."
   aCrescendo := EventModifier function: fcn property: #loudness.
   list apply: aCrescendo."Apply it now,"
   list addModifier: aCrescendo."or add it for later application."
```

[5h2] Formal Specification of Event Modifiers

Keywords:
  EventModifier, Crescendo, Rubato
Primitive Types Used:
  All, Music Magnitudes, Functions

```
         pitch: 4)); "e.g., transp. in half-steps"
      add: (0 => (((EventList named: #highCNote) copy) pitch: 7))]

   [el := EventList newNamed: #demo1. "create a named event list"
               "add an event to it at time 0"
   el add: (MEvent duration: 1000 pitch: 36 ampl: 100);
               "add two more events after the first..."
      add: (MEvent duration: 1000 pitch: 40 ampl: 100);
      add: (MEvent duration: 1000 pitch: 43 ampl: 100);
               "add a sub-list with simultaneous evts"
      add: ((EventList new)
         add: (MEvent dur: 1000 pch: 36 ampl: 100) at: 0;
         add: (MEvent dur: 1000 pch: 40 ampl: 100) at: 0;
         add: (MEvent dur: 1000 pch: 43 ampl: 100) at: 0)]
```

[5f3] Formal Specification of Event Lists

Keywords:
    EventList
Primitive Types Used:
    All, Music Magnitudes, Events
Creation/Manipulation Operators:
    Event behaviors
    new, newNamed:, named:
    add:, add:at:, ,
    do:, select:, collect:, detect:, reject:, inject:
    expanded, expandedFrom:to:, apply:to:
Data Structures:
    EventLists are DEvents with a sorted array, list or, collection of (duration => event) associations.

[5g] Event Generators

Middle-level musical structures from the common vocabulary of Western music (chords, trills, rolls, etc.), are modeled via event generator (EGen) objects. There are three abstract EGen classes: Cluster, Cloud and Ostinato, and several standard concrete descriptions. EGens are described in gory detail in (Pope 1989b) and elsewhere.

```
"EventGenerator examples: A simple progression-C Major cadence."
   [(EventList newNamed: #progression1)
      add: ((Chord majorTetradOn: 'c4' inversion: 0) dur: 1/2);
      add: ((Chord majorTetradOn: 'f3' inversion: 2) dur: 1/2);
      add: ((Chord majorTetradOn: 'g3' inversion: 1) dur: 1/2);
      add: ((Chord majorTetradOn: 'c4' inversion: 0) dur: 1/2)]

"A Trill example--play 2 seconds of 40 msec. long notes on c5 and d5."
   [((Trill length: 2.0 rhythm: 40 notes: #(48 50)) ampl: 100) play]

"A static stochastic wave/cloud."
   [(Cloud dur: 2.0 "duration"
         pitch: (48 to: 64) "pitch range-an interval"
         ampl: (80 to: 120) "amplitude range-an interval"
         voice: (1 to: 8)"voice range-an interval"
         density: 15)]"density per second = 15 notes"

"Pentatonic selection with a transition from 1 chord to another."
   [(DynamicSelectionCloud dur: 9"9 seconds"
                     "starting and ending pitch sets"
```

(e.g., [`anEventList apply: aFunction to: aPropertyName`]), or one can use event modifier objects to have a stateful representation of the mapping.

Event lists can be declared with names (meaning that they will be persistent), and sent messages to add events to them, as in:

```
[(EventList newNamed: #test1)
 add: (0 => (MEvent dur: 1/4 pitch: ‘c3’ ampl: ‘mf’);
 add: (1 => ((MEvent new) dur: 1/4 ampl: ‘mf’ sound: #s73bw))]
```

A named event list is created (and stored) in the first example, and two event associations are added to it, one starting at 0 (seconds, by default), and the second at 1 second. Note that the two events can have different types of properties, and the handy behavior of event associations such that [`(aMagnitude) => (anImmediateDictionary)`] returns the event association [`(duration with value aMagnitude) => ([ADM]Event with given property dictionary)`]. One can use the dictionary style of shorthand with event associations to create event lists, as in this very terse way of creating an anonymous (non-persistent) list with two events:

```
[(440 Hz, (1/1 beat), 44.7 dB),
(1 => ((1.396 sec, 0.714 ampl) sound: #s73bw; phoneme: #xu))]
```

Note here the use of the fact that the “ =>”method uses its receiver as the value of a music magnitude (1 becomes 1 sec. by default as mentioned above), and the tersest event declaration format. The first line creates three music magnitudes, then turns them into associations with default property names (e.g., 1/1 beat is mapped to the duration property, 44.7 dB to the loudness), and then creates an event with these as its properties; the final comma creates an event association with 0 as the default key, and then an event list with this as the first item (am I going too fast?). The second line constructs an event and then event association in the same manner and this is added to the new anonymous event list (via the comma at the end of the first line), which is the return value of the expression.

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 to be defined (see section 5e2 above).

```
   “Create a 4-note event list.”
[(EventList newNamed: #scale)
   add: (0 => (MEvent pitch: #c));
   add: (1 => (MEvent pitch: #d));
   add: (2 => (MEvent pitch: #e));
   add: (3 => (MEvent pitch: #f))]
   “Set its loudness and voice.”
[(EventList named: #scale) loudness: #mf; voice: #harp]
   “Create a named note event with a symbolic pitch.”
[(MEvent named: #highC) pitch: #c6 ampl: 120 voice: 1]

   “Create a chord from copies of the note.”
[(EventList newNamed: #chordFromOneNote)
   add: (0 => (MEvent named: #highC));
   add: (0 => ((MEvent named: #highC) copy) pitch: #e);
   add: (0 => ((MEvent named: #highC) copy) pitch: #g)]

   “Create a chord from one note with half-step
      transposing event lists.”
[(EventList newNamed: #highCNote)
   add: (0 => (MEvent named: #highC)).

 (EventList newNamed: #chordFromOneNote)
   add: (0 => (EventList named: #highCNote));
   add: (0 => (((EventList named: #highCNote) copy)
            "a named note"
    [(MEvent named: #highC) pitch: #c6 ampl: 120 voice: 1]
```

[5e2] Formal Specification of [ADM]Events

Keywords:
  [ADM]Event
Primitive Types Used:
  All, Music Magnitudes
Default Properties (those whose semantics is predefined):
  duration, pitch, loudness, voice, location, sound, tempo
  midiPacket, frequency, instrument
  icon, text, displayList (graphical properties)
  usedToBe, relatesTo, becomes (annotational links)
Operators:
  new, newNamed:, named:
  Many class creation messages, e.g.,
  duration:voice:, duration:pitch:loudness:voice:, etc.
  Arbitrary property names or symbolic link names as messages
Data Structures:
  AEvents are dictionary or property-list data structures with a class pointer and an arbitrary number of (name -> value) pairs (associations).

  (It is assumed that implementation readers will provide for automatic memory allocation on property assignment.)

  The property names are typically symbols, and the values can be any describable object pointer, typically MusicMagnitudes or other Events or EventLists.

  [DM]Events may be implemented to have special slots/instance variables for their special properties, or they may be structurally identical to AEvents.

[5f] Event Lists

Events are grouped into collections that can be thought of as lists or arrays that are sorted by the relative start times of the event elements. Event lists are events, so that they can be freely nested into trees up to arbitrary depths. Event lists can also have their own properties, and can map these onto their events eagerly (at definition time) or lazily (at " performance"time).

EventLists are DEvents that have a sorted collection of associations between start times (durations starting at the start time of the event list) and events or sub-event-lists (nested to any depth). As DEvents, they have all the property and link behavior, and special behaviors for mapping with voices and event modifiers. Event lists can be named (when created or later, using symbolic names); when they are, they become persistent (until explicitly erased within a document or session).

The messages [`add: anAssociation`] and [`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."Note the use of " =>"for associations (rather than the Smalltalk standard " ->"). This is done in order to leave it up to the implementor whether or not to use standard Smalltalk-80 association objects, or to have a special event association class. (This might want to be changed...)

Event lists also respond to Smalltalk-80 collection-style control structure messages such as [`collect: aSelectionBlock`] or [`select: aSelectionBlock`], though this requires the representation of contexts/closures. The behaviors for applying functions to the components of event lists can look applicative

Events can be created with their property lists, or can be identified and operated on interactively. The event classes support the messages [`new`] or [`newNamed: aSymbolicName`] (see the verbose examples below), or one can simply send property list messages to the class and get an instance, as in [`AEvent sound: #komb; color: #salmon`]. An even terser shorthand is to use the Smalltalk immediate dictionary declaration format, as shown below.

In order to keep them as general as possible, events should be behaviorally vacuous--i.e., they should have as little additional behavior (aside from that implicit in their properties) as possible.

Applications should enable users to interactively edit the property lists of objects, and to browse event networks via 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.

```
      "AEvent creation example--the verbose way."
   [anEvent1 := (AEvent newNamed: #flash)
         color: #white;
         place: #there]

      "Create three events with mixed properties--the terse way"
   [(440 Hz), (1/4 beat), (44 dB), (Voice named: #oboe).]
   [490 Hz, (1/7 beat), 56 dB, (#voice -> #flute).]
   [(#c4 pitch, 0.21 sec, 64 velocity) voice: Voice default.]

      "Create a named link between two events."
   [anEvent1 isLouderThan: anEvent2]
```

[5d] D(uration)Event Objects

DEvents are AEvents that have durations and voices as standard properties, and have behaviors for accessing and mapping them. They implement the " play"message by forwarding (delegating) it to their voices (with themselves as the argument).

```
      "DEvent examples."
   [aDEvent := DEvent dur: (250 asMilliseconds)]
   [aDEvent dur]"short-hand for duration"
   [aDEvent play] "forwards to voice, or the default voice"
```

[5e] M(usic)Event Objects

MEvents are DEvents with pitch and loudness behaviors including mapping and defaulting. They extend the short-hand of DEvent as well.

```
      "MEvent examples."
   MEvent duration: (Duration value: 1/2)
         pitch: (Pitch value: #c2)
         loudness: (Loudness value: #mf))
   "same as"
   [MEvent dur: 1/2 pitch: #c2 ampl: #mf]
   "same as"
   [1/2 beat, #c2 pitch, #mf ampl]

   [(MEvent duration: 500)
         at: #color put: #green;
         at: #position put: (0.3 @ 0.8)]
   "same as"
   [(MEvent duration: 500) color: #green;
         position: (0.3 @ 0.8)]
```

Ergon--abstract loudness/amplitude model
  Loudness/Amplitude--dynamic model
Positus--abstract position/space
  Position--1- or more-d location model
  Directionality--direction and radiation pattern
  Spatialization--environment model

Music Magnitude Examples

```
(Duration value: 1/16) asMS "answers 62"
(Pitch value: 36) asHertz "answers 261.623"
(Amplitude value: 'ff') asMidi "answers 106"
```

[5b3] Formal Specification of MusicMagnitudes

Keywords:
Duration, Pitch, Loudness, Meter, Mode (abstract models)
HertzPitch, SecondDuration, RatioDuration, SymbolicLoudness, etc.

Primitive Types Used:
Integer, Float, String, Symbol

Creation Operators:
value:, value:relativeTo:
(Smalltalk-style coercion messages--optional)
  asDynamic, asVelocity, asKey, asPitch, etc.
(terse postfix operators)
  Hz, ms, msec, sec, beat, location, velocity, key, pitch, etc.

Comparison/Arithmetic Operators:
>, < >, <=, !=
+, -, /, *

Data Structures:
MusicMagnitudes require class, species, and value pointers.
The class and species may be symbols or class pointers.(?)
The value is defined to be any immediate type (mostly frequently numbers and strings/symbols).
ConditionalDurations assume the ability to represent block context or closure objects, and to evaluate them with arguments and use the return value in Boolean expressions.

[5c] Generic A(bstract)Event Objects

The simplest view of events is as Lisp-like property lists, Smalltalk-like dictionaries of property names and values, the relevance and interpretation of whom is left up to others (e.g., 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 properties 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, or as direct behaviors. One can, for example, set an event to be " blue"by saying [`anEvent at: #color put: #blue "dictionary-style accessing"`] or simply [`anEvent color: #blue "behavioral accessing"`]. Events can be linked together by having properties that are typed 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."

The representational model classes are used as species for musical quantities such as pitch, duration and loudness. This set of abstractions is extensible, though it's not certain whether significant new abstractions can be described in a generally useful and portable manner. The implementation classes are designed to be easily extended with new implementation types. MM objects can be described by their species and value, or often by a number and postfix type operator, e.g., (Pitch value: 440.0) or (440.0 Hz), (Amplitude value: 0.7071) or (3 negated dB).

All MMs obey the Smalltalk-80 abstract Magnitude protocol, i.e., scalar comparison messages such as > or <=. Most of them also respond to normal arithmetic messages such as + and -. 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?). (Whether or not implementations can be built that reflect these assumptions remains to be seen.)

Music magnitudes also have some special behaviors whereby they can create and answer event associations that include themselves as default properties, as described in section 5c below. This is very useful for terse description of events.

Applications for this representation should support abstract interactive editors for music magnitude objects that support the manipulation of the basic hierarchy described above, as well as its extension via " light-weight"programming.

The minimal system should include the following implementations:

Duration:
- milliseconds and seconds expressed as whole or floating-point numbers (e.g., [250 msec or 1.0 sec]);
- fractional " beats"relative to some metronome or givenwhole-note (e.g., [1/4 beat]); and
- " conditional durations"(e.g., [:ev :t | t value > 40]).

Pitch:
- Hertz frequency (e.g., [440.0 Hz]);
- symbolic well-tempered note name (e.g., [#c4 pitch]);
- MIDI key number in the range (0 .. 127) (e.g., [120 key]);
- fractional ratio to another pitch (e.g., [11/7 of: (C)]);
- octave.pitch notation (e.g., 4.2 for d4); and
- octave.cent notation (e.g., 4.200 for d4)

Loudness:
- relative amplitude in the range (0.0 .. 1.0) (e.g., [0.7071 loudness]);
- symbolic dynamic name (e.g., ['mf' loudness]); and
- MIDI key velocity in the range (0 .. 127) (e.g., [64 velocity]).

Mode:
- Pitch-class, set, or gamut
- Defined with a name- or value-set

Meter:
- metronome number or ratio
- Defined and referred to via event modifiers

Music Magnitude Abstract Model Hierarchy
- Chronos--abstract time model
  - Duration--duration model
  - Meter--beat or metronome model
- Chroma--pitch or color
  - Pitch--scalar pitch model
  - Mode--mode or cyclical pitch class

```
   sec1 add: (...last event...) at: (9947/4 beats).

"section 2--terse, add event assoc. using ‘,’ concatenation operator."
   sec2 := ((0 beat) => (...event1...)), ((1/4 beat) => (event2)),

         "...section 2 events...",
      ((2109/4 beats) => (event3308)).
"Event list composition (may be placed anywhere)"
   piece add: sec1 at: 0. "add the sections in sequence."
   piece add: sec2 at: (9955/4 beats). "sec2 starts after sec1."

"declare global (named) event modifiers, functions, etc."
   ((Rubato newNamed: #tempo)
      function: (...tempo spline function...)
      property: #startTime).
   piece tempo: (Rubato named: #tempo).

   (Crescendo newNamed: #dynamic)
      function: (...dynamic function...)
      property: #loudness.
   piece dynamic: (Crescendo named: #dynamic).

"add a few annotation links."
   sec1 precedes: sec2. "set up annotational links."
   sec2 succedes: sec1. "these messages are property assignment."
      "e.g., same as: [sec2 at: #succedes put: sec1]"
   (...)
```

The sections with declarations of variables, naming of event lists, event definition, functions and event modifiers, and annotation, can be freely mixed (whereby variable names must be declared before they are referenced).

==

5: Language Primitives

[5a] Immediate Values

As described in section 4b, SmOKe requires that a range of primitive data types be expressible as ASCII/ ISO strings. This list includes numbers, strings/symbols, points or complex numbers, and block contexts or closures.

[5b] Music Magnitude Models

Abstract descriptive models for the basic music-specific magnitude types such as pitch, loudness or duration are needed as the foundation for SmOKe. These are thought to be similar to what Smalltalk-80 calls magnitude objects in that they represent partially- or fully-ordered scalar or vector quantities with (e.g.) numerical or symbolic values. Music magnitude objects have values with both a model and a representation; i.e., 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 where meaningful with (ideally) no loss of “ precision.”This decision is made on a species-by-species basis (i.e., the species is used to determine coercion rules in mixed-mode same-species arithmetic), so that, for example, [(261.26 Hz) + (MIDI key number 8)] can be handled differently than [(1/4 beat) + (80 ms)]. The classes of these objects will depend on the “ type”of their values (i.e., floating-point numbers, fractions, or strings). (Each species supports the (somewhat questionable) notion of a “ most general”representation. The current defaults are Hertz, seconds, and dB, though these are up for discussion.)

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, unless such a proposal is quite similar to Smalltalk-80.)

The data structure description presented below attempts a high-level pseudo-C. SmOKe also assumes the ability to create and name persistent objects, and to use object pointers, or other external references, in associations or bindings. Dictionaries or property 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]).

[4c] Naming and Persistency

The names of the abstract classes are known and are treated as special globals. The object creation messages are standardized among the primitive boundaries. In a good, abstract style, 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 SymbolicPitch instance).

All central classes are assumed to support " persistency through naming"whereby an object (e.g., an event list or a voice), which is explicitly named gets stored in a global dictionary under that name until explicitly released. The class messages [`newNamed: aSymbolOrString`] and [`named: aSymbolOrString`] are used for creating and referencing persistent instances. How the global dictionaries are managed, and what the exact (temporal) scope of the persistency is, is not defined here. The (lexical) extent is assumed to be an SmOKe " document"or " module."

[4d] Score Format

A score in the SmOKe representation consists of one or more parallel or sequential streams of event lists whose events may have interesting properties and links. Events are described as messages to the event classes that create event instance objects. 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 or pattern, or grain, 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."The paragraphs below introduce each of the language primitives, which are also the abstract classes of the prototype implementation in Smalltalk-80.

An SmOKe score, viewed as a document, 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, e.g.,

```
"declarations of variable names and top-level event list."
   | piece sec1 sec2 |
                    "name declarations are optional but advised."
   piece := EventList newNamed: #piece.

"section 1--verbose, add events using add:at: message."
   sec1 := EventList newNamed: #section1.
   sec1 add: (...first event (may have many properties)...) at: 0.
   sec1 add: (...second event...) at: 0. "starts with a chord."

         "...section 1 events, in parallel or sequentially..."
```

- 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, DSP code, instrument definitions, mixing scripts, or Pla (application issue); and
- possibility of parsing live performance into some rendition in the representation, and of interpreting it (in some rendition) in real-time (application issue).

==

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. Voice objects and applications determine the interpretation of events’ properties, and may use “ standard”property names such as pitch, loudness, voice, duration, or position.

Events are grouped into event collections or event lists by their relative start times. 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. 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.

Sampled sounds are also describable, by means of synthesis “ patches,”or signal processing scripts involving a vocabulary of sound manipulation messages.

==

4: Language Syntax and Features

[4a] Linear Description Language

The SmOKe music representation can be linearized easily in the form of Smalltalk-80 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).

[4b] Language Requirements

The basic representation itself is language-independent, but assumes that the following immediate types are representable as ASCII/ISO character strings:

- 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-decimal place) floating-point numbers,
- arbitrary-length ASCII/ISO strings,
- unique symbols (i.e., managed with a hash table),
- 2- and 3-dimensional Cartesian points (or n-dimensional complex numbers) (polar representation optional), and
- functions of one or more variables described as breakpoints for linear, exponential or spline interpolation, Fourier sums, series, sample spaces, or probability distributions.

structures. The abstract description language is introduced and comments are made about the desired support for manipulating structures in various formats. 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.

The format of the description is as follows. Section 2 presents the background motivations and coarse-level system requirements. In section 3, a quick overview of SmOKe is given. The syntax of the description language, and the requirements placed on implementation languages are discussed in section 4. Section 5, the bulk of the document, presents each of SmOKe's primitive models, giving a text descriptions, several usage examples, notes and comments, and a semi-formal description of the defined classes, behaviors and data structures. Section 6 is a collection of comments on SmOKe, its design process, or related technical issues.

The naming conventions and the code description examples use the Smalltalk-80 programming language, but the underlying 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.

==

2: Requirements and Motivations

[2a] Motivations

The desire has been voiced (Smallmusic 1991), for an expressive, flexible, abstract, and portable structured music description and composition language. The goal is to develop a kernel language (implemented in a portable language), that can be used for structured composition, real-time performance, processing 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 a wide range of applications can be ported to run on it (by members of the Smallmusic group) within a short period after its " standardization."

[2b] Minimal Engineering Requirements

The representation must provide or support:

- abstract models of the basic musical quantities (scalar magnitudes such as pitch, loudness or duration);
- instrument/note (voice/event, performer/music) abstractions;
- sound functions, granular description, or other (non-note-oriented) description abstractions;
- flexible grain-size of " events"in terms of " notes,"" grains,"" elements,"or " textures;"
- event, control, and sampled sound processing description levels;
- nested/hierarchical event-tree structures for flexible description of " parts,"" tracks,"or other parallel or sequential organizations;
- separation of " data"from " interpretation"(what vs. how in terms of having interpretation objects);
- abstractions for the description of " middle-level"musical structures (e.g., chords, clusters, or trills) (may be optional);
- annotation and marking of event tree structures supporting the creation of heterarchies (lattices) and hypermedia networks;
- annotation including common-practise notation possible;

The *Smallmusic Object Kernel*: A Music Representation, Description Language, and Interchange Format

Smallmusic Discussion Group -- CCRMA/Palo Alto, CNMAT/Berkeley, the Ether
Version 0.6 -- LastEditDate: Feb 4, 1992
Please send comments to: smallmusic@xcf.Berkeley.edu or stp@CCRMA.Stanford.edu

==

Abstract

This document describes an object-oriented description language for musical parameters, events and structures known as the Smallmusic Object Kernel (SmOKe). In object-oriented software terms, the representation is described in terms of software class hierarchies of objects that share state and behavior and implement the description language as their protocol. The authors believe this representation, and its proposed linear ASCII description in Smalltalk-80 syntax, to be well-suited as a basis for: (1) concrete description languages in other languages, (2) specially-designed binary storage and interchange formats, and (3) use within and between interactive multi-media, hypermedia applications in several application domains.

The text before you is at present a vaguely-organized collection of requirements, definitions, and examples. Readers are encouraged (!) to comment, correct, argue, present counter-examples, or add detail where they please. This document, its previous versions, and the discussion among the members of the electronic mail group " smallmusic@xcf.Berkeley.edu,"are archived for new readers at the InterNet network cites " anonymous@ccrma-ftp.Stanford.edu:/pub/st80"at Stanford University and " anonymous@xcf.Berkeley.edu:/misc/smallmusic"at U. C. Berkeley.

==

Outline

==

1: Introduction

This document presents the Smallmusic Kernel, an object-oriented music representation language that 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.

The underlying music representation is presented in terms of one possible text-based description (in the syntax of the Smalltalk-80 programming language [Goldberg and Robson 1989]), and the in-memory data structures that might be used to implement it. It is intended that independent parties be able to implement compatible abstract data structures, concrete interchange formats, and linear description languages based on the definitions given here.

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