Object-oriented music representation

STEPHEN TRAVIS POPE

Center for Computer Music Research and Composition, Department of Music, University of California, Santa Barbara, CA 93106, USA

There is a rich literature on the design of formal languages for music representation on computers. Over the last thirty years, several generations of software technology have been applied to this problem, including structured software engineering, artificial intelligence, and object-oriented (O-O) software technology. This article introduces the basic notions of O-O software technology, and investigates how these might be useful for music representation. In particular, the author's Smalltalk music object kernel (Smoke) music representation language is described and examples given that illustrate the most important of Smoke's features.

1. INTRODUCTION

There have been several systems over the past decade that have used object-oriented (O-O) techniques in a variety of programming languages (Lisp, Objective C, or Smalltalk, for example) to build music representations. In some of these it is invisible to the user that O-O technology is being used 'under the hood', while in others the O-O paradigm is quite obvious and becomes part of the user's conceptual model of the application.

The article begins with a short section on the principles of O-O software technology, then discusses some issues in the design of O-O programming languages and systems. The topic of using O-O languages as the basis of music representation systems is then presented, followed by a detailed description of the Smalltalk music object kernel (Smoke) music representation language.

The intended audience for this discussion includes 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. Other documents (Pope 1993, 1995) describe the software environment within which Smoke has been implemented (the MODE).

2. OBJECT-ORIENTED SOFTWARE TECHNOLOGY

Object-oriented software technology has its roots both in structured software methods and in simulation languages designed in the mid-1960s; it can be said to be *evolutionary* rather than *revolutionary* in that it builds on and extends earlier technologies rather than being a radical new idea in software engineering. Object-oriented technology can be said to be the logical conclusion of the trend to structured, modular software, and to more sophisticated software engineering methodologies and tools of the 1970s and early 1980s.

Object-oriented technology is based on a set of simple concepts that apply to many facets of software engineering, ranging from analysis and design methodologies to programming language design, databases, and operating systems. There is a mature literature on each of these topics (see, for example, the *Proceedings of the ACM Conferences on O-O Programming Systems, Languages, and Applications* (OOPSLA), or any of the journals and magazines devoted to this technology, such as the *Journal of Object-Oriented Programming*, *Object Messenger*, *Object Magazine* or the *Smalltalk Report*); I will focus in this article on O-O programming methods and languages for musical applications.

There are several features that are generally recognised (Wegner 1987) as constituting an O-O technology: *encapsulation*, *inheritance*, and *polymorphism* are the most frequently cited. I will define each of these in the sections below. There are also several issues that arise when trying to provide a software development environment for the rapid development of modern applications; these can be divided into *methodology* issues, *library* issues, and *tool* issues. We will not discuss these in detail here, but refer the interested reader to Goldberg and Pope (1989) and Pope (1994).

2.1. Encapsulation

Every generation of software technology has had its own manner of packaging software components – either into 'jobs' or 'modules' or other abstractions for groups of functions and data elements. In traditional modular or structured technology, a *module* includes one or more (public or private) data types and also the (public or private) functions related to

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

these data items. In large systems, the number of functions and the 'visibility' of data types tended to be large, leading to problems with managing data type and function names, which are required to be unique in most structured programming languages.

Object-oriented software is based on the concept of *object encapsulation* whereby every data type is strongly associated with the functions that operate on it. There is no such thing as a stand-alone data element or an unbound function; data and operations (state and behaviour in O-O terms) are always encapsulated together. A data type known as a *class*, of which individual objects are *instances*, has a definition of what data storage (state) is included in the instances (called *instance variables* in the Smalltalk family of languages[1]), and of what functions (behaviour or *methods*) operate on instances of the class. Strict object encapsulation means that the internal state of an object is completely invisible to the outside and is accessible only through its behaviours.

Any object must be sent messages to determine its state so that, for example, an object that represents a two-dimensional (2D) point in Cartesian geometry would have x and y instance variables and methods named x and y for accessing them. One advantage of this strict separation of private state from public behaviour is, for example, that another kind of geometrical point might store its data in polar coordinates, with its magnitude and angle (r and Θ). It would be very good to be able to use these two kinds of points interchangeably, which is possible if we are only concerned with *what* they can do, and not with *how* they do it. Behaviourally, they are identical (ignoring performance for now); we can send them each messages such as 'x' to get their x coordinate, without having to know whether it is cached or computed. Figure 1 shows these two kinds of point objects.

As an example from the musical domain, imagine an object that represents a musical event or 'note'. This object would have internal (strictly private) data to store its parameters (e.g. duration, pitch, loudness, timbre, and other properties) and would have methods for accessing these data and for 'performing' itself on some medium such as a MIDI channel or note list file. Because the internal data of the object is strictly hidden (behind a behavioural interface), one can only access its state via behaviours so that, if the note object understood messages for several kinds of pitch (e.g. pitchInHz, pitchAsNoteNumber, and pitchAsNoteName) then the user would not have to worry about how exactly the pitch was stored within the note. Figure 2 illustrates a possible note event object. The state vs behaviour differentiation is related to what old-fashioned structured software technology calls *information hiding* or the separation of the *specification* (the *what*) from the *implementation* (the *how*). In *uniform* O-O languages (e.g. Smalltalk), this object encapsulation is strictly enforced, whereas it is weaker (or altogether optional) in *hybrid* O-O languages (such as C++).

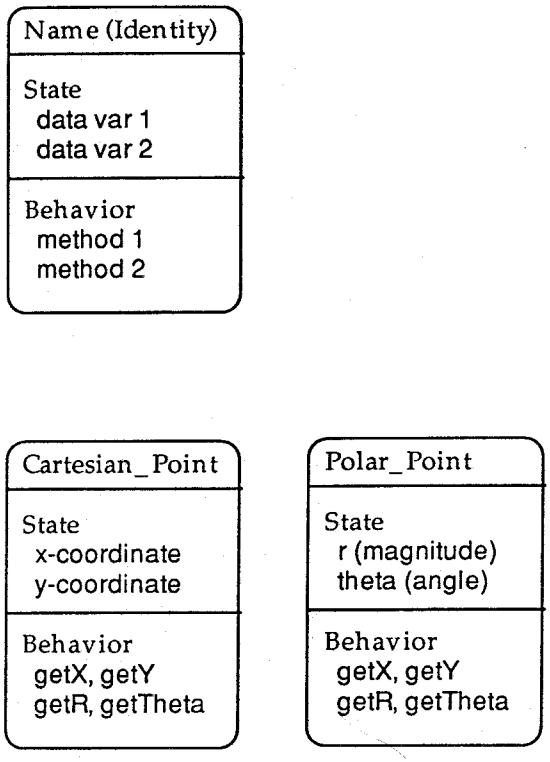


Figure 1. Object encapsulation. (a) The components of an object, showing that the object may have an internal state (data storage) that is hidden from the outside, and provides a public interface in terms of its behaviours (functions). (b) Two different objects representing geometrical points are shown whose behaviours are identical even though their internal storage formats differ. The external user (client) of these points has no way of telling from the outside which is which.

2.2. Inheritance

Inheritance is a simple principle whereby classes can be defined as *refinements* or *specialisations* of other classes, so that a collection of classes can form a tree-like specialisation or *inheritance* hierarchy. At each level of this kind of subclass/superclass hierarchy, a class only needs to define how it differs from its superclass.

Examples of class hierarchies are well known (and widely misunderstood) in the literature on O-O systems. The hierarchy of classes used to represent numbers makes a good starting example. If one had a

[1] This includes the several dialects of Smalltalk, Self, and other Smalltalk-derived languages.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

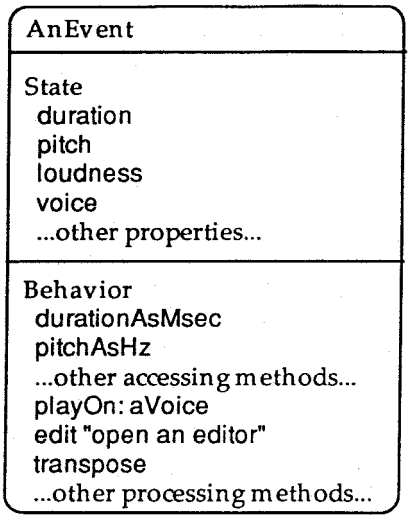


Figure 2. A musical event or 'note' object. This exemplary note event object has state for its duration, pitch, loudness, and position, and behaviours for accessing its properties and for performing it.

class for objects that represented 'units of measure' (often called *magnitudes* in the abstract sense) then it is easy to see that numbers would be one possible subclass (refinement) of it, and that the various types of numbers (integers, fractions, and floating-point numbers) might be further subclasses of the number class. There are also other possible magnitude classes that are not numbers: dates and times-of-day, for example. Figure 3 shows a possible hierarchy of magnitudes and numbers.

A musical example of this would be a system with several types of event parameters: different notations

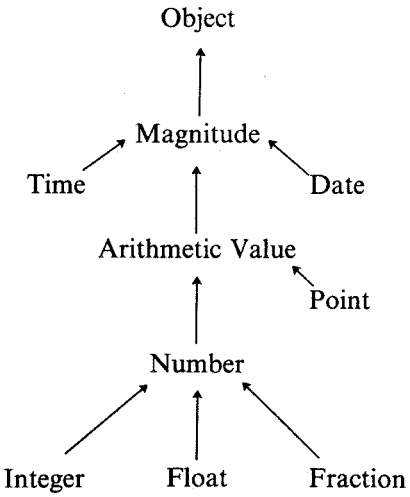


Figure 3. A class hierarchy for magnitudes. This example shows one possible inheritance hierarchy (class tree) for magnitude objects such as numbers. This is a subset of the actual Smalltalk-80 Magnitude class hierarchy.

for pitches, durations, loudness values, etc. If there was a high level ('abstract') class for representing musical pitches, then several possible pitch notations could be constructed as subclasses ('concrete classes') of it. Further examples of this kind of subclass/superclass hierarchy will be presented below. Remember that at each level we are most interested in behavioural differences (rather than storage representations).

2.3. Polymorphism

In simple terms, polymorphism means being able to use the same function name with different types of arguments to evoke different behaviours. Most traditional programming languages allow for some polymorphism in the form of *overloading* of their arithmetical operators, meaning that one can say (3 + 4) or (3.5 + 4.1) in order to add two integers or two floating-point numbers. The problem with limits on polymorphism (overloading) is that one is forced to have many names for the same function applied to different argument types (e.g. function names like playEvent(), playEventList(), playSound(), playMix(), etc.). In uniform O-O languages, all functions can be overloaded, so that one can create many types of objects that can be used interchangeably (e.g. many different classes of objects can handle the message play in their own particular ways).

Using polymorphism may mean that some additional runtime overhead is incurred, but it can be considered essential for a language on which to base an exploratory programming environment for music and multimedia applications. In message-passing O-O languages, the receiver of a message (i.e. the object to which the message is sent) determines what method to use to respond to a message. In this way, all the various types of (for example) musical events and event collections can receive the message play and will respond accordingly by performing themselves, although they may have very different methods for doing this.

2.4. Hybrid and uniform O-O programming languages

There are some systems that mix an 'O–O flavour' into a programming language that is based on other princples, such as structured programming; Ada, Common Lisp, C++, and Objective C are examples of this kind of hybrid language. Several other languages provide only strictly O-O facilities (data types and operation paradigms), and can be said to be uniformly object oriented; examples of these are

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

Smalltalk, Self, Actor, and Eiffel. There is some debate as to whether it is necessary to adopt a uniform O-O approach to achieve the full benefit of O-O software technology. Some commercial users of O-O technology (who are least likely to be interested in theoretical or 'religious' arguments) cite a productivity increase of 600 per cent when moving from a hybrid to a uniform O-O programming language (data from C++ relative to Smalltalk-80) and a 1,400 per cent difference in productivity between uniform O-O and structured languages (Hewlett-Packard 1993).

2.5. Other language, methodology, library, and tool issues

There are a number of other issues that influence how appropriate a software system will be for developing music representation tools. *Uniformity*, *simplicity*, *expressiveness*, and *terseness* are all important in a programming language; a large and well documented *class library*, a set of integrated software development *tools*, and an appropriate analysis and design *methodology* are all necessary components of a programming system that will facilitate building sophisticated modern music software applications (Deutsch and Taft 1980, Barstow, Shrobe and Sandewall 1985, Goldberg and Pope 1989, Pope 1994). As interesting as an in-depth discussion of these issues may be, it is outside of the scope of the present text.

3. OBJECT-ORIENTED MUSIC REPRESENTATION

Several O-O music description languages have been described in the literature, starting soon after the first O-O environments became practical (Krasner 1980, Rodet and Cointe 1984). Today, systems such as the Music Kit (Jaffe and Boynton 1989), Fugue/Nyquist (Dannenberg 1989), Common Music (Taube 1991), Kyma (Scaletti 1989), DMix (Oppenheim 1989), and the MODE (Pope 1993) are in widespread use.

In some of these systems (e.g. the NeXT Music Kit), the end-user is relatively unaware of the use of the object paradigm, while in others (e.g. Dmix, Kyma, and the MODE), it is presented directly to the user as the primary organisational technique for musical 'objects'. Most O-O music representation languages use a hierarchy of different classes to represent musical events and collections or sequences thereof. Some systems have many 'event' or 'note' classes (e.g. MIDI events vs note list events) and use polymorphism among their messages, while others have few of these classes and use 'drivers' or 'performers' to interpret events. As we will see below, the Smoke language (part of the MODE environment) falls into the latter category.

3.1. Language requirements

Several of the groups that have worked on developing music representations have started by drawing up lists of requirements on such a design (Dannenberg, Dyer, Garnett, Pope and Roads 1989), and separating out which items are truly determined by the underlying representation, and which are interface or application issues. The group that designed the Smoke language developed the following list, using the results of several previous attempts as input.

A useful O-O music representation, description language, and interchange format should provide or support:

- abstract models of the basic musical magnitudes (units of measure such as pitch, loudness, and duration),
- sound objects represented as functions of time, grains, or other (non-note-oriented) abstractions,
- flexible grain size of 'events' in terms of 'notes', 'grains', 'elements', or 'textures',
- description and manipulation levels of data streams including event, control, and sampled function abstractions,
- flexible, hierarchical, event trees (i.e. nested event lists) for 'parts', 'tracks', or other parallel or sequential structures,
- separation of 'data' from 'interpretation' (*what* vs *how* in terms of having separate objects such as the note/instrument, event/voice, or part/performer 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 with many roots) and hypermedia networks,
- graphical annotation including common practice notation (this is an application issue),
- description of sound synthesis and processing models such as sound file mixing or DSP,
- convertors for many common formats, such as MIDI data, Adagio, note lists, HyTime/SMDL, DSP code, Music V-style instrument definitions, mixing scripts (this is an interface issue), and
- parsing live performance into some rendition of the representation, and of interpreting it in real time (this is an application issue related to simplicity, terseness, etc.).

3.2. Programming environment requirements

The high level components of a good O-O music language should be written in a rapid-turnaround incrementally compiled software development environment. The use of a powerful and abstract central programming language integrated with the user interface 'shell' is very important; the goal is to

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

address the issues of learnability, scalability, abstraction, and flexibility, and provide a system that meets the requirements of *exploratory programming systems* as defined in Deutsch and Taft (1980) or *interactive programming environments* as defined in Barstow *et al*. (1984). The system should also be designed to support interchangeable ('pluggable') interactive front ends (e.g. graphical editors, or musical structure description languages) and back ends (e.g. MIDI output, sampled sound processing commands, sound compiler note lists, or DSP coprocessor control). The components of such packages have been defined (see also Layer and Richardson (1991) and Goldberg and Pope (1989)) as: (i) a powerful, abstract programming language with automatic storage management, interpreted and/or incrementally compiled execution, and a runtime available compiler; (ii) software libraries including reusable low and high level modules; (iii) a windowed interactive user interface 'shell' text and/or menu system; (iv) an integrated set of development, debugging, and code management tools; (v) an interface to 'foreign language' (often C and assembler) function calls; and (vi) a software framework for constructing interactive graphical applications.

The primary languages for such systems have been Lisp and Smalltalk (and to a lesser extent Prolog and Forth), because of several basic concepts. Both languages provide an extremely simple, single paradigm programming model and consistent syntax that scales well to complex expressions (a matter of debate). Both can be interpreted or compiled with ease and are often implemented within development environments based on one or more interactive 'read-eval-print loop' objects. The history of the various Lisp machines demonstrates the scalability of Lisp both up and down, so that everything from high level applications frameworks to device drivers can be developed in a single language system. The Smalltalk heritage shows the development of the programming language, the basic class libraries, the user interface framework, and the delivery platform across at least four full generations. The current VisualWorks system from ParcPlace-Digitalk Inc. is a sophisticated development environment that is also portable, fast, commercially supported, and stable. Other commercial (e.g. from IBM) or public domain (e.g. GNU) Smalltalk systems are largely source code compatible with 'standard' Smalltalk-80, and there is an ANSI standards effort underway.

The remainder of this article will describe one O-O music description language – the Smalltalk music object kernel or Smoke – in detail. To a large degree, the same issues that arise in the design of music representation languages using non-O-O foundations (Dannenberg 1994, Wiggins, Miranda, Smaill and Harris 1994) must also be addressed in O-O systems. Smoke shares many features with the other O-O music languages mentioned above, and is radically different from most of them in several ways as well.

4. THE SMOKE MUSIC REPRESENTATION

The Smalltalk music Object Kernel (Smoke) is a uniformly object-oriented representation, description language and interchange format for musical parameters, events, and structures. This representation, and its linear ASCII text description, are well suited as a basis for (i) concrete description interfaces in other languages, (ii) binary storage and interchange formats, and (iii) 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 (e.g. Music V), the flexibility and extensibility of 'real' music programming languages (e.g. Common Music), and the non-sequential description and annotation features of hypermedia description formats (e.g. HTML). The description given below 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.

4.1. Summary

The 'executive summary' of Smoke from Pope (1992) is as follows. Music (i.e. a musical surface or structure), can be represented as a series of *events* (which generally last from tens of milliseconds 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 loudness values), and models of many common musical magnitudes are provided. At the minimum, all events have a duration property (which may be zero). 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); 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

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

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.

4.2. The Smoke language

The Smoke music representation can be linearised 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 compiler to 'instantiate' the objects (rather than needing a special formatted reading function). The postfix expression format taken from Smalltalk (receiverObject keyword: argument) is easily parsable in C++, Lisp, and other languages.

4.2.1. Language requirements

The Smoke representation itself is independent of its implementation language, but assumes that the following immediate types are representable as character strings in the host language:

- arbitrary-precision integers (or at least very large ones – 64 or more bits),
- integer fractions (i.e. stored as numerator/denominator, rather than the resulting whole or real number),
- high-precision (64 or more bits) floating-point numbers,
- arbitrary-length ASCII/ISO strings,
- unique symbols (i.e. strings that are managed with a hash table),
- 2 and 3D 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 summations, series, sample spaces, and various probability distributions.

The support of *block* 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 behavioural information. (It is beyond the scope of the present design to propose a metalanguage for the interchange of algorithms.) Associations (i.e. key/value tuples) and dictionaries (i.e. lists of associations that can be accessed by key) must also either be available in the host language or be implemented in a support library.

4.2.2. Naming and persistency

The names of abstract classes are widely known and are treated as special global objects. (As in Smalltalk, they are written with an initial capital letter (see Appendix 1).) The names of abstract classes are used wherever possible, and instances of concrete subclasses are returned, as in the expressions (Pitch value: 'c3') or ('c3' pitch) (strings are written between single quotes in Smoke), which both return an instance of class SymbolicPitch. All central classes are assumed to support 'persistency through naming' whereby any object that is explicitly named (e.g. (Pitch named: #tonic value: 'c3')) 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'.

4.2.3. Score format

A Smoke score consists of one or more parallel or sequential event lists whose events may have interesting properties and links among them. Magnitudes, events, and event lists are described using class messages that create instances, or using immediate objects and the terse postfix operators demonstrated below. These objects can be named, used in one or more event lists, and their properties can change over time. There is no predefined '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 subsounds of a complex 'note', or as 'motives', 'tracks', 'measures', 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 programme. A score is structured as executable Smalltalk 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 (see the examples below).

4.3. Smoke music magnitudes

Descriptive models for the basic music-specific magnitudes (units of measure such as pitch, loudness and duration) are the foundation of Smoke. These are similar to Smalltalk magnitude objects in that they represent partially or fully ordered scalar or vector quantities with (for example) numerical or symbolic values. Some of their behaviour depends on what

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

they stand for (the *representation* class), and some of it on how they are stored (the *implementation* class). These two aspects are the objects' *species* and their *class*. The two pitch-species objects described as (440.0 Hz) and ('c#3' pitch), for example, share some behaviour, and can be mixed in arithmetic with no loss of precision. The expression ('f4' pitch+261.26 Hz) should be handled differently than (1/4 beat+80 msec). The class of a music magnitude object depends on the 'type' of its value (e.g. a floating-point number or a string), while its species will denote what it represents. Only the species is visible to the user.

Music magnitudes can be described using prefix class names or postfix-type operators, e.g. (Pitch value: 440.0) or (440.0 Hz) or ('a3' pitch) all represent equivalent objects, as do (Amplitude value: 0.7071) and (–3 dB). The representation and interchange formats should support all manner of 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 in this case).

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 some form of 'lightweight' programming.

Figure 4 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 shown in figure 5 demonstrate the verbose (class message) and terse (value + post-operator) forms of music magnitude description. Note

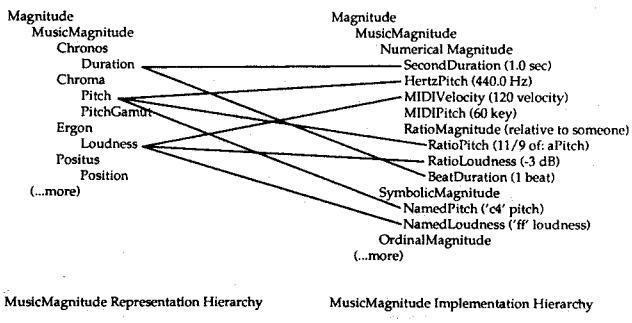


Figure 4. Smoke music magnitude model abstractions and implementation classes. This figure shows the two hierarchies used for modelling music magnitudes: the representation or species hierarchy on the left, and the implementation or class hierarchy on the right.

```
"Verbose examples--Smalltalk class messages."
(Duration value: 1/16)          "Same as 1/16 beat--a 16th note duration"
(Pitch value: 60)               "Same as 60 pitch--middle C"
(Amplitude value: 'ff')         "Same as 'ff' dynamic--fortissimo"

"Music Magnitude coercion examples."
(Duration value: 1/16) asMS     "Answers 62 msec at tempo = mm 60"
(Pitch value: 36) asHertz       "Answers 261.623 Hz."
(Amplitude value: 'ff') asMIDI  "Answers MIDI velocity 106."

"Terse examples--value + post-operator."
"Pitches"          ('f#4' pitch)  (261 Hz)  (71 key)
"Dynamics"         ('pp' dynamic)  (-38 dB)  (56 velocity)
"Durations"        (1/4 beat)  (2381 msec)  (2.3 sec)
"Spatial Positions"        (0@0 position)  (1@2@3 position)
```

Figure 5. Music magnitude usage examples (comments are written between double-quotes, as in "This is a comment.").

that comments are delineated by double quotes (or nesting curly braces) in Smoke.

The minimal set of music magnitudes is illustrated in figure 5. There are additional desirable formats (e.g. Duration until: [some condition]) that are more difficult to describe in a portable format, and so are not presented here.

Note that these music magnitudes ignore the 'relative' quantities of pitch interval, meter, and relative loudness; each of these is handled by a special *ratio magnitude* class, e.g. (Pitch value: 11/9 of: (Pitch named: #tonic)).

The great perception debates of pitch vs frequency and loudness vs dynamic vs amplitude are not addressed here; it is no problem to have a single event with contradictory or redundant magnitudes (e.g. [anEvent pitch: ('a3' pitch); frequency: (446 Hz)]). It is left up to voices or other interfaces to interpret these values as appropriate – a music printing application might use the pitch rather than the frequency, but a note-list-oriented output voice might take the frequency value instead.

4.4. Smoke event objects

The simplest view of events is as Lisp-like property lists – dictionaries of property names and values, the relevance and interpretation of which are left up to others (such as voices and applications). Events need not be thought of as mapping one-to-one to notes, though they can be used to represent note level objects. There may be one-to-many or many-to-one relationships (or both) between events and notes. Events may have arbitrary properties, some of which 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.

An event's properties can be accessed as keyed dictionary items (i.e. events can be treated as record-like data structures), or as direct behaviours (i.e. events can be thought of as purely programmatic). One can

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

set an event to 'have the colour blue' (for example), by saying

```
"dictionary-style accessing--use the at:put:
   message"
(anEvent at: #color put: #blue)
```

or more simply

```
"behavioural accessing--use
   the prop. name as a message"
(anEvent color: #blue)
```

(whereby #string represents the unique symbol whose name is 'string' – see Appendix 1). Events can be linked together by having properties that are associations to other events or event lists, as in (anEvent soundsLike: anotherEvent), 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 runtime as in Smalltalk 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 in figure 6. As shown there, the concatenation of several music magnitudes (properties) is read as the declaration of an event with the given properties. Applications should enable users to interactively edit the property lists of event objects, and to browse event networks via their times or their links using flexible link description and filtering editors. Common properties such as pitch, duration, position, amplitude, and voice are manipulated according to their 'standard' semantics by many applications.

```
"Event creation examples--the verbose way (using class messages)."
"Create a named (and therefore persistent) event."
(Event newNamed: #flash)
        color: #white;
        place: #there.

"Create and play a musical event the long-hand way."
(Event duration: (Duration value: 1/2)
                pitch: (Pitch value: #c2)
                loudness: (Loudness value: #mf))
        playOn: (MIDIVoice onMIDIPort: 0 channel: 1)

"Create three events with mixed properties--the terse way."

        "Abstract properties, use the default voice"
event1 := (440 Hz), (1/4 beat), (-3 dB), Voice default.

        "MIDI-style properties, use a numbered voice"
event2 := 38 key, 280 ticks, 26 vel, (#voice -> 4).

        "Note-list-style properties, use a named voice"
#c4 pitch, 0.21 sec, 0.37 ampl, (Voice named: #oboe).

        "Create a named link between two events."
event1 isLouderThan: event2.
```

Figure 6. Smoke musical event description examples.

Note that events have durations but no start times; this is provided by the context in which they are used (see below).

4.5. Event lists

Events are grouped into collections (event lists) where a list is composed of associations between start times and events or sublists (nested to any depth). Schematically, this looks like:

```
(EventList:=(dur1=>event1),
            (dur2=>event2),
            . . .)
```

where (x=>y) denotes an association or tuple with key x and value y. The durations that are the keys of these associations can be thought of as relative delays between the start of the enclosing event list, and the start of the events with which the delays are associated. A duration key with value zero means that the related event starts when the enclosing event list starts. In the case that dur1 and dur2 in the example above are equal, the two events will be simultaneous. If (dur1+event1 duration)=dur2, the two events will be sequential. Other cases, such as overlap or pause between the events, are also possible, depending on the values of the dur1 and dur2 variables and the durations of the events with which they are associated. There should be processing methods in the environment that support Smoke to remove 'gaps' between events, or to apply 'duty cycle' factors to event durations, for example to make a *staccato* performance style.

Figure 7 shows a simple example of the use of the duration keys in event list declarations; in it, a G major chord of three notes is followed by the first three steps of the G major scale in sequence.

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 behaviour of events, and special behaviours for mapping that are used by voices and event modifiers (see below). Event lists can be named, and when they are, they become persistent (until explicitly erased within a document or session).

```
EventList new
                "Add 3 simultaneous notes--a chord"
        add: (#g3 pitch, 1 beat) at: 0;
        add: (#b4 pitch, 1 beat) at: 0;
        add: (#d4 pitch, 1 beat) at: 0;
                "Then 3 notes in sequence after the chord"
        add: (#g3 pitch, 1 beat) at: 1 beat;
        add: (#a4 pitch, 1 beat) at: 2 beat;
        add: (#b4 pitch, 1 beat) at: 3 beat.
```

Figure 7. A mix of parallel and sequential events in an event list.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

The messages (anEventList add: anAssociation) and (anEventList add: anEventOr EventList 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 collection-style control structure messages such as (anEventList collect: aSelectionBlock) or (anEventList select: aSelectionBlock), though this requires the representation of contexts/closures. The behaviours 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 concrete (reified) 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 #via_script_14), is defined by applications and voices.

A named event list is created (and stored) in the first example in figure 8, and two event associations are added to it, one starting at 0 s, and the second one starting at 1 s. Note that the two events can have different types of properties, and the handy instance creation messages such as (dur: d pitch: p amp: a). The second example is the terse format for event list declaration using the concatenation message (,) – concatenation of events, or of (duration=>event) associations, is read as the declaration of an event list. One can use this style of shorthand as a terse way of creating the anonymous (non-persistent) list with three events shown in the second example in figure 8. The third example shows the first few notes from the C minor fugue from J. S. Bach's *The Well-Tempered Clavichord* in which the first note begins after a rest

```
"Event List creation the verbose way."
(EventList newNamed: #test1)
                "Add an event with musical note-like properties"
        add: 0 sec => (Event dur: 1/4 pitch: 'c3' ampl: 'mf');
                "Add an event with a sampled sound's name"
        add: 1 sec => (Event new dur: 6.0; ampl: 0.3772; sound: #s73bw)

"Terse Lists--concatenation of events or (dur => event) associations."
                "First event--simple"
(440 Hz, 1 beat, -9.7 dB),
                "Second event--with a phoneme name"
((1.396 sec, 0.714 ampl) phoneme: #xu),
                "Third event--a MIDI-style note after a pause"
(4 sec, 250 msec, 64 key, 86 vel).

"Bach WTC C-minor fugue theme."
"start time      duration   pitch        voice"
(1/2 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).
```

Figure 8. Simple event list examples.

(that could also be represented explicitly as an event with a duration and no other properties).

Note from the above examples that there is no model of time in Smoke; only relative durations are used. Absolute time is not represented explicitly because everything is thought to be 'relative'. An event is always contained within some event list, so its start time is described as the relative delay (duration) from the start of the enclosing event list to its own. Thus, a time model is not needed, and events have no notion of their own start times, meaning that the same event can be shared in several places by being 'pointed to' by several event lists at different relative delays.

4.6. Event generators and modifiers

The event generator and event modifier packages provide for music description and performance using generic or composition specific 'middle level' objects (Pope 1989). Event generators are used to represent the common structures of the musical vocabulary such as chords, ostinati, or compositional algorithms. Each event generator class knows how it is described (e.g. a chord has type, root, and inversion, or an ostinato has an event list and a repeat rate) and can perform itself once or repeatedly, acting like a function, a control structure, or a process, as appropriate.

Some event generators describe relationships among events in composite event lists (e.g. chords described in terms of a root and an inversion), while others describe melismatic embellishments of, or processes on, a note or collection of notes (e.g. mordents). Still others are descriptions of event lists in terms of their parameters (e.g. ostinati). Most standard examples (chords, ostinati, rolls, etc.) above can be implemented in a simple set of event generator classes; the challenge is to make an easily extensible framework for composers whose compositional process will often extend the event generator hierarchy.

All event generators can either return an event list, or they can behave like processes, and be told to play or to stop playing. We view this dichotomy, between views of event generators as functions vs event generators as processes, as a part of the domain, and differentiate on the basis of the musical abstractions. It might, for example, be appropriate to view an ostinato as a process (and send it messages such as start and stop), or to ask it to play three times.

4.6.1. Shared behaviour of event generators

Every event generator class has behaviours for the creation of instances, including, normally, some terse description formats (see the examples below). All event generator instances must provide a way of returning their events – the eventList method.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

Process-like event generators such as ostinati also provide start/stop methods for their repeated performance. The examples in figure 9 illustrate other event generator behaviours.

4.6.2. Abstract classes: cluster, cloud, and ostinato

There are three abstract classes of event generators: *cluster*, *cloud*, and *ostinato*. An instance of a cluster class is a description of the pitch set or a group of simultaneous or repeated events. Examples of concrete cluster classes are *chords* and *rolls*. A cloud is a pitch/time surface described in terms of its data sets or stochastic selection criteria. Ostinato objects describe the various ways of controlling the repetition of an event list.

The examples given in figure 9 demonstrate the use of event generators and event modifiers. Figure 10 shows the output of two of the cloud examples from figure 9(b) as represented in Hauer–Steffens notation. This is similar to piano roll notation – lines in the staff correspond to accidentals, and the note heads map onto the 'voice' used. The lines extending out from the note head are used to signify notes' durations (horizontal lines) and amplitudes (vertical lines).

The basic set of event generators is illustrated in figure 9; there are several extensions in the standard MODE release, and a number of others in current usage. This begs the question of 'how far to go'. Are complex artificial-intelligence-based compositional expert systems themselves simply event generators with very complex selection criteria?

4.6.3. Event modifiers and operators

Event modifier objects hold onto a function of one variable and a property name; they can be told to apply their functions to any property of an event list eagerly (at creation time) or lazily (at performance time). Functions of one or more variables (see below) can be described in a number of ways, including linear, exponential, or cubic spline interpolation between breakpoints. The examples in figure 11 illustrate the simple use of event modifiers. Figure 12 shows two simple functions: a linear ramp function from 0 to 1, and a spline curve that moves around the value 1.

4.7. Functions, probability distributions, and sounds

Smoke also defines several types of discrete or continuous probability distributions, and granular and

(a)

```
"Create a pentatonic cluster."
(Cluster dur: 2.0 sec
        pitches: #(c3 d3 e3 g3 a4)
        ampl: #mp
        voice: #piano)

"Create and manipulate a chord."
| aChord |                          "Declaration (optional)"
        "Create a C-Major tetrad in the root position lasting 1/2 beat"
aChord := Chord majorTetradOn: 'c3' inversion: 0 duration: 1/2.
aChord ampl: #mf.                 "Make it mezzoforte"
aChord eventList.                 "Answers an event list with the chord's 4 not

"A Simple Progression--C Major full cadence (with terrible voice-leading)."
(EventList newNamed: #progression1)
        add: (Chord majorTetradOn: 'c3' inversion: 0 duration: 1/2);
        add: (Chord majorTetradOn: 'f2' inversion: 2 duration: 1/2);
        add: (Chord majorTetradOn: 'g2' inversion: 1 duration: 1/2);
        add: (Chord majorTetradOn: 'c3' inversion: 0 duration: 1/2).

"Arpeggiate the Chord"
Arpeggio on: aChord
        delay: 100        "give a delay for arpeggiation (msec)"
        ampl: 'f'                 "set the amplitude"

"Simple rolls and trills"
Roll length: 2 sec        "duration of roll"
        delay: 50         "delay between notes"
        note: #f2         "pitch = f2"
        ampl: #ff.        "amplitude"

"The unit messages here can be left out as they are assumed."
Trill length: 1           "1.0 sec duration"
        delay: 80         "80 msec per note"
        notes: #(c d)     "give the array of pitches"
        ampl: 100.        "loudness = MIDI velocity 100"
```

(b)

```
"Create a stochastic cloud with values taken from the given ranges"
(Cloud dur: 4                     "duration in seconds"
        pitch: (60 to: 69)        "pitch range--an interval (C to A)"
        ampl: (80 to: 120)        "amplitude range--an interval"
        voice: (1 to: 4)          "voice range--an interval"
        density: 10)              "density in notes-per-sec"
"The result is shown in Hauer-Steffens notation in Figure 10a."

"Create a selection cloud with values from the given data sets."
(SelectionCloud dur: 2            "duration in seconds"
        pitch: #(c d f)           "select from this pitch array"
        ampl: #(mf mp pp)         "and this array of amplitudes"
        voice: #(viola)           "and this voice"
        density: 16)              "play 16 notes-per-sec"

"Make a low rising crescendo cloud."
(DynamicCloud dur: 6 sec
                        "starting and ending pitch ranges"
                        "i.e., focus onto one note"
        pitch: #((30 to: 44) (60 to: 60))
                        "amplitude ranges--crescendo"
        ampl: #((40 to: 60) (80 to: 120))
        voice: 1                "static voice"
        density: 20)    "20 notes-per-sec"

"Make a textured crescendo roll."
(DynamicCloud dur: 12
        pitch: (20 to: 20)      "fixed pitch"
                        "crescendo with a small random jitter"
        ampl: #((26 to: 30) (114 to: 120))
        voice: (1 to: 1)        "fixed voice"
        density: 20)            "20 notes-per-sec"

"Make a transition between two chords."
(DynamicSelectionCloud dur: 5
                        "starting and ending pitch sets"
        pitch: #( #(57 59 60) #(67 69 72 74))
        ampl: #(30 40 60)       "static amplitude set"
        voice: #(1 3)           "and voice set"
        density: 16)            "20 notes-per-sec"
"The result is shown in Hauer-Steffens notation in Figure 10b."
```

(c)

```
"Set up an Ostinato on the named scale."
(Ostinato named: #scale2 onList: (EventList named: #scale1)).
                "Turn it on."
(Ostinato named: #scale2) play.
                "Stop it."
(Ostinato named: #scale2) stop.
                "Play it 5 times."
(Ostinato named: #scale2) play: 5.
```

Figure 9. Event generator description examples: (a) clusters, (b) clouds, and (c) Ostinati.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

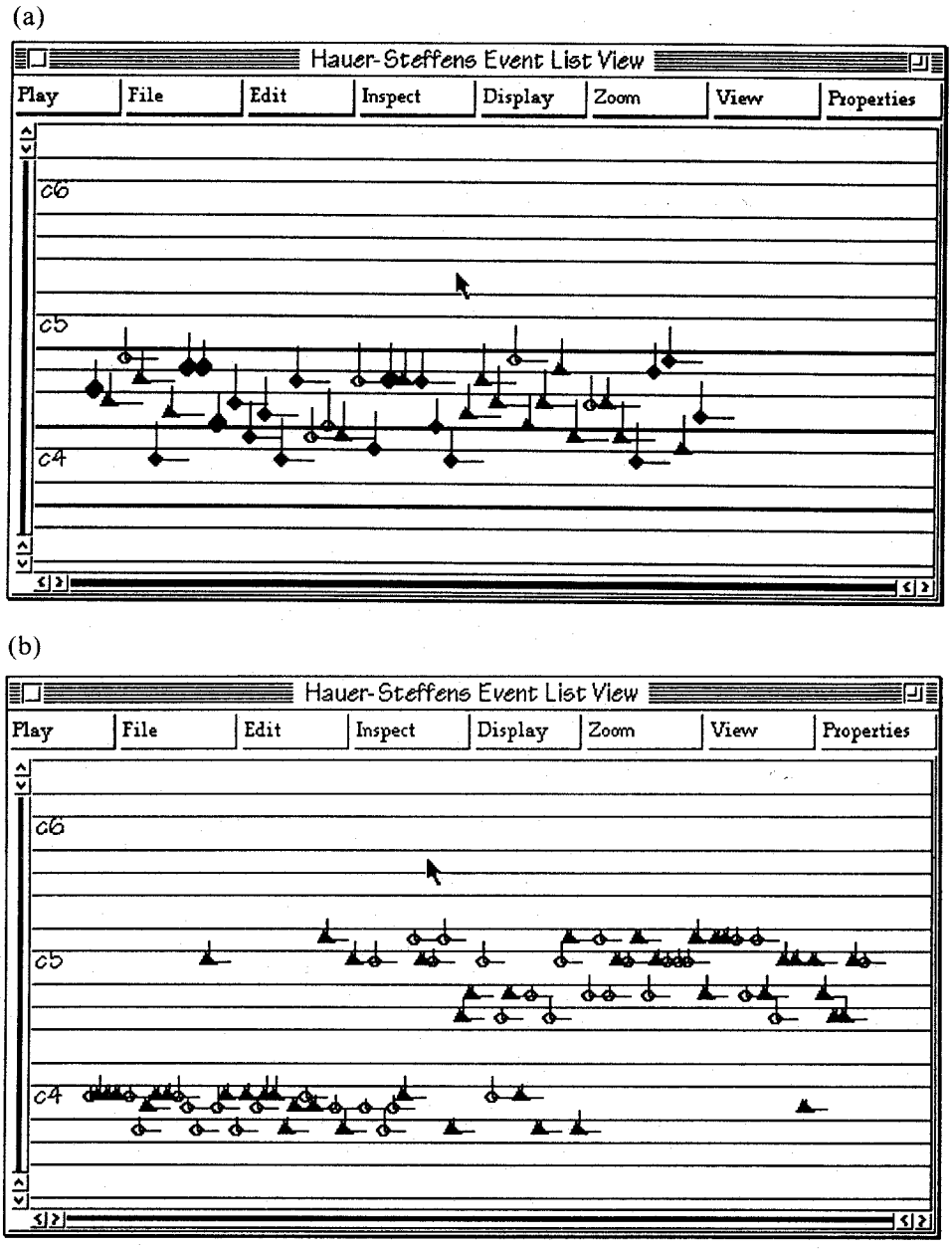


Figure 10. Output of example clouds in Hauer–Steffens notation.

sampled sounds. The description of these facilities is, however, outside the scope of this paper, and the reader is referred to Pope (1992, 1993, 1995).

4.8. 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 *structure accessor* 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 (e.g. 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, 1993).

The required voices include MIDI I/O (both real-time and file-based), Music-V-style note-lists (for the

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

```
"Apply a crescendo to an event list using a linear ramp function."
        "Declaration of temporary variables"
| roll fcn |
        "Create a roll 4 sec long with 20 notes-per-sec."
roll := (Roll length: 4.0 sec delay: 50 msec
                note: #d pitch ampl: #mf ampl) eventList.
        "Play it"
roll play.
        "Create a linear function that ramps from 0 to 1 over its extent."
        "An x/y plot of this function is shown in Figure 12a."
fcn := LinearFunction from: #((0@0) (1@1)).
        "Apply it immediately to the loudness of the roll."
roll apply: fcn to: #loudness.
        "Play the roll again."
roll play.

"Playing 'molto rubato'."
| roll fcn |
        "Create a roll using the assumed types of sec, msec, key, velocity"
roll := (Roll length: 4.0 delay: 80 note: 36 ampl: 100) eventList.
        "Apply a spline function to the start times."
roll apply:
        "An x/y plot of this function is shown in Figure 12b."
                (SplineFunction from: #((0@1) (0.35@1.2) (0.8@0.4) (1@1)))
                to: #start.
        "Play the roll."
roll play.
```

Figure 11. Event modifier examples.

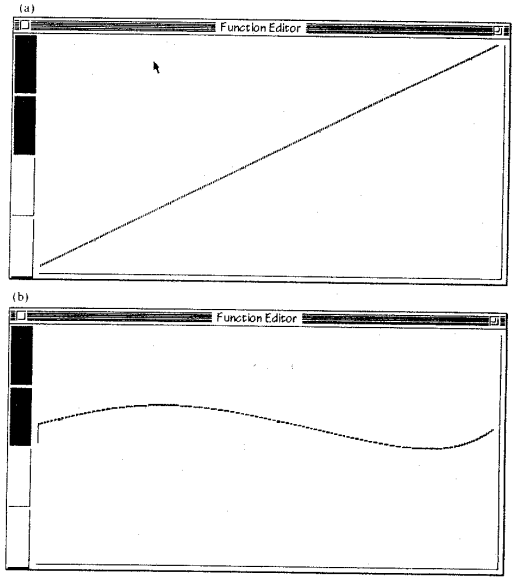


Figure 12. Linear and spline functions.

Cmix, cmusic, and Csound formats), and real-time sound output; others are optional. Figure 13 shows the desired usage of voices in shielding the user from the details of any particular output format. In this case, an event list is created and then played on both MIDI and Cmix output voices in turn.

4.9. Smoke score example

In Smoke scores, sections with declarations of variables, naming of event lists, event definition, functions and event modifiers, and annotational links and data, can be freely interspersed. Note that one tries to avoid actually ever typing Smoke at all, leaving that to interactive graphical editors, algorithmic generation or manipulation programmes, or read/write interfaces to other media, such as MIDI. The example given in figure 14 shows the components of a Smoke score for a composition with several sections declared in different styles.

5. CONCLUSIONS

The Smalltalk music 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 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 Smoke in the MODE system, are freely available via Internet file transfer (see Pope (1993, 1995) for details).

APPENDIX 1. READING SMALLTALK

Smalltalk is an object-oriented message-passing programming language where expressions are built to be read as English sentences. Names and identifiers in Smalltalk are separated by white space and are often composed of a concatenated noun phrase written with embedded uppercase letters, e.g. anEvent or MusicMagnitude. By convention, shared or global objects (such as class names) are written with the first letter capitalised (e.g. MusicMagnitude, AbstractEvent or HertzPitch), while private or temporary objects are written lower-case (e.g. anEventInstance or thePosition).

In a Smalltalk expression, the noun (i.e. the receiver object) comes first, followed by the message; for example, to ask the size of an array, one sends it the message size, as in the expression (anArray size), which answers an integer object. This kind of message (without any arguments) is called *unary*.

If the message has any arguments, they are separated from the message keywords in that the keywords all end in ':', so, for example, to index the first element of the array, one writes (anArray at: 1); to set the first element to zero, use (anArray at: 1 put: 0). Note the use of multiple keywords (at: and put:), and the fact that the order is always to alternate keywords and arguments. This means that if you choose your keyword names well, even rather complex messages with many arguments are quite readable.

There is a special case of keyword messages that is used for arithmetic and logical operations: binary messages. In this case, one can leave the colon off of the message keyword allowing one, for example, to write (3+4) or (x>0) as legal Smalltalk. There are several other binary messages that are widely used, among them '@' for creating points (the expression 3@4 answers a point whose $x = 3$ and

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

whose $y = 4$) and '–>' for creating associations (the expression key–>value 'associates' the given key with the given value).

Expressions can be nested using parentheses (i.e. evaluate what is inside the innermost parentheses first), and fancy expression constructs and control structures are possible. Normally, message expressions are separated by periods (.), but multiple messages sent to the same receiver object may be separated by semi-colons (;), in which case the receiver need not be repeated.

Double-quotes delineate comments in Smalltalk (e.g. "This is a comment."); single-quotes are used for immediate string objects (e.g. 'This is a string.'). Names for temporary variables are declared between vertical bars (e.g. |varName1 varName2|). Symbols are special strings that are stored in a table so as to be unique; they are written with the hash-mark '#' as in #blue, meaning 'the symbol blue'.

Smalltalk supports deferred execution in the form of closures or anonymous functions called *blocks*; they are written in square brackets '[. . .]' and can have arguments and/or local temporary variables. The up-arrow or caret (^) is used to return values (objects) from within blocks. For example, a block that takes two arguments and answers their sum would look like

```
["arguments" :x :y|"body" ^x+y].
```

There are many more features of the language that are of interest to anyone concerned with programming in Smalltalk (e.g. how control structures work), but they are beyond the scope of the current discussion.

Smalltalk programs are organised as the behaviours (methods) of classes of objects. To programme a graphical application, for example, one might start by adding new methods to the point and 3D point classes for graphical transformations, and build a family (a class hierarchy) of display objects that know how to present themselves in interesting ways. Classes are described as being abstract or concrete depending on whether they are meant as models for refinement within a framework (and not be instantiated), or for reuse 'off the shelf' as in the elements in a tool kit.

Inheritance and polymorphism mean that one reads Smalltalk programs by learning the basic protocol (messages/methods) of the abstract classes first; this gives one a feel for the basic behaviours of the system's objects and applications.

```
"Create some notes in an event list"
| list cmVoice |
list := Trill length: 1 delay: 80 notes: #(c d  g) ampl: #mp.

        "Play it on a MIDI synthesizer."
list voice: (MIDIVoice on: (MIDIDevice new) channel: 1).
list play.

        "Create a Cmix-format note-list voice for the given file."
cmVoice := CmixVoice on: ('test.out' asFilename writeStream).
        "Tell it how to map event properties to note command fields."
cmVoice map: #ampl  toField: 4.
        "Play the event list onto the file."
list playOn: cmVoice on: aFileStream.
```

Figure 13: Voice usage example.

```
"Declarations of variable names and top-level event list."
| piece section1 section2 |
        "name declarations--optional but advised."
piece := EventList newNamed: #Opus1.

        "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.
                "i.e., it starts with a chord."

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

"section 2--terse, add event associations using ',' for concatenation."
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 with event data taken from the given arrays."
piece add: (EventList
        durations: #(250 270 230 120 260 ... )   "duration data array"
        loudnesses: #(mp)                        "loudness is all mp"
        pitches: #(c3 d e g ... )).                      "pitch value array"

"Add an event with the given samples (the very-low-level!)"
piece add: (Event rate: 44100 channels: 1
                samples: #(0 121 184 327 441 ... )) at: 34 sec.

"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."
```

Figure 14: Extended score example.

APPENDIX 2. MODE AND SMOKE AVAILABILITY

The definitive implementation of the Smoke music representation language (well, the only one to date) is written in the ParcPlace-Digitalk dialect of Smalltalk-80, but it is portable to other dialects of Smalltalk such as IBM's or GNU Smalltalk. It is included in the Musical Object Development Environment (MODE), and serves as the basis of the MODE's applications. The MODE, version 2, is available by Internet ftp from ccrma-ftp.stanford.edu/pub/st80 and via the World-wide Web from the URL http://www.cnmat.berkeley.edu/~stp/MODE.

ACKNOWLEDGEMENTS

Smoke, and the MODE of which it is a part, is the work of many people. Craig Lata 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. Many others have contributed to the MODE environment.

REFERENCES

Barstow, D., Shrobe, H., and Sandewall, E. 1984. *Interactive Programming Environments*. New York: McGraw-Hill.

Dannenberg, R. B. 1989. The Canon score language. *Computer Music Journal* **13**(1): 47–56.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press

Dannenberg, R. B. 1993. Music representation issues, techniques, and systems. *Computer Music Journal* **17**(3): 20–30.

Dannenberg, R. B., Dyer, L., Garnett, G. E., Pope, S. T., and Roads, C. 1989. Position papers for a panel on music representation. *Proceedings of the 1989 International Computer Music Conference.* San Francisco: International Computer Music Association.

Deutsch, L. P., and Taft, E. A. 1980. *Requirements for an experimental programming environment*. Research Report CSL-80-10. Palo Alto, CA: Xerox PARC.

Goldberg, A., and Pope, S. T. 1989. Object-oriented is not enough! *American Programmer: Ed Yourdon's Software Journal* **2**(7): 46–59.

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

Hewlett-Packard. 1994. *Hewlett-Packard Distributed Smalltalk Release 2.0 Data Sheet*. Palo Alto, CA: Hewlett-Packard Company.

Jaffe, D., and Boynton, L. 1989. An overview of the Sound and Music Kits for the NeXT computer. *Computer Music Journal* **13**(2): 48–55. Reprinted in S. T. Pope (ed.) 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Cambridge, MA: MIT Press.

Krasner, G. 1980. Machine Tongues VIII: the design of a Smalltalk music system. *Computer Music Journal* **4**(4): 4–22. Reprinted in S. T. Pope (ed.) 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press.

Layer, D. K., and Richardson, C. 1991. Lisp systems in the 1990s. *Communications of the ACM* **34**(9): 48–57.

Oppenheim, D. 1989. DMix: an environment for composition. *Proceedings of the 1989 International Computer Music Conference*, pp. 226–33. San Francisco: International Computer Music Association.

Pope, S. T. 1989. Modelling musical structures as eventgenerators. *Proceedings of the 1989 International Computer Music Conference*. San Francisco: International Computer Music Association.

Pope, S. T. 1992. The Smoke music representation, description language, and interchange format. *Proceedings of the 1992 International Computer Music Conference*. San Francisco: International Computer Music Association.

Pope, S. T. 1993. The Interim Dynapiano: an integrated computer tool and instrument for composers. *Computer Music Journal* **16**(3): 73–91.

Pope, S. T. 1994. Letter to the Editors. *International Computer Music Association Array* **14**(1): 2–3.

Pope, S. T. 1995. *The Musical Object Development Environment Version 2 Software Release*. Source code and documentation files available from the Internet server ccrma-ftp-stanford.edu in the directory pub/st80.

Rodet, X., and Cointe, P. 1984. FORMES: composition and scheduling of processes. *Computer Music Journal* **8**(3): 32–50. Reprinted in S. T. Pope (ed.) 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press.

Scaletti, C. K. 1989. *Computer Music Journal* **13**(2): 23–38. Reprinted in S. T. Pope (ed.) 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press.

Taube, H. 1991. Common Music: a music composition language in Common Lisp and CLOS. *Computer Music Journal* **15**(2): 21–32.

Wegner, P. 1987. Dimensions of object-based language design. *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems*, *Languages*, *and Applications* (*OOPSLA*), pp. 168–82. New York: ACM Press.

Wiggins, G., Miranda, E., Smaill, A., and Harris, M. 1993. A framework for the evaluation of music representation systems. *Computer Music Journal* **17**(3): 31–42.

https://doi.org/10.1017/S1355771896000180 Published online by Cambridge University Press